Towards Lifelong Reasoning with Sparse and Compressive Memory Systems

Jack William Rae

PhD Thesis

CoMPLEX

University College London

Declaration

I, Jack William Rae confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

Humans have a remarkable ability to remember information over long time horizons. When reading a book, we build up a compressed representation of the past narrative, such as the characters and events that have built up the story so far. We can do this even if they are separated by thousands of words from the current text, or long stretches of time between readings. During our life, we build up and retain memories that tell us where we live, what we have experienced, and who we are.

Adding memory to artificial neural networks has been transformative in machine learning, allowing models to extract structure from temporal data, and more accurately model the future. However the capacity for long-range reasoning in current memory-augmented neural networks is considerably limited, in comparison to humans, despite the access to powerful modern computers.

This thesis explores two prominent approaches towards scaling artificial memories to lifelong capacity: sparse access and compressive memory structures. With sparse access, the inspection, retrieval, and updating of only a very small subset of pertinent memory is considered. It is found that sparse memory access is beneficial for learning, allowing for improved data-efficiency and improved general-

isation. From a computational perspective — sparsity allows scaling to memories with millions of entities on a simple CPU-based machine. It is shown that memory systems that compress the past to a smaller set of representations reduce redundancy and can speed up the learning of rare classes and improve upon classical data-structures in database systems. Compressive memory architectures are also devised for sequence prediction tasks and are observed to significantly increase the state-of-the-art in modelling natural language.

Impact Statement

This study investigates memory structures for artificial neural networks that can scale to longer periods of time and larger quantities of data than previously demonstrated possible. It is demonstrated that neural networks can learn to store and retrieve salient information over hundreds of thousands of time-steps — an improvement of around two orders of magnitude from prior work. This is achieved using efficient sparse memory access operations and compressive memory data structures to reduce the computational cost of accessing memory and the amount of space required to store it.

The long-term goal for this work is to embed these memory structures within a complex agent, so it can reason over the stream of data it experiences during its lifetime. This could be a robotic agent which acts in the real world, or a dialogue agent which can reason over past conversations and auxiliary text context (e.g. the web).

This study demonstrates that better long-range memory architectures can be used to improve state-of-the-art performance in question answering (Chapter 3), database systems (Chapter 5), and the modelling of language and speech (Chapter 4 and 6).

Acknowledgements

I dedicate this work to my wife Laura and to my son William. I would like to thank my family and friends for their support throughout this project, in particular my mother who has given considerable guidance along the way.

I would like to thank my supervisor Tim Lillicrap for both introducing me to the topics of scalable and compressive memory systems, and for guiding me through the research process over the past couple of years. His consistent energy and creativity has propelled me through many patches of uncertainty and it has been a wonderful experience to work with him. I would also like to thank my supervisor Peter Dayan for his wider perspective spanning multiple fields. Peter's detailed, analytical process of thinking and questioning has shaped the structure of my research. I would like to thank Yori Zwols, my manager throughout the majority of my doctoral studies. Yori helped me to select research projects that suited the self-contained nature of doctoral study, whilst being relevant for DeepMind's mission of solving intelligence.

I would like to thank Alex Graves and Greg Wayne for their mentorship. Alex has introduced me to many ideas in the space of memory and sequence modelling. He has also taught me how to write, indirectly via his own works and directly via

his astute feedback. Greg has taught me to select projects that can be tentatively validated or disproved within a few weeks of work, which has proven to be one of the most important pieces of advice in the uncertain game of research. I would like to thank Demis Hassabis and Joel Liebo for initially introducing me to memory at DeepMind in 2014 and piquing my interest from their own innovative neuroscience-based memory research. Several research leads at DeepMind have overseen these research projects at DeepMind, providing resources and guidance — namely Daan Wierstraa, Oriol Vinyals, and Koray Kavukcuoglu.

For work on sparse attention, I would like to thank Jonathan Hunt who collaborated closely on this project. Jonny provided ideas for sparsifying the DNC's link matrix and instrumented the bAbI evaluation. I would also like to thank Malcolm Reynolds, Fumin Wang and Yori Zwols for their advice and comments.

For work on compressive memory systems for classification, I would like to thank Chris Dyer and Gabor Melis for providing invaluable direction in getting set up with language modelling. I would also like to thank Oriol Vinyals, Pablo Sprechmann, Siddhant Jayakumar, Charles Blundell and Adam Santoro for a multitude of enlightening comments and suggestions.

For work on the Neural Bloom Filter, I would like to thank my collaborator Sergey Bartunov who implemented the database experiments and had several architectural contributions. I thank Andras Gyorgy and Tor Lattimor for their input on the general space complexity of approximate set membership, from an information theoretic perspective.

For work on the Compressive Transformer I would like to thank my collaborator

Anna Potapenko who ran several experiments and developed the PG-19 dataset. I would also like to thank Siddhant Jayakumar for advice and experiments on the memory maze tasks, and Chloe Hillier for management of the research project, and help in open-sourcing the model code and dataset. I thank Tim Lillicrap for his numerous inputs on model design, experiment design and inputs on the manuscript. I thank Chris Dyer, Felix Gimeno, and Koray Kavukcuoglu for reviewing the text. I also thank Peter Dayan, Adam Santoro, Jacob Menick, Emilio Parisotto, Hyunjik Kim, Simon Osindero, Sergey Bartunov, David Raposo, and Daan Wierstra for ideas regarding model design. I thank Yazhe Li and Aaron Van de Oord for their help and advice in instrumenting speech modelling experiments.

Finally I would like to thank the long list of unnamed DeepMind colleagues and UCL staff that have provided library code, talks, discussions, feedback, and endless enthusiasm and inspiration.

Contents

List of Figures

List of Publications

Below is a list of publications related to the topic of memory-augmented neural networks that I have authored or co-authored during the doctoral program. The publications highlighted in bold form a substantial part of the thesis.

1. **Jack W. Rae, Jonathan J. Hunt, Ivo Danihelka, Timothy Harley, Andrew W. Senior, Gregory Wayne, Alex Graves, and Timothy Lillicrap. "Scaling memory-augmented neural networks with sparse reads and writes." In Advances in Neural Information Processing Systems. 2016.**

2. Charles Blundell, Benigno Uria, Alexander Pritzel, Yazhe Li, Avraham Ruderman, Joel Z. Leibo, Jack W. Rae, Daan Wierstra, and Demis Hassabis. "Model-free episodic control." arXiv preprint arXiv:1606.04460. 2016.

3. Andrew Trask, Felix Hill, Scott E. Reed, Jack W. Rae, Chris Dyer, and Phil Blunsom. "Neural arithmetic logic units." In Advances in Neural Information Processing Systems. 2018.

4. Adam Santoro, Ryan Faulkner, David Raposo, Jack W. Rae, Mike Chrzanowski, Theophane Weber, Daan Wierstra, Oriol Vinyals, Razvan Pascanu, and Tim-

othy Lillicrap. “Relational recurrent neural networks.” In Advances in Neural Information Processing Systems. 2018.

5. Pablo Sprechmann, Siddhant M. Jayakumar, Jack W. Rae, Alexander Pritzel, Adria Puigdomenech Badia, Benigno Uria, Oriol Vinyals, Demis Hassabis, Razvan Pascanu, and Charles Blundell. “Memory-based Parameter Adaptation.” In International Conference on Learning Representations. 2018.

6. Greg Wayne, Chia-Chun Hung, David Amos, Mehdi Mirza, Arun Ahuja, Agnieszka Grabska-Barwinska, Jack W. Rae et al. “Unsupervised predictive memory in a goal-directed agent.” arXiv preprint arXiv:1803.10760. 2018.

7. **Jack W. Rae, Chris Dyer, Peter Dayan, and Timothy Lillicrap. “Fast Parametric Learning with Activation Memorization.” In 35th International Conference on Machine Learning (ICML 2018), pp. 4228-4237. International Machine Learning Society. 2018.**

8. Nenad Tomasev, Xavier Glorot, Jack W. Rae, Michal Zielinski, Harry Askham, Andre Saraiva, Anne Mottram et al. “A clinically applicable approach to continuous prediction of future acute kidney injury.” Nature 572, no. 7767. 2019.

9. **Jack W. Rae, Sergey Bartunov, and Timothy Lillicrap. “Meta-Learning Neural Bloom Filters.” In International Conference on Machine Learning, 2019.**

10. Cyprien de Masson d’Autume, Mihaela Rosca, Jack W. Rae, and Shakir Mohamed. “Training language GANs from Scratch.” In Advances in Neural Information Processing Systems. 2019.

11. Francis Song, Abbas Abdolmaleki, Jost Tobias Springenberg, Aidan Clark, Hubert Soyer, Jack W. Rae, Seb Noury et al. "V-MPO: On-Policy Maximum a Posteriori Policy Optimization for Discrete and Continuous Control." In International Conference on Learning Representations. 2020.

12. Siddhant M. Jayakumar, Jacob Menick, Wojtek Czarnecki, Jonathan Schwarz, Jack W. Rae, Simon Osindero et al. "Multiplicative Interactions and Where to Find Them." In International Conference on Learning Representations. 2020.

13. Sergey Bartunov, Jack W. Rae, Simon Osindero, and Timothy P. Lillicrap. "Meta-Learning Deep Energy-Based Memory Models." In International Conference on Learning Representations. 2020.

14. **Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, and Timothy P. Lillicrap. "Compressive Transformers for Long-Range Sequence Modelling." In International Conference on Learning Representations. 2020.**

15. Jack W. Rae. "Do Transformers Need Deep Long-Range Memory?" In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. 2020.

16. Emilio H. Parisotto, Francis Song, Jack W. Rae, Razvan Pascanu, Caglar Gulcehre, Siddhant M. Jayakumar, Max Jaderberg et al. "Stabilizing Transformers for Reinforcement Learning." In International Conference on Machine Learning. 2020.

Chapter 1

Introduction

Memory-augmented artificial neural networks have risen in prominence over recent years as a general solution to sequence modelling tasks. By writing and attending to an external memory, neural networks can learn to correlate information across time and make accurate predictions of the future — from predicting the next word in a sequence of text, to predicting the early onset of patient deterioration in a stream of healthcare data.

However the current memory systems incorporated by neural networks are still relatively limited in contrast to the brain's memory circuitry. Where a human is able to read a book and recall relevant pieces of information from its entirety, a memory-augmented neural network processing text is currently limited to a few thousand words of context. During day-to-day life, humans enjoy the ability to recall salient information over a large range of timescales: hours, weeks, years, and even decades. For artificial intelligence to master the seemingly simple tasks of daily life: recalling the location of the car keys, the route to work, the names of

colleagues and the set of outstanding to-do items; it must be able to reason over memories of comparably large time horizons. We say *reason*, because an agent must be able to use its memory as part of a learning process for many downstream tasks. A distributed computer database may be able to store a lifetime's worth of sensory inputs, and it may be performant at recalling specific pieces of information; however it may be brittle under uncertainty, slow to update existing information, poor at registering correlations over time, or require a query interface which is difficult to learn. An agent needs to not only *recall* information from a wide range of timescales, but learn to reason over it.

Existing memory-augmented neural networks are powerful temporal learning machines however they have a limited range of attention for two principal reasons. Firstly they require a large compute budget per unit of memory: executing queries requires a linear or quadratic number of floating-point operations (FLOPS) as a function of memory capacity. Secondly they struggle to compress information over time at scale, and thus contain a large amount of redundant information. These points are elaborated on in the next section.

We aim to bring neural networks closer to being able to reason over a lifetime's worth of experience. We pursue two general approaches towards this, sparse memory access and compressed memory representations. The former emphasizes efficient interaction with memory, the latter emphasizes increased memory capacity and improved reasoning. We develop several novel memory architectures and continually evaluate them on natural-data tasks such as question answering and language modelling.

We emphasize scalable memory systems under which we can learn effectively, and

we provide analysis of the theoretical and empirical run-time performance of these systems to ensure they are performant. In several well-studied benchmarks, we obtain state-of-the-art performance across the field with improved memory systems — adding to the growing set of findings that better memory systems can result in better temporal reasoning.

1.1 Limitations of Existing Approaches

We remark that one of the greatest drawbacks to current memory systems is their limited capacity, despite the abundance of computing power that is available in the present day. We explain why this is the case with a brief history of neural networks and memory.

Early memory systems in artificial neural networks were formalized as recurrent neural networks (Rumelhart et al., 1986) which took inspiration from the brain's recurrent circuitry. Here, a recurrent feedback of activations (or recurrent 'state'), which are manipulated and re-circulated over time-steps, serve as the stored memory. In the field of machine learning, recurrent neural networks are formalized as a learnable function approximator which receives the current timestep's input alongside the previous recurrent state, and transforms this to an output and an updated state which will be passed to the next time-step.

Recurrent neural networks, of which the LSTM (Hochreiter and Schmidhuber, 1997) and GRU (Cho et al., 2014) are the most ubiquitous examples, have displayed a remarkable ability to make effective use of the recurrent connections to store, manipulate, and recall information (Sutskever et al., 2014). These models contain

learnable weights to influence how each unit of the recurrent state is updated as a function of every other unit of the recurrent state and the current input. This allows the memory to be highly expressive, but it results in the number of trainable parameters growing quadratically with the size of the recurrent state. This means the size of the recurrent state, and thus the memory capacity of the model, cannot be very large. Some attempts to remedy this have included making the recurrent weight matrix low-rank (Sak et al., 2014; Jozefowicz et al., 2016) or sparse (Narang et al., 2017), however this has only facilitated up to $2-4\times$ memory size increase. A more successful approach is to augment the neural network with an external memory.

Memory-augmented neural networks allow the network to interact with an external memory matrix which can grow in capacity without requiring any additional network parameters. This is done by making use of parameter-free write and read operations which dictate how information flows to and from the memory. These operations do not contain learnable parameters, but they are defined to be differentiable, and thus a neural network can learn to control them by shaping their inputs. This idea was originally pioneered by the Neural Turing Machine (Graves et al., 2014), Memory Networks (Weston et al., 2014), and attention-based translation models (Bahdanau et al., 2014) in 2014. These early architectures largely differ in how they write and update information to external memory, however interestingly they all use the same operation to read from memory: *content-based attention*.

Content-based attention can be thought of as a differentiable analogue to a database lookup. With a database lookup, one may select rows with a query which evaluates

the properties of several fields row-by-row, and then extract relevant fields for the matching set. Attention works by using a differentiable comparison function across each row in memory, to create a weight per row, and then linearly combining the values based upon this weight. Because the operation is differentiable, it can be placed within a neural network which is optimized via gradient descent.

However despite the considerably larger memory capacity of memory-augmented neural networks over recurrent neural networks, the expense of attention is considerably prohibitive if we wish to scale this approach to lifelong memory. Unlike a database, which can make use of logarithmic-time lookups via data indexing, attention compares a given query with every row in memory — meaning queries are linear-time. Furthermore, these memory-augmented neural networks do not tend to jointly represent information across rows in their memory matrices — which means it typically contains a multitude of repeated, redundant information. We aim to tackle these two issues by using efficient sparse access of memory with logarithmic time complexity, and more compressive memory structures that contain a more enriched representation of the past.

1.2 Overview of Contributions

In Chapter 3 we propose a sparse memory-augmented neural network called SAM (Sparse Access Memory) which uses sparse attention to read from memory in logarithmic time and a sparse write scheme to update memory in constant time. We show that it empirically can train up to hundreds of thousands of memory slots, and crucially show that learning is possible even in settings of extremely sparse attention. We also propose a more powerful but complex sparse memory

model called the Sparse Differentiable Neural Computer and apply it to the task of question-answering for the bAbI benchmark (Weston et al., 2014) where it outperformed all prior models.

In Chapter 4 we explore classification. Here memory has been successfully incorporated as an additional non-parametric classifier alongside a neural network classifier. The intuition is that memory can serve as a fast-learning mechanism to enhance the slow-learning parametric neural network. The memory often improves the modelling of rare or previously unseen classes within a limited horizon of attention. We consider replacing an external memory with a compressed memory that stores the cumulation of hidden activations for each given class. Such a system reduces the redundancy of storing many separate memories for frequently observed classes, and extends the model's memory of rare classes.

We then place this compressed memory *within* a subset of weights in the classifier, instead of storing it as a separate memory of recurrent activations, taking inspiration from the neuroscience literature of memory as a complementary learning system within the brain (McClelland et al., 1995). We show the memory update becomes an instance of the Hebbian learning update rule (Hebb, 1949), which is an early model of synaptic plasticity. The resulting architecture is a modified classification layer in a neural network which we call the *Hebbian Softmax*. We show this significantly improves the prediction of rare words for language modelling, achieving state-of-the-art (at time of study) on the long-range language modelling benchmark, WikiText-103 (Merity et al., 2016). We also show the prediction of newly observed classes is improved in the few-shot case.

In Chapter 5 we explore a scalable compressive memory which stores overlapping

representations across rows in memory. It takes inspiration from the overlapping representations in the CA3 of the brain, and the highly compressive data-structure which uses overlapping sparse binary codes to represent information — called the Bloom Filter (Bloom, 1970). We name our architecture the Neural Bloom Filter. We benchmark this for performance on a core memory task: familiarity. Specifically we inspect a number of natural-data settings of familiarity, and compare how much space each models' memory consumes at a fixed performance. We see recurrent neural networks tend to struggle to solve the tasks, but can be very space-efficient if they do. However Memory Networks can always solve the task but use a very large memory to do so. The Neural Bloom Filter is both easy to train, and compressive, and we remark of its potential utility in high-performance computing systems, such as databases.

Finally in Chapter 6 we combine the compression ideas from prior chapters to propose a state-of-the-art long range sequence model, the Compressive Transformer, which incorporates a mixture of granular short-term memory with a coarser-grained compressed memory of longer-term information. We show this outperforms the existing best-performing language models over wikipedia articles, and also book text. We also show significant performance benefit in the modelling of audio over strong baselines such as WaveNet (Oord et al., 2016), and in the domain of navigation within a reinforcement-learning agent. As such we bring the ideas of previous chapters, which largely are tested on synthetic memory tasks, to real-world problems with significant potential for impact.

The key idea of the Compressive Transformer is to introduce a learnable compression network which maps a collection of m consecutive memories to a smaller

set, e.g. m/c where c is a compression factor. We consider simple compression networks, from simple linear convolutions (which we find to perform best) to deep dilated convolutions, alongside baselines such as max and mean pooling. We show the model is able to make use of its long-range compressed memory, and in the case of language, better models words which require longer spans of dependency such as rare named entities. We also create a new language model benchmark derived from book data, called PG-19, to further advance research in the area of long-range sequence modelling.

We conclude with future directions, namely focusing on a combination of state-of-the-art sequence models with sparse compressive memories. We consider the incorporation of a more compressive memory architecture for the Compressive Transformer that uses optimization routines — such as energy-based optimization — to learn writing and reading rules. This is following recent work showing that energy-based models can be used within deep neural networks to facilitate associative memory tasks (Bartunov et al., 2020). We also remark at the growing popularity in incorporating sparse attention into memory-based models since the inception of this study, and the potential for advances in hardware that could bolster the widespread adoption of sparse attentive models.

Chapter 2

Background

To frame the scope of this study, we discuss the classifications of memory from the psychology literature including *episodic memory*. We outline the key known components to lifelong episodic memory in the human brain — notably the hippocampus — before moving on to the field of computer science for foundational data storage architectures and machine learning for a review of prominent memory-augmented machine learning models. Finally we discuss a selection of memory tasks which have driven the development of better memory architectures, and make the case for modelling natural language to be placed amongst them.

2.1 Memory in Animals

Memory is an umbrella term for many processes that occur in animals, machines, and even plants. In animals it is an integral aspect of learning, and quite often the distinction between learning and memorisation becomes difficult to assert.

Finding an association between correlated events from repeated trials is one of the mainstays of learning. Ivan Pavlov noted in the early 20th century in his classical conditioning experiments (Pavlov and Gantt, 1928) that his dog could elicit the same conditioned response (salivation) from observing an unconditioned stimulus (dog food) as it would elicit with a conditioned stimulus (the sound of a metronome) if the two events followed one another. This lead to the study of human and animal learning via behaviourism in psychology; however the initial finding is in itself a display of associative memory. The dog had the cognitive machinery to observe two events, the sound of the metronome and the subsequent delivery of food, and over several trials, associate them.

The ability to form associations may not be confined to animals. Recent studies with pea plants have shown that they can be classically conditioned to associate the direction of a fan emitting a breeze of air, placed at random within a circle surrounding the plant, with the direction of a corresponding light source (Gagliano et al., 2016). How then can we distinguish the role of memory from general associative learning, and define a reasonable set of categories for different types of long-term memories?

2.1.1 Tulving's Taxonomy for Long-Term Memories

Aristotle mused on the distinction of different types of long-term memories by considering the collections of experience of specific events in our lifetime and the resulting behavioural wisdom, or *phronesis* that arises therefrom (Ameriks and Clarke, 2000). That is, the forms of memories that we regard as life experience, versus the skills that we acquire. The search for a taxonomy for long-term memory

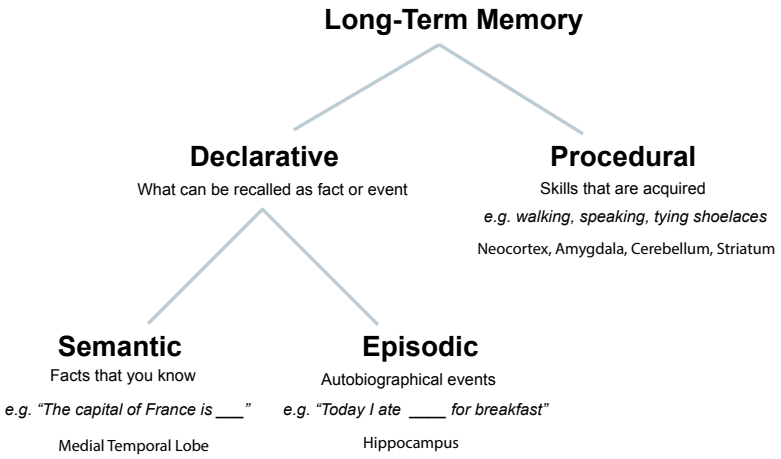


Figure 2.1: Tulving's classification of long-term memory (Tulving et al., 1972).

has undergone considerable debate, especially in the field of psychology during the twentieth century.

The psychologist Endel Tulving provided a modern and intuitive taxonomy in the 70s, that provided a more fine-grained classification than Aristotle's life experience versus skills (Tulving et al., 1972). This tree of classification is shown in Figure 2.1. Long-term memory could first be partitioned as either *procedural* or *declarative*. Procedural memories indicate skills, acquired via learning over many repetitions, such as learning to walk. Declarative memories indicated facts that can be asserted, such as the capital of France or the meal one chose for breakfast. Tulving crucially introduced a classification of declarative memories, as either *semantic* or *episodic* (Tulving et al., 1972; Tulving, 1985). Semantic memories could be described as those which one "knows" without any specific memory of when (such as the capital of France); episodic memories are autobiographical and may be associated with many other memories from the same time (e.g. what one ate for breakfast).

Different regions of the brain have a role to play in the formation of these different kinds of long-term memories. The neocortex is associated with procedural memories, and the hippocampus has been found to be crucial to the development of new episodic memories (Rolls, 2010). Famously when 'Patient H.M.' underwent a temporal lobectomy in 1953 to treat his severe epilepsy, which removed the majority of his hippocampus, he was still able to learn new skills (such as drawing a star from a mirrored view) but could not form new episodic memories (Corkin, 1968). Thus learning long-term skills can occur without memory.

We can not only draw analogies from Tulving's taxonomy to biological neural networks in the brain, but we can also draw analogies from Tulving's taxonomy to artificial neural networks. These analogies should be thought of as useful roadmaps for deep learning research, versus absolute truths — since we do not have a mechanism to probe exactly what a neural network 'knows it knows'.

We can think of the non-declarative procedural memories, and the declarative semantic memories, as being contained within the trained weights of an artificial neural network — learned slowly through backpropagation. For example if we train a neural network to perform next-word prediction over a corpus of text, it will implicitly learn many grammatical constructs within its trained weights — which the network cannot express directly in a single instance. These can be thought of as procedural memories. Furthermore if we provide a simple prompt "The capital of France is" then the model may return "Paris". This is a long-term association that can be declaratively expressed (and would not be considered episodic given there was no mention of Paris in the episode), so it roughly fits the description of a semantic memories. Unlike biological neural networks, we cannot

point to sub-regions of network weights where we would expect the procedural memories to be stored versus semantic — although this would be an interesting line of future research.

Episodic memories are not considered to be stored within the standard set of slow-moving model parameters that are learned via backpropagation. Instead these may lie in the hidden state of recurrent neural networks, in the memory of memory-augmented neural networks, or in a separate set of 'fast-moving' weights. We will introduce these architectures in Section 2.3, however roughly these are stores of past activations containing information from the past (*what*) along with *when* they occured. In this thesis we do not approach all forms of long-term memory as defined by Tulving. We specifically focus on building lifelong *episodic memory* systems. We represent, store, and recall salient memories that specify the *what* and *when* of observed events.

2.1.2 The Hippocampus

We have specified our scope for life-long episodic memory within artificial neural networks, and have mentioned this is principally in the brain via the Hippocampus. Thus we give a brief overview of this well-studied piece of neural circuitry that is present in humans and vertebrates, including rodents where its function is often probed.[1] The Hippocampus is a recurrent circuit of neurons, it integrates with the Entorhinal Cortex (EC) to receive input and feed its output (see Figure 2.2). Input activations pass through from lower layers in the EC (layer 2) to the Dentate

[1]For example, by monitoring the firing rate of a specific set of neurons within rodent hippocampi, it was observed that certain neurons' firing rates could be attributed to particular locations. This lead to the Nobel-prize-winning discovery of *place cells*, which serve as a spatial memory firing at specific observed landmarks (O'Keefe and Dostrovsky, 1971).

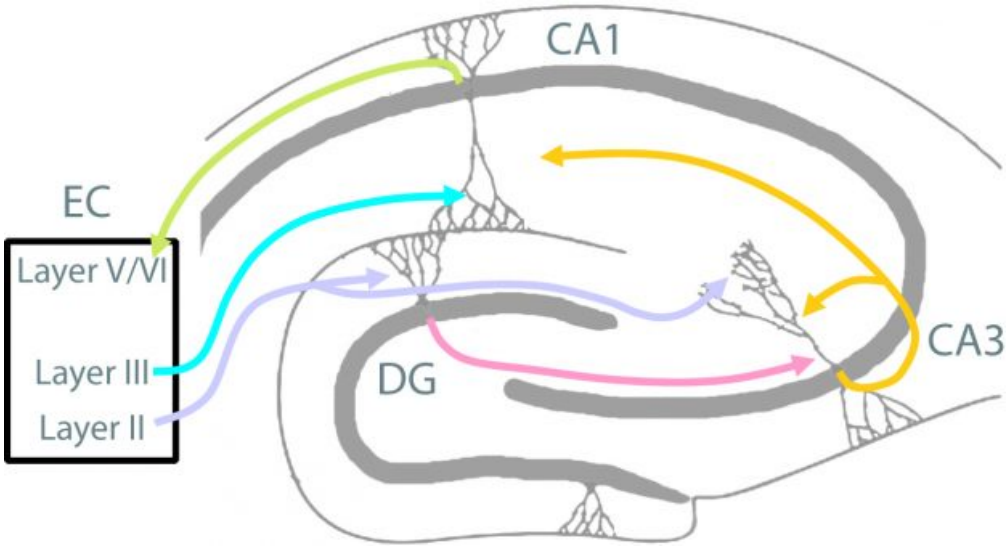


Figure 2.2: The hippocampo-entorhinal circuit, schematic from Diba (2018).

Gyrus (DG), which is believed to perform *pattern separation* to differentiate similar neural codes. The DG is one of the only known areas of the brain to facilitate adult neurogenesis, that is the continued generation of new neurons. These activation then pass to the CA3 along with another direct channel of inputs from the EC layer 2.

The CA3 contains recurrent connections and is a critical region for the storage of episodic memories. The excitation of CA3 neurons is sparse and overlapping, there is no clear compartmentalisation between regions of the CA3 for specific types of memories. The CA3 recirculates activations and the synapses contained within its pyramidal cells with interconnecting synapses that display long-term potentiation after a single instance of exposure (Bliss and Collingridge, 1993; Nakazawa et al., 2003). In essence the CA3 contains a one-shot writing associative writing mechanism into a set of fast-adapting weights.

Where the DG separates patterns, it is believed the auto-associativity of episodic

memories comes from the joint representation of information in the CA3. The CA3 then feeds activations to the CA1 which also has a direct connection with the input activations from the EC layer 3, and this results in an output fed to the upper layers (4 and 5) of the EC (see Shepherd (2004) for a thorough account of the hippocampal circuit). We conclude that the Hippocampus is one of the human mind's most essential tools for life-long episodic memory, and takes the form of a recurrent neural network. The Hippocampus sparsifies and separates its inputs to create sparse decorrelated codes, which it then stores jointly in an attractor-based optimisation.

2.1.3 Memory as a Complementary Learning System

Memory can allow us to learn tasks which require remembrance of the past, naturally, however it may also have a role to play in general learning. The role of the hippocampus appears to be not only as a gateway to memory formation, but as a facilitator of fast learning. The theory of *Complementary Learning Systems* (McClelland et al., 1995) posited the hippocampus acted as a fast learning mechanism to enhance the neocortex's slow but stable learning. In this paradigm the neocortex must update slowly with many decorrelated sensory inputs otherwise *catastrophic forgetting* may occur, where the acquisition of a new skill degrades the capability of a prior skill. The role of memory is to adapt quickly to each new task, and then to *consolidate* information to the neocortex, e.g. via the replaying of memories during sleep. Where the neocortex may require many examples to learn a new skill the hippocampus can allow us to adapt quickly, this is often termed **one-shot learning**. This is thought to be crucial to animal survival in adverse settings, for

example it is crucial to remember the sight and smell of a poisonous berry after consumption, such that it can be avoided thereafter. We explore memory as a means for fast-learning in Chapter 4.

2.2 Data Storage in Computing

The use of pattern separation and sparsity, alike to that of the Dentate Gyrus in the Hippocampus, are also fundamental tools in the efficient storage and retrieval in computing systems — notably hashing. In this section, we shall cover the use of hashing for exact and nearest-neighbour data retrieval in computing.

In the field of computer science there have been many advances in the construction of data structures which store and query data efficiently. Unlike in the brain, or in machine learning models, computer architectures can operate in an exact symbolic space where information is encoded in entirety at given memory locations, and queries can be executed for very specific attributes. Nevertheless data structures which support recalling stored information efficiently, or simply indicating whether a given piece of data has been previously observed, do incorporate sparse and distributed representations. We find some of these systems, such as fast search, can be used within neural networks (as demonstrated in Chapter 3) and others, such as the Bloom Filter, can be used as inspiration to design neural networks which outperform their classical counterparts (such as the Neural Bloom Filter, described in Chapter 5). In this section we give a brief overview of several core data structures that provide utility or inspiration to the architectures developed in this study.

2.2.1 Hashing

The simplest desiderata for a data structure which can store and retrieve data is that its retrieval mechanism is efficient. One can store a collection of data in an un-ordered list; this is space efficient and writing to this list can be done in constant time, however querying this list requires a linear scan over each element. One can maintain an ordering over the data, and store it in a binary search tree. This incurs very little space overhead (e.g. 16 bytes per storage element) and retrieval can now be performed in logarithmic time, however inserting elements increases from constant time to logarithmic.

Hash tables achieve a constant-time read and write operation by maintaining a map from input elements to their corresponding index in a table. The map is called a hash function, and they are designed to require constant time to execute. The overall data-structure is known as a hash table (Cormen et al., 2009). The elements themselves are stored in an un-ordered list, and their locations (e.g. pointers, in modern computing systems) are stored in the hash table. When two different inputs are mapped to the same location this is known as a *collision*, a simple resolution is to *chain* the corresponding elements mapped to this location in a linked list, as shown in Figure 2.3.

Hash tables do incur a number of implementation decisions, there is an additional space overhead consumed by empty locations in the hash table, there are considerations for methods to adaptively increase the size of the table as more data is stored, and there are different approaches for dealing with *collisions* when multiple input elements are mapped to the same slot. Crucially many different implementations of hash tables do arrive at a constant-time input and retrieval time in

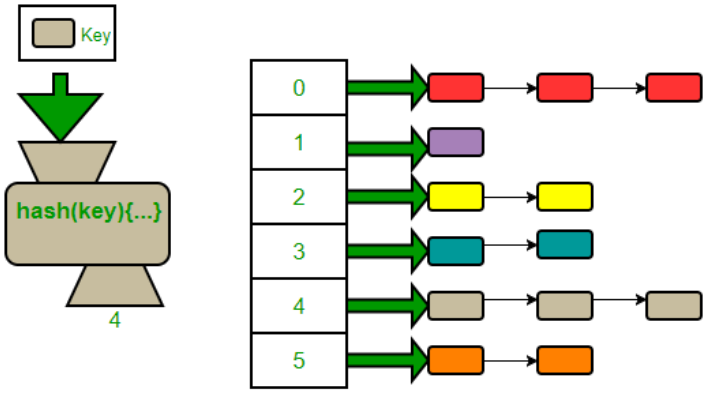


Figure 2.3: Hash table schematic with collisions chained as lists, from Smith (2019).

theory and practice, and they are a central data-structure to many databases and networking high-performance computing systems (Maurer and Lewis, 1975). Thus the problem of memory retrieval can be solved very efficiently — constant time — with a sparse-access hash function if we know exactly what we are looking for; is this the case for non-exact matching queries? Neural networks may wish to query for memories which resemble the current input but do not exactly match it. We shall discuss this problem in the following section.

2.2.2 Nearest Neighbour Search

Nearest neighbour search is the problem of finding the closest element $x_1, \ldots, x_n$ in a set of stored data to a given query element: $\arg\min \|q - x\|$. Naively storing the data in an un-sorted list allows for $\mathcal{O}(nd)$ nearest neighbour retrieval, where d is the dimensionality of the data-points. If $d = 1$ we can trivially improve upon this by indexing the data with a binary search tree, and serve queries in $\mathcal{O}(\log n)$ time, however the approach of indexing the data grows exponentially in the size of the

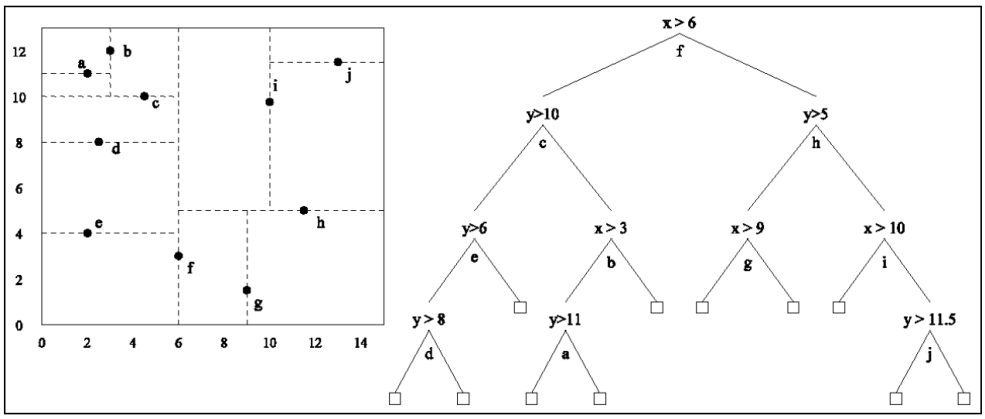


Figure 2.4: K-d tree partition of 2-D data-points. Schematic from Kraus and Dzwinel (2008).

dimensionality d (Borodin et al., 1999), due to the curse of dimensionality. Thus unlike the constant-time retrieval for when one knows exactly what one is looking for, there is a linear-time cost for when one only knows of a similar query.

Approximate nearest neighbour search rephrases the problem to allow for a factor of error ϵ. Namely instead of returning the closest element, it returns the element which is no more than $1 + \epsilon$ times further away than the nearest neighbour, with some probability p. Finding the approximate nearest neighbour can be done in $\mathcal{O}(\frac{d}{\epsilon^2} \log n)$ time (Andoni and Indyk, 2006), i.e. exponentially faster than the exact nearest neighbour. There are several approaches to building data structures which can efficiently solve the approximate nearest neighbour problem. Many partition the data-points $x_1, \ldots x_n$ into buckets of points which are close together. This includes tree-based partition algorithms such as *k-d trees* (Bentley, 1975). K-d trees split the data-points along their axis, as shown in Figure 2.4. This split value is usually chosen to be the median value along the chosen dimension, to arrive at a tree with $\log n$ depth and an equal number of data-points in each resulting

hypersphere. The accuracy of the tree is dependent on the first dimensions that the data are split on; often many such trees are built with a random shuffling of dimension ordering, i.e. randomised k-d trees (Muja and Lowe, 2014).

When the number of dimensions is large, tree-based space partitioning degrades in accuracy, a favoured approach in this setting is *locality-sensitive-hashing* which buckets the data-points mapped with a hash function which contains some distance-preserving properties (Gionis et al., 1999). One class of distance-preserving hash functions are random projections. For example it can be noted that for two points $x, y \in \mathbb{R}^d$, their distance $\|x-y\|_2$ can be approximated by their distance under random projection with a gaussian vector $\mathbb{R}^d \ni a \sim N(0, I)$. Namely the 1-dimensional distance $a \cdot x - a \cdot y \sim N(\|x-y\|_2, \|x-y\|_2^2)$ is an unbiased estimator of the d-dimensional distance. Thus one can use multiple random projections $a_1, \dots a_k$ and partition each 1-d projection space into equal buckets of width $r, h(x) = (\lfloor \frac{a_i \cdot x}{r} \rfloor,\ j = 1, \dots k)$. At query-time the query is hashed into the k buckets and the exact distance is computed between the query and the associated data-points contained in these buckets, often prioritised by the number of matching buckets. This is known as p-stable LSH hashing because it can be used for any p-norm $\|x-y\|_p$. One can specify a distribution D_p such that we can sample random vectors $a \sim D_p$ for which the p-norm follows the distribution of D_p under projection: $a \cdot (x-y) \sim \|x-y\|_p Z, Z \sim D_p$ (Datar et al., 2004). For example, the 2-stable distribution D_2 is the Gaussian distribution, and the Cauchy distribution is 1-stable.

If the data is highly isotropic then splitting projection axes into equally-sized buckets can result in some buckets which have too many associated data-points

and many empty buckets. *Data-dependent* schemes incorporate the distribution of the data into the algorithm. K-means nearest-neighbour algorithms bucket the data into clusters from a k-means clustering scheme where the number of clusters is typically chosen to be the square-root of the number of data-points $\sqrt{n}$. We see in Figure 2.5 that k-means LSH outperforms a number of baselines including random projection when benchmarked on features derived from images. The distance is computed between the query and each of the $\sqrt{n}$ cluster centroids, and then after with the $\sqrt{n}$ points contained in the closest cluster. This $\mathcal{O}(\sqrt{n})$-time query can be lowered to $\mathcal{O}(\log n)$ time by having a hierarchical clustering scheme, i.e. recursively partitioning the data into two separate clusters until a balanced tree of depth $\log_2 n$ is constructed (Nister and Stewenius, 2006). This can both reduce the computational cost of clustering all data-points simultaneously (which can consume quadratic time) and speed up queries, at a small reduction to query accuracy. See Nister and Stewenius (2006) for a more complete survey on approximate nearest neighbour search using locality-sensitive hashing.

2.2.3 Set Membership

Instead of storing and retrieving data, we may want to store it and simply know whether we have seen it or not. I.e. instead of *retrieval*, we may only be interested in *familiarity*. The problem of *exact set membership* is to state whether or not a given query q belongs to a set of n distinct observations $S = \{x_1, \ldots, x_n\}$ where x_i are drawn from a universe set U. By counting the number of distinct subsets of size n it can be shown that any such exact set membership tester requires at least $\log_2 \binom{|U|}{n}$ bits of space. To mitigate the space dependency on $|U|$, which can

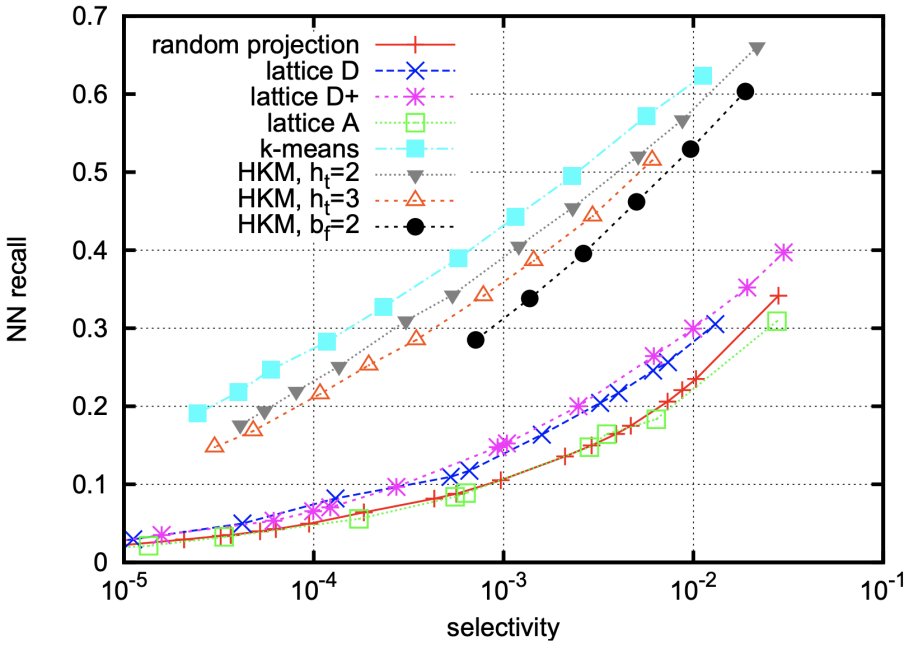


Figure 2.5: Comparison of approximate nearest-neighbour on image features (Lowe, 2004). Schematic from Paulevé et al. (2010).

be prohibitively large, one can relax the constraint on perfect correctness.

Approximate set membership allows for a false positive rate of at most ϵ. Specifically we answer $q \in A(S)$ where $A(S) \supseteq S$ and $p(q \in A(S) - S) \leq \epsilon$. It can be shown[2] the space requirement for approximate set membership of uniformly sampled observations is at least $n \log_2(\frac{1}{\epsilon})$ bits (Carter et al., 1978) which can be achieved with perfect hashing. So for a false positive rate of 1%, say, this amounts to 6.6 bits per element. In contrast to storing raw or compressed elements this can be a huge space saving, for example ImageNet images require 108 KB per image on average when compressed with JPEG, an increase of over four orders of magnitude.

[2]By counting the minimal number of $A(S)$ sets required to cover all $S \subset U$.

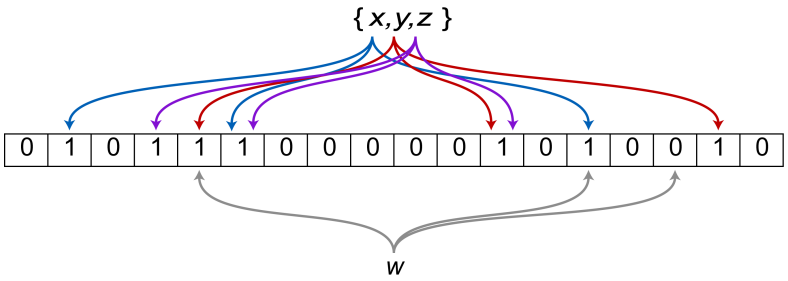


Figure 2.6: Bloom Filter using $k = 3$ hash functions storing the inputs 'x', 'y', 'z', and querying 'w' with a negative result. Schematic from Eppstein and Goodrich (2007).

2.2.4 Bloom Filter

The Bloom Filter is a data structure which solves the dynamic approximate set membership problem with near-optimal space complexity (Bloom, 1970). It assumes access to k uniform hash functions $h_i : U \to \{1, \ldots, m\},\ i = 1, \ldots, k$ such that $p(h_i(x) = j) = 1/m$ independent of prior hash values or input x. The Bloom Filter's memory $M \in [0, 1]^m$ is a binary string of length m which is initialised to zero. Writes are performed by hashing an input x to k locations in M and setting the corresponding bits to 1, $M[h_i(x)] \leftarrow 1;\ i = 1, \ldots, k$. For a given query q the Bloom Filter returns true if all corresponding hashed locations are set to 1 and returns false otherwise,

$$Query(M, q) := M[h_1(q)] \wedge M[h_2(q)] \wedge \ldots \wedge M[h_k(q)] \ .$$

We see an example of this in Figure 2.6 where a Bloom Filter stores three characters 'x, y, z', each with three hash functions. When the character 'w' is queried some locations collide with the prior characters' however at least one hash function maps 'w' to an un-written location (holding a zero) which indicates we have not seen

'w' before.

More generally we can assert that a Bloom Filter incurs zero false negatives, as any previously observed input must have enabled the corresponding bits in M, however there can be false positives due to hash collisions. To achieve a false positive rate of ϵ with minimal space one can set $k = \log_2(1/\epsilon)$ and $m = n \log_2(1/\epsilon) \log_2 e$, where e is Euler's constant. The resulting space is a factor of $\log_2 e \approx 1.44$ from the optimal static lower bound given by Carter et al. (1978).

Bloom Filters provide no guarantees of false positive rates under approximate matching, elements are only familiar if they have been exactly seen before. Furthermore they have a fixed operation which is independent of the data distribution — their performance is bounded by a worst-case analysis. However having memory components which can work under uncertainty and can specialise to the data distribution is crucial for intelligent agents which need to reason over a complex and noisy stream of sensory inputs. In the next section we discuss such learnable memory structures.

2.3 Memory-Augmented Neural Networks

In this section the topic progresses to memory in artificial neural networks. The topic is naturally complementary to the prior two sections: *memory in animals* and *data storage in computing*, because we wish to obtain the same complex reasoning and capacity from biological memory systems however we wish to ultimately devise algorithms which run efficiently on modern hardware and can be analyzed in terms of their asymptotic complexity.

The key differentiator between a memory system for neural networks versus classical data-structures, is that its read and write operations must be compatible with a learning algorithm. For modern neural networks this learning algorithm has settled upon *backpropagation* (Rumelhart et al., 1986), an instance of gradient-descent optimisation. If the reads and writes to memory are differentiable, they can be placed within a neural network which is trained via backpropagation and the network can learn what to read and write in order to solve the underlying task. However can be optimised in theory may not result in a practical memory system, becaues the optimisation may be slow or too expensive as the capacity of the memory is augmented.

The scope of this section is restricted to memory in artificial neural networks because these systems have driven such a large increase in performance for temporal reasoning tasks. However it is worth noting that the intersection of memory and machine learning is broader. The storage of exemplar data-points or embeddings, which can be thought of as memories, has long been used within machine learning systems. For example, Support Vector Machines (Cortes and Vapnik, 1995) which can learn to classify or regress input data by storing *support vectors*. These support vectors are pertinent inputs with respect to the task (e.g. they are inputs which sit closest to the classification boundary), and they can be thought of as a compact memory. Nevertheless it would require a non-trivial modification to the SVM architecture for such a model to learn a function approximator over a temporally ordered sequence of inputs. For a more general background in non-parametric machine learning models see Wasserman (2006).

We will cover a number of seminal memory architectures in the literature of arti-

ficial neural networks, discussing the mechanics of such models alongside remarks of capacity limitations that motivate this thesis.

2.3.1 Hopfield Networks

Hopfield networks are an early mathematical model of associative memory (Hopfield, 1982). They operate by storing binary-pattern inputs into the weights of a fully-conected neural network using a local update rule. For an input $\xi \in \{-1, 1\}^d$ containing d binary values (positive or negative), a Hopfield network contains d^2 real-valued connections, $W \in \mathbb{R}^{d \times d}$. The Hopfield learning algorithm specifies a write over n binary memories $\xi_1, \xi_2, \ldots, \xi_n$, represented as column vectors, by cumulating their outer product:

$$W \leftarrow \sum_{l=1}^{n} \xi_l \xi_l^T \tag{2.1}$$

This specific write update is termed the Hebbian update rule as it follows the "fire together, wire together" principle

$$W[i, j] = \sum_{l=1}^{n} \xi_l[i] \xi_l[j]$$

proposed by the psychologist Donald Hebb as a model of synaptic learning (Hebb, 1949).

Hopfield Networks are a model of associative memory, as they can store a sequence of patterns, and reconstruct a partiall occluded prior pattern $\tilde{\xi}$. Thus if our binary input patterns contain several overlapping piece of information, such as the sight and smell of a wild fruit and the label of whether it is poisonous, then we can

reconstruct an important piece of information (whether it is poisonous) from a subset of other stimuli (sight of fruit). This is also known as *content-addressable* memory, a term which will be used frequently throughout this study.

The Hopfield Network read operation involves the minimisation of an *energy function*

$$E(W, s) = -\frac{1}{2} s^T W s \tag{2.2}$$

where $s \in \mathbb{R}^d$ is the *state* of the network, initialised to the query $\tilde{\xi}$ initially, and then optimised to a stable state known as an attractor. This can be viewed as the retrieved memory. The Hopfield Network defines the optimisation of the energy via the update:

$$s_0 \leftarrow \tilde{\xi} \in \{-1, 1\}^d \qquad \textit{Initialise with query}$$

$$s_{t+1}[i] \leftarrow 1 \text{ if } W[i]^T s_t > 0; \text{ else -1} \quad \text{for } i = 1, 2, \ldots, d \qquad \textit{Update}$$

The update is applied from $s_0 \to s_T$ such that T is the first time-step satisfying $\text{Update}(s_T) = s_T$. The resulting s_T is known as an attractor state. Crucially it is a local minima of the energy function (2.2), and is the resulting value read from the Hopfield Network. Thus we can think of the Hopfield read operation as a discrete optimisation of the energy function given a start state, the query $\tilde{\xi}$.

The capacity of a Hopfield Network is linear in terms of the number of weights. Specifically, for an input dimension d the network can store at most $d/4 \log d$ patterns, if each corrupted query is at most $d/2$ Hamming distance away from the original pattern (McEliece et al., 1987). Thus each real-valued weight increases the capacity by $1/4 \log d$ binary-valued patterns. Thus the read and write operation

is linear in terms of the number of real-value connections, and thus the capacity of the network.

Hopfield Networks serve as an interesting weight-based method of memory storage. However although they use optimisation as a method of memory retrieval, their learning rule is not differentiable due to the use of discrete states. Whilst this is still possible to interface with a differentiable neural network using straight-through gradient estimates or reinforcement learning, this added complexity has arguably stymied Hopfield Networks from being adopted as a memory store for modern neural networks. We will next investigate such end-to-end learning systems in the form of differentiable recurrent neural networks, which represent memories as recycled activations in a recurrent state — versus a set of fast-changing auto-associative weights.

2.3.2 Recurrent Neural Networks

Recurrent neural networks are a class of models which map a sequence of inputs $x_{1:t}$ to a sequence of outputs $y_{1:t}$ by feeding a *recurrent state* h_{t-1} from the previous timestep to the next (Rumelhart et al., 1986). Whilst many computational neural network models contain recurrent connections, including Hopfield Networks as part of their weight update mechanism, here the focus is on differentiable recurrent neural networks that can be trained via gradient descent.

These models can be expressed with a differentiable function f with learnable parameters ϕ:

$$f_\phi(x_t, h_{t-1}) \;\rightarrow\; y_t, h_t \;.$$

The recurrent state cycles information into the network from the previous time-step and allows the network to condition on information from the past, serving as a memory. One of the simplest recurrent neural networks is the Elman network (Elman, 1990) which chooses a single-layer Perceptron (Rosenblatt, 1961) to update the recurrent state and produce an output,

$$h_t = \sigma\left(W_h h_{t-1} + W_{hx} x_t + b_h\right)$$
$$y_t = \sigma\left(W_y h_t + b_y\right) .$$

Here we use σ to denote the sigmoid activation function, and $W_h \in \mathbb{R}^{d \times d}, W_{hx} \in \mathbb{R}^{d_i \times d}, W_y \in \mathbb{R}^{d \times d_o}$ are the learnable weight matrices for a model with hidden state d and input-output dimensions d_i, d_o respectively. The $b_h \in \mathbb{R}^d, b_y \in \mathbb{R}^{d_o}$ are learnable bias terms.

An empirically effective way to train neural networks is via gradient descent. For recurrent neural networks, and more generally neural sequence models, this is most commonly done with *backpropagating through time (BPTT)*. This is an application of the iterative backpropagation algorithm, used to calculate the derivative of the loss with respect to the model parameters $\partial L / \partial \phi$. The RNN is unrolled over the input sequence, the loss with respect to each output step is calculated, and backpropagation is calculated over the unrolled graph of computation; starting from the last time-step and working backwards.

Due to the multiplication of error signals over time, choosing f to be a standard multi-layer perceptron (MLP) or even a linear transform results in vanishing or exploding gradients (Hochreiter, 1998). This can most clearly be seen if the update

to the recurrent state is a simple linear transformation, i.e. $h_t = Wh_{t-1}$. Then $\partial h_t/\partial h_{t-1} = W$, which implies the n-step gradient $\partial h_t/\partial h_{t-n} = W^n$. If the largest eigenvalue of W is greater than one, this norm explodes in magnitude, and if it is less than one, it tends to zero.

2.3.3 Gated Recurrent Neural Networks

The Long Short Term Memory (LSTM) is a recurrent neural network designed to avoid this issue (Hochreiter and Schmidhuber, 1997). It works by gating the modification of the recurrent state. Namely it uses an *input* gate to filter information that will be used to update the recurrent state, a *forget* gate to remove information from the state (introduced later by Gers et al. (1999), and an *output* gate to select which information will influence the current time-step's predictions, and the next time-step's update. The LSTM's hidden state consists of two $d-$dimensional vectors, c referred to as the 'cell' and h referred to as the hidden state which is also the output of the network. It can be visualised with a network diagram as shown in Figure 2.7 but more concretely described by the following set of update

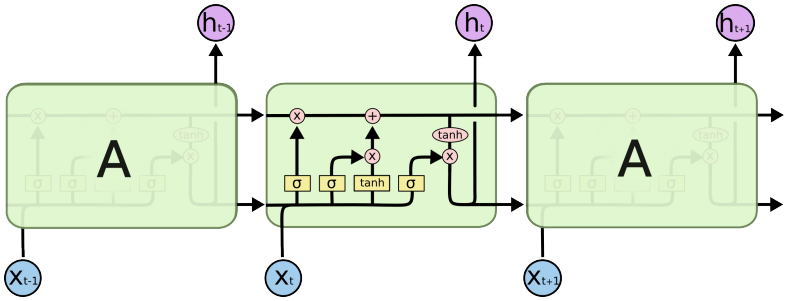


Figure 2.7: The LSTM recurrent neural network, schematic from Olah (2015).

equations:

$$
\begin{aligned}
i_t &= \sigma\left(W_{ih}h_{t-1} + W_{ix}x_t + b_i\right) \quad \text{Input gate} \\
f_t &= \sigma\left(W_{fh}h_{t-1} + W_{fx}x_t + b_f\right) \quad \text{Forget gate} \\
y_t &= \sigma\left(W_{yh}h_{t-1} + W_{yx}x_t + b_y\right) \quad \text{Output gate} \\
c'_t &= \sigma\left(W_{ch}h_{t-1} + W_{cx}x_t + b_c\right) \quad \text{Cell update} \\
c_t &= f_t \odot c_{t-1} + i_t \odot c'_t \\
h_t &= y_t \odot \tanh(c_t)
\end{aligned}
$$

Here, $\odot$ signifies element-wise multiplication. All gate activations are bounded between $[0, 1]$ due to the sigmoid activations. The hidden state is bounded between $[-1, 1]$ due to the tanh activation function, although a popular LSTM variant (peephole LSTM) removes the tanh and allows the cell to be unbounded on the real axis. The hidden cell c_t is unbounded. The crucial observation is that gate values can allow or prevent information to pass at a given time-step, which means gradient values can be reset to zero for certain indices — and so the model does not necessarily suffer from the vanishing or exploding gradients problem. LSTMs

have since been applied widely to domains such as handwriting and speech recognition (Graves and Schmidhuber, 2009; Graves et al., 2013), machine translation (Wu et al., 2016), continuous control (Heess et al., 2015), and deep reinforcement learning (Mnih et al., 2016).

One simplification of the LSTM, which has received widespread use, is the Gated Reccurent Unit (GRU). The Gated Recurrent Unit passes information over time via a single hidden state vector $h_t \in \mathbb{R}^d$, and simplifies the gating to an *update* gate and a *reset* gate. The GRU can be be formally defined with a simpler set of update equations:

$$
\begin{aligned}
z_t &= \sigma\left(W_{zh} h_{t-1} + W_{zx} x_t + b_z\right) \quad \text{Update gate} \\
r_t &= \sigma\left(W_{rh} h_{t-1} + W_{rx} x_t + b_r\right) \quad \text{Reset gate} \\
u_t &= \tanh\left(W_{uh}(h_{t-1} \odot r_t) + W_{ux} x_t + b_u\right) \quad \text{Hidden state update} \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot u_t
\end{aligned}
$$

The GRU has proven popular over the LSTM due to its comparable performance with a reduced amount of computation per step (Chung et al., 2014). Namely, the GRU requires $2\times$ less computation per step ($8d^2$ FLOPs for the LSTM vs $4d^2$ for the GRU) and the state size requires $2\times$ less space ($2d$ floating point numbers for the LSTM and d for the GRU). It has often been applied to performance-critical sequence modelling tasks, such as financial modelling (Shen et al., 2018) and healthcare monitoring systems (Zhao et al., 2019). Intuitively the GRU is able to achieve comparable performance via reduced computation and space by taking advantage of the redundancy in the LSTM's recurrent state (as c_t and h_t contain

very similar information) and some of the memory access computation (i.e. using a reset gate versus an input and output gate). We apply a similar philosophy in this study, by investigating architectures that reduce redundant memory access computation (e.g. in Chapter 3 where we obtain an exponential reduction in compute as a function of memory capacity) and those which learnably remove redundant information from their memory (e.g. in Chapter 6 via a compressive memory bottleneck).

One general short-coming of recurrent neural networks, including the Elman network, LSTM and GRU, is that the number of trainable parameters is tied to the memory capacity. For any of these RNNS, for example, if the state is a vector of dimension d then the networks contain $\mathcal{O}(d^2)$ parameters. For LSTMs it is in fact $8d^2 + 4d$. This naturally results in a quadratic compute requirement. Therefore RNNs are typically constrained to state sizes in the order of hundreds or singular thousands in common applications. This motivates neural networks that contain memory systems which are independent of the parameter count in the network, and are more compute-efficient.

2.3.4 Attention

Attention allows neural networks to reason over memories that are much larger capacity than the states of traditional recurrent neural networks. If we define an external memory $\mathbf{M} \in \mathbb{R}^{n \times d}$ is a collection of n real-valued vectors, or *words*, of fixed size d. A soft *read* operation is defined to be a weighted average over memory words,

$$r = \sum_{i=1}^{n} a(i)\mathbf{M}(i)\,, \tag{2.3}$$

where $a \in \mathbb{R}^n$ is a vector of weights with non-negative entries that sum to one. *Attending* to memory is formalised as the problem of computing a. A *content addressable memory* is an external memory with an addressing scheme which selects w based upon the similarity of memory words to a given query q (Graves et al., 2014; Weston et al., 2014; Bahdanau et al., 2014; Sukhbaatar et al., 2015). Specifically, for the ith read weight $a(i)$ we define,

$$a(i) = \frac{f\left(d(q, \mathbf{M}(i))\right)}{\sum_{j=1}^{n} f\left(d(q, \mathbf{M}(j))\right)}, \tag{2.4}$$

where d is a similarity measure, typically Euclidean distance or cosine similarity, and f is a differentiable monotonic transformation, typically the exponential function. We can think of this as an instance of kernel smoothing where the network learns to query relevant points q.

The addressing scheme is content-based because the content of the memory dictates which rows we read from. Because the read operation (2.3) and content-based addressing scheme (2.4) are smooth, their derivative can be calculated and so they can be placed in a neural network and optimised via backpropagation.

2.3.5 Memory Networks

Memory Networks make use of a content addressable memory that is accessed via a series of read operations (Weston et al., 2014; Sukhbaatar et al., 2015). Given a context that we wish to reason over, such as a collection of sentences $x_1, x_2, \ldots, x_n$, these are individually embedded into n d-dimensional hidden states $h_1, h_2, \ldots, h_n$. These are used to construct a key-value memory, where *keys* are equal to $k_i = h_i K$

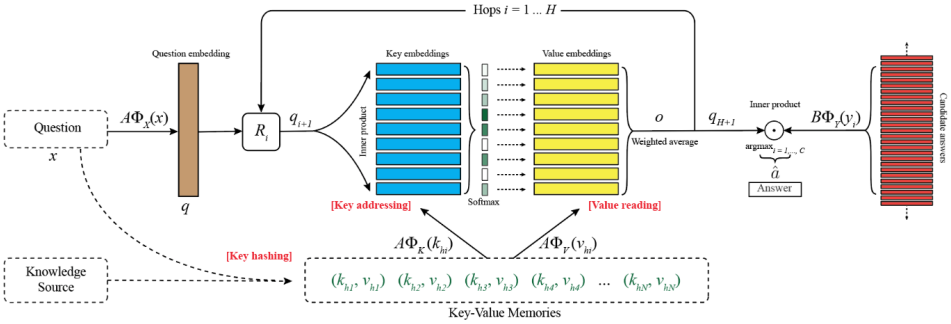


Figure 2.8: Memory Networks for question answering, schematic from Weston (2016).

and *values* equal to $v_i = v_i V$ where $K, V \in \mathbb{R}^{d \times d}$ are learnable weights. The memory contains pairs of keys and values, $M = [(k_1, v_1), \ldots, (k_n, v_n)] \in \mathbb{R}^{n \times 2d}$. A read was originally defined as the $v_l; l = \arg\max_i q \cdot k_i$, i.e. the value corresponding to the key which most closely resembled the query. However a differentiable variant proposed in Sukhbaatar et al. (2015) defined the read as a weighted sum of values where the weighting is defined by attention between the queries and keys, $r = \sum_{i=1}^{n} a_i v$; $a_i = e^{q \cdot k_i} / \sum_{j=1}^{n} e^{q \cdot k_j}$. The number of network parameters in Memory Networks is $\mathcal{O}(d^2)$, thus crucially it is not a function of the number of rows in the memory, n. This allows us to train with a larger memory capacity than RNNs.

This process is shown for the task of question answering in Figure 2.8, the network queries this memory either once or several times (i.e. H times in the above figure). Memory networks have been successfully applied to a number of question answering tasks (Weston et al., 2015; Hill et al., 2015). They are best suited to tasks with fixed contexts that need multiple steps of reasoning over, such as open-book question answering, instead of tasks where the entire contents of memory may need to be

updated or re-written, such as machine translation.

2.3.6 Neural Turing Machines

The Neural Turing Machine (NTM) is a recurrent neural network equipped with a content-addressable memory, similar to Memory Networks, but with the additional capability to write to memory over time and read memories based on absolute location in memory alongside content. The NTM consists of a controller network which can specify what to read and write to memory, alongside the address of where in memory these reads and writes should occur. The address in this case is specified both from **content**, using the same content-based attention mechanism as Memory Networks, but also relative **location** — allowing the network to learn shift operators along consecutive slots of memory.

Controller

The NTM incorporates an inner *controller* neural network to decide how to read and write from memory. Broadly the write operation specifies how the external memory $M_{t-1} \in \mathbb{R}^{n \times d}$ is updated to M_t and the read operation results in a single vector $r \in \mathbb{R}^d$ that is read from M_t. This is often chosen to be an LSTM but it can also be a feed-forward neural network, it takes the current time-step's input x_t along with the hidden state of the controller (if any) h_{t-1}, and the previously read word r_{t-1} and outputs a tuple of tensors which dictate the interaction with external memory:

- $\rho^R = (q^R, \gamma^R, g^R, s^R)$ read address inputs
- $\rho^W = (q^W, \gamma^W, g^W, s^W)$h write address inputs

- $w \in \mathbb{R}^d$, write word.
- $e \in [0,1]^d$, erase word.

where the address inputs for read and write are,

- $q \in \mathbb{R}^d$ content-based query.
- $\gamma \in \mathbb{R}^+$, sharpening scalar.
- $g \in [0,1]$ interpolation gate.
- $s \in \mathbb{R}^3$, shift weighting.

The query, sharpening scalar, and interpolation gates are used to calculate an address $a^R \in \mathbb{R}^n, a^W \in \mathbb{R}^n$ for the read and write operations respectively. Each address specifies a probability distribution over the rows of M, namely $\sum_j a(j) = 1$ and $a(j) \geq 0, j = 1, \ldots, n$. It is a differentiable vector specifying where in memory to read and write from. The address vector is a mixture of content-based addressing (using attention) and location-based addressing (using shift operators), we shall describe these in more detail.

Addressing

The NTM addressing mechanism is identical for read and write operations and so we describe the general computation. Because different parameters are used to specify the inputs to the read and write address calculation, the NTM can learn to simultaneously read and write to different locations in memory.

The content-based addressing is defined to be content-based attention from (2.4) using cosine-similarity for D and the gaussian kernel,

$$a_t^c(i) \propto \exp(q_t \cdot M_t(i) / \|q_t\| \, \|M_t(i)\|) \, .$$

The address is then an interpolation between the current timestep's content-based address and the previous timestep's address,

$$a_t^g = a_t^c \cdot g_t + a_{t-1} \cdot (1 - g_t).$$

Further location-based addressing is controlled by applying a shift to the address

$$\tilde{a}_t(i) = \sum_{j=1}^{n} a_t^g(j) \cdot s_t(i - j)$$

where $s_t \in \mathbb{R}^n$ is a shift vector with $\sum_i s_t(i) = 1, s_t(j) \geq 0$ that determines how far to shift the attention weight, e.g. for a four-row memory matrix $s = [0, 1, 0, 0]$ would shift the attention exactly up one row (with attention over the first row cycling round to the last). In practice s can be of lower-dimensions than n to restrict the maximum possible shift i.e. $s \in \mathbb{R}^3$ to allow only shifts of -1, 0, and 1.

Finally the attention is *sharpened* to allow the network to concentrate the address (i.e. push its value close to one) around a particular row,

$$a_t(i) = \frac{a_t(i)^{\gamma_t}}{\sum_j a_t(j)^{\gamma_t}}.$$

Write

A *write* to memory, consists of a copy of the memory from the previous time step M_{t-1} decayed by the erase matrix R_t indicating obsolete or inaccurate content, and an addition of new or updated information A_t

$$M_t \leftarrow (1 - R_t) \odot M_{t-1} + A_t \ . \tag{2.5}$$

The erase matrix $R_t = a_t^W e_t^T$ is constructed as the outer product between the write address $a_t^W \in \mathbb{R}^n$ and erase vector $e_t \in [0, 1]^d$. The add matrix $A_t = a_t^W w_t^T$ is the outer product between the write address and a new *write word* $w_t \in \mathbb{R}^d$, which the controller outputs.

The write takes inspiration from the gated update of cells in LSTMs, where the multiplication with a vector of values within $[0, 1]$ can be used to block or erase signal from memory, and the addition operator is used to introduce new signal from the current step's computation.

Read

Following a write, the read operation is simply a weighted sum of the rows using the location and content-based addressing scheme,

$$r_t = \sum_{i=1}^{n} M_t(i) a^R(i) \ .$$

Parameters

The NTM contains all of its learnable parameters in the controller which is independent in size to the external memory, similar to Memory Networks. The number of parameters is $\mathcal{O}(d^2)$, the capacity of memory is $\mathcal{O}(nd)$, and the cost of addressing, reading, and writing to memory is $\mathcal{O}(nd)$. A selection of models are compared in Table 2.1. The Neural Turing Machine and Memory Networks have comparable

Table 2.1: A comparison of parameter count and access complexity for different memory-augmented neural networks.

Model	Capacity	No. Params	Read Cost	Write Cost
LSTM	d	$\mathcal{O}(8d^2)$	$\mathcal{O}(d^2)$	$\mathcal{O}(d^2)$
Memory Networks	$n \times d$	$\mathcal{O}(d^2)$	$\mathcal{O}(nd + d^2)$	$\mathcal{O}(d^2)$
NTM	$n \times d$	$\mathcal{O}(5d^2)$	$\mathcal{O}(nd + d^2)$	$\mathcal{O}(nd + d^2)$

capacity and read cost, however they differ from Memory Networks in their use of linear-time (with respect to the number of memory slots n) write operations. This is crucial to allow the model to update existing information in external memory and organise its contents into a semantic structure, e.g. for copying sequences of data.

Multi-head Memory Access

The NTM can optionally be configured to have multiple read and write *heads*. That is, instead of specifying a single set of read and write inputs (ρ^R, ρ^R, w), the controller outputs h of them, and each produce a separate write: $M_t^{(i)}, r_t^{(i)}; i = 1, \ldots, h$ which are averaged: $M_t = \frac{1}{h} \sum_i M_t^{(i)}, r_t = \frac{1}{h} \sum_i r_t^{(i)}$.

Models that use multiple heads have been observed to be more stable and faster to train, even when the number of trainable parameters is kept constant. This is likely

due to the geometric benefit of having multiple different attempts to retrieve the correct information, once one read head begins to function correctly the rest can follow suit. Furthermore, multiple heads may learn different but complimentary functions. Multi-head memory access has since become a standard feature of memory-augmented neural networks.

2.3.7 Neural Stacks

The NTM and DNC both contain a 'content-based' write scheme where inputs are written into memory with a location that is a function of their contents. This uses attention and places a non-zero write weight over all slots in memory, which implies the write operation is 'dense' as every row in memory is written to. Both networks also use a location-based write scheme, where the write address is a function of past addresses (i.e. the NTM can shift the write head to the left and write).

Another class of memory-augmented neural networks are those which restrict writes to purely location-based schemes and emulate the data access of classical data structures. Grefenstette et al. (2015) proposes a *neural stack*, *neural queue*, and *neural dequeue*, which are three variants of the same general idea: a recurrent neural network controller can push or pop data into a slot-based memory either from the top (a stack), the bottom (a queue) or from both sides (a dequeue).

We will describe the stack architecture, since the other two are variations of the same core idea. The model contains a memory $M \in \mathbb{R}^{t \times d}$ which dynamically grows by one slot for each timestep. After t timesteps it contains t rows, and a write word vector $w \in \mathbb{R}^d$ is written at timestep t to the final row — which has index t. Using the terminology of this section we class this write as being *sparse*

as the write word w is written to $\mathcal{O}(1)$ slots in memory — in this case exactly 1. The neural stack maintains differentiability by maintaining a strength vector $s \in \mathbb{R}^t$ which controls the strength of reading each past memory; it is modified via continuous *push*, $d_t \in (0, 1)$, and *pop*, $u_t \in (0, 1)$, scalar values.

The strength update is defined as:

$$s_t[t] = d_t \quad \textit{(strength is initialised with the push weight)}$$
$$s_t[i] = \max(0, s_{t-1}[i] - \max(0, u_t - \sum_{j=i+1}^{t} s_t[j])), \quad i < t$$

thus the strength is initialised as the push weight and the pop weight is used to arithmetically decrease the strengths from the highest index to the lowest, with each strength being thresholded to a minimum value of zero. The read uses these strengths to traverse down the memory matrix M. By iterating from the highest indices down to the lowest the model maintains a cumulative read weight $c_t[i] = max(0, 1 - \sum_{j=i+1}^{t} s_t[j])$ and continues reading rows until their read strength is less than this maintained value, $s'[i] = \min(s_t[i], c_t[i])$. We interpret the read strenghts s'_t as selecting the strengths s_t from the top index down until at most a cumulative strength of 1 has been selected. The resulting read is

$$r_t = \sum_{i=1}^{t} s'_t[i] V_t[i]$$

Alike to other memory-augmented neural networks a controller neural network outputs the write word and push/pop weights which control the stack's read and write operations. Since the reads and writes are differentiable the full model can

be optimised with gradient methods. The read operation is not necessarily sparse as it is possible to have positive strengths across over all past memories however this could easily be constrained by ensuring the strenghts vector contains some maximum cumulative weight or a maximum number of non-zero entries.

Grefenstette et al. (2015) found the inductive bias of the stack architecture allowed for very data-efficient learning of algorithmic tasks such as sequence copying and reversal, alongside some synthetic grammar translation tasks such as mapping Subject-Verb-Object phrases to an equivalent formulation using a Subject-Object-Verb grammar. More recently Cai et al. (2016) showed that a stack-augmented neural network could learn more generalisable recursive programs given example traces.

The neural stack explicitly does not employ a content-based memory, which can support auto-association. It is interesting to see how powerful a location-only read and write memory scheme can be, in this thesis we consider sparsifying models with both content-based and location-based memory however we place more emphasis on content-based addressing since this is a computational bottleneck for many memory-augmented neural networks which support auto-association.

2.3.8 Differentiable Neural Computer

The Differentiable Neural Computer (DNC) is a variant of the Neural Turing Machine. It contains a neural network 'controller' with a set of differntiable write and read operations to an external memory. In contrast to the Neural Turing Machine it contains a garbage collection mechanism that supports the forgetting of infrequently accessed information, and a location-based addressing scheme which

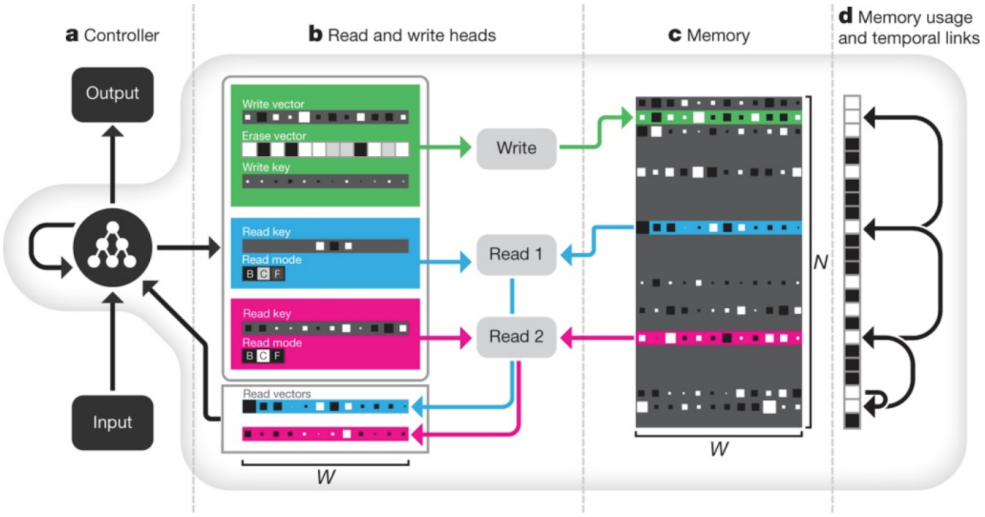


Figure 2.9: Read and writing to external memory in the Differentiable Neural Computer (Graves et al., 2016)

supports transitioning between subsequently accessed slots in memory via a temporal 'linkage' matrix, as shown in Figure 2.9. These links can be thought of as pointers in memory, they allow the network to walk through subsequently-accessed regions of memory. This combination of content-based and location-based memory lookups is quite powerful as the network can still perform auto-association but can organise memory alike to the neural stack. The DNC was found to be state-of-the-art at several synthetic algorithmic tasks involving reasoning over graph data, solving combinatorial planning tasks such as SHRDLU using reinforcement learning, and synthetic question-answering on the bAbI dataset.

2.3.9 Transformer

The Transformer is a feed-forward memory-augmented neural network which uses attention to access information over time (Vaswani et al., 2017). It avoids the use of recurrence, unlike the DNC and NTM which feed back the external memory,

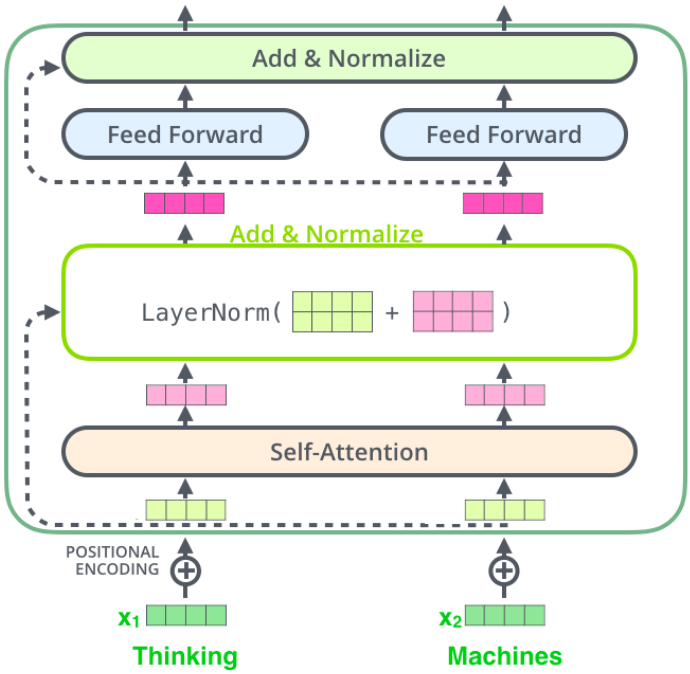


Figure 2.10: A Transformer layer, schematic from Alammar (2018)

and unlike Memory Networks which typically use a recurrent neural network to construct the memory embeddings. This allows the model to process multiple timesteps in parallel, which make it well suited to hardware specialised to facilitate batched matrix multiplications with high throughput, such as GPUs and Google's Tensor Processing Unit (TPU) (Jouppi et al., 2017).

The Transformer contains interleaved layers of attention and MLPs as shown in Figure 2.10, the original architecture was designed for machine translation and contains an encoder variant that allows for bi-directional attention forward and backwards in time, alongside a decoder architecture which is causally masked such that queries can only access the past. We shall describe the decoder variant, as this model corresponds to a sequence model with a multi-layer memory.

The model's layers consist of two blocks, a multi-head attention block and an MLP. To facilitate stable training layer-normalisation (Ba et al., 2016b) and residual connections (He et al., 2016) are used. For a sequence of embedded inputs $\mathbf{R}^{n \times d} \ni H^{(0)} = h_1^{(0)}, \ldots, h_n^{(0)}$, layer i is defined as:

$$
\begin{aligned}
H_{attn}^{(i)} &= \text{multi-head attention}(H^{(i-1)}) \\
H'^{(i)} &= \text{layer norm}(H_{attn}^{(i)} + H^{(i-1)}) \\
H_{mlp}^{(i)} &= MLP(H'^{(i)}) \\
H^{(i)} &= \text{layer norm}(H_{mlp}^{(i)} + H'^{(i)})
\end{aligned}
$$

Multi-head attention is defined to be content-based attention using separate learnable projection matrices $Q, K, V \in \mathbb{R}^{d \times d}$ to map the input embedding $H^{(i)}$ to a set of *queries* which attend to *keys* using inner product similarity, and then select the corresponding *values*. The queries, keys, and values are split into h equally-sized tensors of dimension $n \times \frac{d}{h}$ and attention is performed separately for each *head* (thus the *multi*-head). The result of each head are mixed with a learnable projection O. To ensure elements only attend to the past, the attention matrix can be

masked with a lower-triangular matrix.

$\mathbf{q} = H^{(i-1)}Q,\ \mathbf{k} = H^{(i-1)}K,\ \mathbf{v} = H^{(i-1)}V \in \mathbb{R}^{n \times d}$ queries, keys, and values

For head $j = 1, \ldots, h$

$\mathbf{q_j}, \mathbf{k_j}, \mathbf{v_j} \in \mathbb{R}^{n \times \frac{d}{h}}$ j-th slice of queries, keys, and values

$\mathbf{a_j}' = \sigma(q_j k_j^T) \in \mathbb{R}^{n \times n}$ bi-directional attention weights

$\mathbf{a_j} = \mathbf{a_j}' \cdot \text{LowerDiagonal}(n, n)$ causal attention weights

$\mathbf{r_j} = \mathbf{a_j v_j} \in \mathbb{R}^{n \times \frac{d}{h}}$ read values

$H_{attn}^{(i)} = [\mathbf{r_1}, \mathbf{r_2}, \ldots, \mathbf{r_h}]\ O \in \mathbb{R}^{n \times d}$ concatenate and mix across heads

The multi-head attention scheme in the Transformer takes inspiration from the multiple attention heads from the NTM and from the key-value memory in MemNets. By having multiple heads, the network can query multiple separate pieces of information at a given time-step, it has been later observed that different attention head can specialise to different tasks. For example, for the task of modelling natural language some attention heads learn to take weighted averages over words in a given history (essentially performing an n-gram-like specialisation) whereas others will always fixate on the previous token (Sukhbaatar et al., 2019).

Positional Encodings

An RNN can learn about the ordering of time from its recurrent weight matrix, however a Transformer applies an identical learnable function at each timestep. To understand time and the ordering of the input sequence, a distributed timestamp, in the form of sinusoidal waves of varying frequency, are added to the inputs.

For the inputs $x_t \in \mathbb{R}^d$ we calculate $h_t^{(0)} = x_t + S[t]$ where

$$S[t] = [\sin\left((t/T_{max})^{2i/d}\right) \text{ for i} = 1, \ldots, \text{d}] \tag{2.6}$$

i.e. the dimensions represent sinusoidal curves with a range of wavelengths from 2π to $2T_{max}\pi$. The authors originally proposed using a combination of sine and cosine waves and setting $T_{max} = 10,000$, i.e. such that timesteps from 0 to $10,000$ could be uniquely represented.

This scheme has shown to be effective, however simpler positional encoding schemes can also work. If one is always evaluating a model on sequence lengths that are within the range observed during training, one can instead construct a learnable parameter $S' \in \mathbb{R}^{T_{max} \times d}$ and compute $x_t + S'[t]$, i.e. learn each unit of time as independent tokens (Al-Rfou et al., 2019). The proposed benefit of using sinusoidal encodings is it injects the prior of a continuity of time and better facilitates generalisation to time ranges unseen during training.

Further iterations on how transformers should understand time have become popular. The notion of *relative* positional encoding avoids the placement of absolute time-stamps on inputs, instead preferring attending from timestep t to a timestep r-timesteps ago. One simple way of doing this is by adding a term to the attention computation:

$$a_{ij'} = (qk^T)_{ij} + q_i R[i-j]^T$$

where $R \in \mathbb{R}^{n \times d}$ is a learnable matrix and $R[i-j] \in \mathbb{R}^{1 \times d}$ is the row which indicates attending '$i - j$' steps back in time (Shaw et al., 2018).

A slight variation on relative positional encodings was proposed by Dai et al.

(2019), where we construct R not as a full learnable matrix of size $n \times d$ but as a combination of sinusoidal encodings and a learnable embedding $R_{sin} = SR'$, where S is defined in (2.6) and $R' \in \mathbb{R}^{d \times d}$. The authors note this outperforms the previous brute-force relative positional encoding scheme when the model is evaluated outside of temporal ranges observed during training — alike to that of the absolute positional encoding.

One benefit to a relative view of time is that one does not have to maintain a consistent representation of a timestamp in memory. That is, in the absolute positional encoding case, old activations essentially store their underlying timestep. If there is a drift in representations over time, this may become meaningless. With relative positions, one makes use of the fact that activations are temporally ordered in the Transformer and thus the network can discern which activations are older or younger at the time of query. In Chapter 6 we introduce a transformer variant with a compressive memory which compacts information over time. Crucially this model preserves the temporal ordering of memories, thus we use relative position encodings.

2.3.10 Non-parametric Classification

Instead of having neural networks read and manipulate memory, there has been a prevalence of using memory as a non-parametric classifier which is paired with a neural network. One combines the two models by mixing their output probability distributions from the parametric model and memory respectively. The benefit of this approach is that the neural network does not have to learn how to read memory and thus can be trained as a standalone system. The memory is simply

appended to improve the model's ability to recognise and model unfamiliar (or previously unseen) words.

One particular architecture which uses memory as a mixture model in this manner is the *Neural Cache* (Grave et al., 2016b). This uses a memory for the modelling of natural language, specifically the prediction of the next word — otherwise referred to as *language modelling*. The cache is a store of the last n hidden activations along with their corresponding target outputs (next words) from a trained parametric language model, such as the Long Short Term Memory (LSTM). The conditional probability of a word w occurring is proportional to the sum over kernalised inner product similarities between the current hidden state h_t and past hidden states when word w occurred.

$$p_{np}(w \mid h_t) \propto \sum_{i=t-n}^{t-1} e^{h_t^T h_i} \mathbb{I}\{y_i = w\} \tag{2.7}$$

Where $\mathbb{I}\{p\} = 1$ if p is true, 0 otherwise. The overall model output

$$p = \lambda p_{np} + (1 - \lambda) p_p$$

is an interpolation of the cache's output probability p_{np} and the neural network's p_p using a fixed hyper-parameter λ, which is swept over during validation. Although the cache is of fixed size n, it can be defined to be very large with sparse attention and efficient data-structures (Rae et al., 2016; Kaiser et al., 2017; Grave et al., 2017).

2.4 Memory Tasks

The creation of tasks to benchmark and analyse artificial memory systems is of equal importance to the creation of architectures themselves, as better tasks often drive the development of better architectures. We highlight some tasks that have been associated with seminal architectures in the development of artificial neural networks and make the case for modelling natural language as a non-synthetic problem which contains challenges for long-range memory systems.

2.4.1 Selective Processing of a Stream of Numbers

As part of the development of the LSTM (Hochreiter and Schmidhuber, 1997), six synthetic memory tasks were proposed to display the LSTM's ability to retain information over a long period of time. Crucially each of these tasks had the format of receiving a stream of bits or floating point values, a channel which would indicate which memories would be selected, and then and then the target output which would consist of a simple function over the selected bits (e.g. an OR operation). This included very simple tasks, such as copying the first timestep's bit pattern, after a sequence of distractors, to more challenging tasks such as adding or multiplying two selected floating-point numbers within a sequence of distractor floating-point numbers.

LSTMs performed very well at these tasks because they could learn to ignore distractor inputs, and could retain information over sequences of up to 1,000 timesteps long. Crucially, none of these tasks required the storage of many pieces of information. In all tasks the model would only need to retain up to two numbers, along with their temporal ordering in some instances.

2.4.2 Algorithmic Tasks

A set of algorithmic tasks were proposed by Graves et al. (2014) as part of the development of the NTM. These had a similar flavour to the LSTM tasks however they required the storage of a large quantity of information, and richer processing of multiple timestep's worth of data. These included copying a sequence of bits a variable number of times (shown in Figure 2.11) and sorting a sequence of values. This also included associative recall: retrieving an element from a sequence that follows a given query.

Figure 2.11: The repeated copy task from Graves et al. (2014), given an input sequence of bits and a number of repeats specification n, repeat the sequence n times.

These tasks required the content-based selection of past inputs, the ability to sequentially iterate through past elements in temporal order, and the maintenance of temporary local values such as a counter to track the number of copies performed in a repeated copy operation. Whilst the LSTM was able to solve these tasks at small scale, it simply could not scale to sequences containing hundreds of timesteps. The inclusion of a larger-capacity external memory was thus shown to be essential. We investigate these tasks at huge capacity in Chapter 3, pushing the number of items to store and the length of sequences, to the order of hundreds of thousands.

2.4.3 Language Modelling

Finding machine learning tasks which both drive the development of better memory architectures, push us further towards artificial general intelligence, and have real-world downstream applications is challenging. Statistical language modelling is one such task that could be valuable for all such purposes as argued by Rae and Lillicrap (2020). We incorporate it several times throughout the thesis as a core task. Language models work by sequentially predicting the next word in a stream of text. One can express the probability of a sequence of text as the product of conditional word probabilities,

$$p(w_1, w_2, \ldots, w_t) = \prod_{i=1}^{t} p(w_i \mid w_1, w_2, \ldots, w_{i-1})$$

which are estimated separately.

Traditional n-gram models take frequency-based estimates of these conditional probabilities with truncated contexts $p_n = p(w_i \mid w_{i-n}, \ldots, w_{i-1})$ and smooth between them to estimate the full conditional probability, $p(w_i \mid w_1, \ldots, w_{i-1}) = \sum_{j=1}^{n} \lambda_j p_j$. A popular approach is Kneser-Ney smoothing (Kneser and Ney, 1995). More recently, neural language models such as LSTMs, convolutional neural networks, and Transformers directly model the conditional probabilities by characterising the past with their internal memory structures. This has lead to a continual advancement in state-of-the-art performance across established benchmarks over the years (Mikolov et al., 2010; Collobert and Weston, 2008; Sundermeyer et al., 2012; Kalchbrenner et al., 2014; Jozefowicz et al., 2016; Dauphin et al., 2016; Melis et al., 2017; Dai et al., 2019).

Language models can be used to model existing texts and also to generate novel texts. As they get better at modelling the past, their predictions become more accurate, and the texts they generate become more realistic. Thus better memory systems can become drivers for better language models.

In Claude Shannon's seminal article 'A Mathematical Theory of Communication' published in 1948, which founded the field of information theory, he discussed primitive language models and illustrated how adding more context improves the quality and realism of generated text (Shannon, 1948). He does this by introducing the most simple model of English text, which has no contextual modelling at all: a character-level model which treats each character independently. By sampling characters with their relative frequencies (8% of the time for *a*, 1.5% for *b* etc.) we arrive with a nonsensical string :

```
xfoml rxkhrjffjuj zlpwcfwkcyj ffjeyvkcqsghyd qpaamkbzaacibzlhjqd.
```

However, he remarks at the improvement in sample quality if one instead models the probability of words independently. Now the modelled context is approximately 7X larger (the average number of characters in a word):

```
representing and speedily is an good apt or come can different
natural here he the a in came the to of to expert gray come
to furnishes the line message had be these.
```

By modelling the probability of word pairs, a further 2X in context length, even more realistic text emerges:

```
the head and in frontal attack on an english writer that the
character of this point is therefore another method for the
```

```
letters that the time of who ever told the problem for an
```

In other words, an increase in the length of context leads to an improvement in the quality of text generated. Shannon remarks on the quality of his produced samples and conjectures that natural text samples may emerge from a sufficiently complex statistical model, "The particular sequence of ten words '`attack on an English writer that the character of this`' is not at all unreasonable. It appears then that a sufficiently complex stochastic process will give a satisfactory representation of a discrete source."

One criticism of language modelling as a task for long-range reasoning is that models can capture a large portion of their predictions from the local context. Neural language models have traditionally ignored the wider context, focusing mostly on the short term. For example, it was observed an LSTM language model augmented with attention would rarely look beyond the preceding five words in order to make a prediction (Daniluk et al., 2017).

However in the past year large Transformer models have been shown to make use of hundreds of words of context to generate ever-more realistic text with a longer range of coherence. A demo from OpenAI's GPT-2 (Radford et al., 2019), a 1.5B parameter Transformer, indicate that the model is able to generate realistic text and retain key entities (e.g. Dr Jorge Perez and unicorns) across multiple paragraphs:

> ""The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white **unicorns** were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge **Perez**, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. **Perez** noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

Perez and the others then ventured further into the valley. 'By the time we reached the top of one peak, the water looked blue, with some crystals on top,' said **Perez**.

Perez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them, they were so close they could touch their horns.

While examining these bizarre creatures the scientists discovered that the creatures also spoke some fairly regular English. **Perez** stated, 'We can see, for example, that they have a common language, something like a dialect or dialectic.'

Dr. **Perez** believes that the **unicorns** may have originated in Argentina, where the animals were believed to be descendants of a lost race of people who lived there before the arrival of humans in those parts of South America.

While their origins are still unclear, some believe that perhaps the creatures were created when a human and a unicorn met each other

in a time before human civilization. According to **Perez**, 'In South America, such incidents seem to be quite common.'

However, **Perez** also pointed out that it is likely that the only way of knowing for sure if **unicorns** are indeed the descendants of a lost alien race is through DNA. 'But they seem to be able to communicate in English quite well, which I believe is a sign of evolution, or at least a change in social organization,' said the scientist."'

It is becoming increasingly clear that a powerful model can thus use long contexts to improve their modelling of natural language, and this can even be evident when evaluating long-form samples. In this thesis we find we are able to make significant gains in language model performance by effectively modelling over 5,000 words of context (see Chapter 6).

Language Models Applications

Modern language model samples would likely astound Shannon, 70 years on from his early language model experiments. However the real benefit of powerful neural language models and their relevance to the goal of AGI is their ability to transfer knowledge to a suite of tasks. In the process of learning how to model text, neural language models appear to build up a knowledge-base of associations, and a plethora of skills.

Language modelling can be directly applied to predictive text applications, however language models are also implicitly part of many other systems, such as speech recognition models and even machine translation systems. More recently, it has been observed that pre-training language models can result in strong representa-

tions of natural language, and the models can be fine-tuned to obtain state-of-the-art results in question answering, text classification benchmarks, search ranking, and dialogue.

Researchers at OpenAI showed that GPT-2 can be applied to natural-language processing tasks such as question answering, paraphrasing, or sentiment analysis with surprisingly good performance, especially for a model that has never been explicitly trained to perform such tasks (Radford et al., 2019). When large Transformer language models are fine-tuned on particular tasks such as question answering, the resulting performance is significantly better than models that were designed and trained solely for question answering (Raffel et al., 2020). Google's prominent natural language model, BERT (Devlin et al., 2019), is a transformer which achieves state-of-the-art performance on a wide array of NLP benchmarks, and is now incorporated to rank results in Google Search. And more recently, it was shown that GPT-2 can learn to play rudimentary chess by training it on strings of game moves (Alexander, 2020).

Benchmarking language models

The canonical benchmark for language models which has seen progress over from n-gram language models to the latest transformer architectures is Penn Treebank, a selection of Wall Street Journal articles initially used by Mikolov et al. (2010) to demonstrate the effectiveness of RNN-based LMs. However the average article length is 350 words and the dataset size is very small, which renders overfitting as the principal challenge to overcome. A more recent popular long-range language model benchmark is WikiText-103, which is comprised of English-language

Wikipedia articles (Merity et al., 2016). Articles are around 3,600 words on average, which, at the time of creation, was far beyond the memory window of state-of-the-art models.

However researchers at Google recently showed that a Transformer variant called the TransformerXL which maintains a memory of past network activations and recently obtained state-of-the-art results on WikiText-103 can make use of contexts spanning over one thousand words (Dai et al., 2019). This raises the question: will models soon saturate these benchmarks? As such, we've compiled and released a new, longer-range language model benchmark based on books called PG-19 which we discuss in Section 6.4.

2.4.4 Question Answering

One of the most closely related tasks to memory is question answering (QA); probing for information from a knowledge base or from a model's past observations. Auto-associative memory retrieval can be phrased as a question answering problem, "find a past observation similar to this query" as can many temporal reasoning tasks. We break down question answering into the synthetic, where a templated observation is seen and a query is generated which tests a particular form of temporal reasoning, and natural-data question answering where the knowledge base and questions are sourced from real data. Question answering benchmarks cover both visual question answering over video (Tapaswi et al., 2016), and images (Antol et al., 2015), however in this section we restrict the scope to the rich domain of text-only question answering.

Synthetic QA with bAbI

A suite of synthetic text-reasoning tasks named 'bAbI' was developed alongside the creation of Memory Networks (Weston et al., 2015). These are framed as question-answering tasks where there is a specific underlying template to the context and answer, to test a different feature of memory. It includes, in full, 20 separate tasks and models are encouraged to train jointly on all problems. Some include simple yes-no questions, e.g. Task 6: *"John moved to the playground. Daniel went to the bathroom. John went back to the hallway. Is John in the playground? A:no. Is Daniel in the bathroom? A: yes"*, some are specifically designed to require two-hope reasoning, e.g. Task 2: *"John is in the playground. John picked up the football. Bob went to the kitchen. Where is the football? A: playground"*.

These tasks emulate a natural-language reasoning over text, and specify different functions of memory: the ability for content-based retrieval, the updating of facts about an entity (e.g. the location of a person), and the redundancy of language. We introduce an architecture, the Sparse Differentiable Neural Computer, in Chapter 3 that obtained state-of-the-art results on this task for several years. The benefit of these tasks is they clearly isolate different properties of memory we would like from an agent, e.g. transitive reasoning. However one drawback is that the context lengths are quite short, a couple of sentences, and the language is very simplistic. The text does not emulate the challenges of memory in the real world, that is they do not incorporate noise, a natural curriculum of temporal range, nor the often indirect purpose of memory. In the next section we discuss the modelling of real-world text as another driver of memory research.

Open-Book QA

Much of the focus on question answering from natural data has focused on open-book QA. In the open-book question answering formulation we have a triplet of (context, query, answer) with the intention that an answer should be plausibly obtained from the given context. Typically the context is not very long, a paragraph extract or more recently a collection of articles up to several thousand words.

Within this broad category there has been a progression of question answering from **closed domain** tasks where the answers are always constrained to be words or entities from the context. This has then transitioned to **open domain** question answering benchmarks which usually start with a set of naturally occuring questions, e.g. from web searches, and they provide a set of contexts which are deemed to be helpful, with answers that may a single word or even a sentence. At the beginning of this study the prominent question answering benchmarks were closed-domain and took the form of a Cloze task (Taylor, 1953).

Closed-Domain QA with Cloze

In the Cloze formulation, the model with a body of context text and an additional sentence (the query) with one or more missing words that need to be predicted (the answer). The CNN/Daily Mail dataset (Hermann et al., 2015) uses news articles as context, LAMBADA (Paperno et al., 2016) and the Children's Book Test (Hill et al., 2015) uses book text, SQUAD Ȧn example of a Daily Mail context, query and answer triplet is shown in Table 2.2. The classification of these tasks as QA is fuzzy, these Cloze-formulated benchmarks can also be considered to be regular masked language modelling tasks or even summarisation tasks in the case

Table 2.2: Example context, query and answer from the Daily Mail dataset (Hermann et al., 2015).

Context: The BBC producer allegedly struck by Jeremy Clarkson will not press charges against the "Top Gear" host, his lawyer said Friday. Clarkson, who hosted one of the most-watched television shows in the world, was dropped by the BBC Wednesday after an internal investigation by the British broadcaster found he had subjected producer Oisin Tymon "to an unprovoked physical and verbal attack." ...
Query: Producer X will not press charges against Jeremy Clarkson, his lawyer says.
Answer: Oisin Tymon

of CNN/Daily Mail, since for this dataset the held-out terms (answers) are all sourced from article summaries.

The motivation behind these benchmarks is to discern models with better reading comprehension, an ability to process some stream of input and display an understanding of its contents by answering or completing carefully constructed questions. For example LAMBADA chooses book extracts that human raters consider to have a clear correct completion with a longer context but also a compelling but incorrect completion if one considers only the local context. CNN/Daily Mail chooses queries that are picked from bulleted summaries of the articles. However it is difficult to construct probes for true comprehension. An analysis by Chen et al. (2016) showed that from a sample of 100 datapoints from the CNN/Daily Mail benchmark only 2% required reasoning across more than 1 sentence, with 54% being either exact or paraphrased by a sentence in the context, 8% containing incorrect answers, and 17% being considered ambiguous.

The types of architectures that have demonstrated best performance for these types of question answering benchmarks have been models with a very simple

attention mechanism. One of which is the Attentive Reader (Hermann et al., 2015) which is a variant of a Memory Network (Weston et al., 2015). The Attentive Reader encodes both the query and document tokens using a bi-directional RNN, computes a content-based attention score between the query and each document token feeds the query and attention result through an MLP to give the predicted answer token. Better performance was later obtained by simplifying the memory scheme to attend and retrieve tokens directly instead of embeddings; this model is called the Attention Sum Reader (Kadlec et al., 2016).

The Attention Sum Reader can be briefly written down as computing

$$P(w|q,c) \propto \sum_{i=1}^{n} \mathbb{I}\{c_i = w\} e^{f(q)^T g(c_i)} \tag{2.8}$$

where w is a candidate answer, c_i is the i-th token in the context, q is the query string and f and g are neural network encoders that map a string query and context token respectively to fixed-sized embedding with d dimensions. The model uses a very similar token-level attention scheme as the previously mentioned Neural Cache (2.7). The Attention Sum Reader contains a key, value memory; the key being a document token embedding $g(c_i)$ the value being the document token c_i, and the query being a fixed-size embedding of the question $f(q)$. Thus this model can only use its memory to copy a token from the context. This model is actually an instance of a Pointer Network (Vinyals et al., 2015), which is a general encoder-decoder model that uses attention to copy from its stream of inputs, however it is specialised to QA.

At the time of initiating this study the Attention Sum Reader was state-of-the-art

across the Children's Book Test, Daily Mail and CNN benchmarks despite being much simpler than existing memory models in construction (e.g. versus the Attentive Reader, or even the Neural Turing Machine). This was another indication, prior to the dataset analysis by Chen et al. (2016) that these datasets favour models which can copy a particular word from the context with some contextual clues from the query, but such models are, by construction, limited in their ability to reason over time. For example, the Attention Sum Reader would not be able to count the occurrence of a mentioned entity, or transitively infer the relationship between two disparately mentioned entities.

Due to the emphasis on single-term retrieval versus temporal reasoning over long time-scales for these benchmarks, the incorporation of question answering to benchmark lifelong reasoning must be treated with careful deliberation. We want to pursue a model that can read a book, for example, and answer many questions on its contents that require multi-hop reasoning and require the model to compress salient pieces of information.

Open Domain QA

Several question answering benchmarks have tested open-domain question answering capability where a selection of questions is gathered, e.g. from web searches, and relevant contexts and answers are coupled together. Such datasets in this category are WikiQA (Yang et al., 2015) which takes search queries from Bing and matches them to relevant extracts from Wikipedia; SQuAD (Rajpurkar et al., 2016) which takes paragraphs from Wikipedia and crowd-sources questions and answers based on the content, TriviaQA (Joshi et al., 2017) which sourced questions

and answers from trivia sites and contexts from Bing and a Wikipedia entity-matching search, HotpotQA (Yang et al., 2018) from crowd-sourced questions and Wikipedia contexts with a filtering system to isolate difficult multi-hop reasoning questions, and Natural Questions (Kwiatkowski et al., 2019) which pairs queries, contexts and answers from Google Search results.

These datasets are making progress on the front of requiring more complex reasoning. For example, from CNN/DailyMail's estimated 2% of questions which require multi-sentence reasoning, SQuAD is estimated to contain 14%, and HotPotQA it is estimated to be over 90%. They also push dataset scale and context lengths, from sentences in WikiQA with less than 100 words on average to entire Wikipedia pages in Natural Questions with thousands of words on average. The encouraging progression we are seeing from open domain QA is that progress is being made by very recent transformer variants which incorporate deep and sparse (via static attention sparsity) and compressive (via global attention) memory systems, such as BigBird (Zaheer et al., 2020) and the LongFormer (Beltagy et al., 2020). Thus whilst simplifying the memory architecture improved performance for the closed-domain Cloze QA tasks that were present at the inception of this study, we are seeing better tasks now drive better architectures. Some of these datasets can be approached without the open-book formulation, i.e. where a relevant context is required. New neural network architectures such as REALM (Guu et al., 2020) and RAG (Lewis et al., 2020) do not make use of the provided context page but search for relevant contexts from a large knowledge base, such as Wikipedia. These very recently created systems, formulated at the point this study had been concluded, combines many architectural components that this study finds necessary for life-

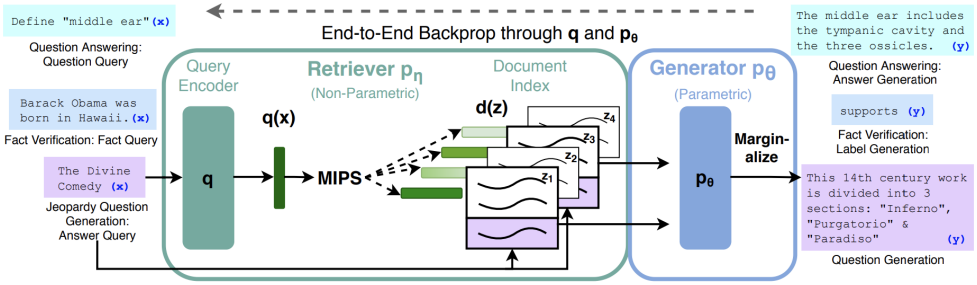


Figure 2.12: A schematic of the Retrieval-Augmented Generation (RAG) approach from (Lewis et al., 2020). Here MIPS refers to Maximum-Inner Product Search, an instance of Approximate Nearest Neighbour Search discussed in Chapter 3. The model combines a large-scale sparse retrieval system with a granular reasoning over the retrieved documents for answer generation.

long reasoning: approximate nearest neighbour indexing for efficient neural search, sparse attention of documents, compression via transformers to create small document embeddings and more granular attention over relevant documents. Thus question answering is not only morphing from closed-domain open-book question answering to open-domain, but it is removing the contextual supervision entirely to create more and more powerful memory architectures and retrieval systems.

Closed-Book Question Answering

One future direction for question answering which has a small but growing uptake near this study's conclusion is closed-book question answering. In contrast to starting with a question and scanning through the context to look for relevant key-words, which emphasizes the information-retrieval component of memory, closed-book question answering maps directly from question to answer. Thus when processing context, the model must compress salient pieces of information with much less supervision over what to store. This more closely resembles the role

of episodic memory in the human brain, where we choose what to store up-front without knowing what we will need to remember in future.

The most powerful closed-book question answering system at present is a large-scale language model tuned or primed on question answering data (Brown et al., 2020; Raffel et al., 2020). These models are able to store and recall a huge quantity of information and facts from pre-training in the model parameters, and flexibly access this to answer unseen questions. However this process moreso resembles the semantic memory in Tulving's classification versus episodic memory.

If we give a model sufficient storage capacity and compute, the harder problem of closed-book question answering boils down to open-book question answering, as the model can simply store everything it has seen in an exact form before encountering its query. Then upon reading the query, it can search over its uncompressed knowledge base using an information-retrieval system such as RAG or REALM. Whilst this may seem contrived, this becomes more and more likely with language models whose parameters' capacity far exceeds the training set size.

An interesting benchmark that does not appear to exist at present, but would stress the type of lifelong episodic memory architecture of interest, would be a closed-book question answering task where the model is constrained in the size of its memory. The model would read a book, for example, and then answer questions about the content but it would be forced to store only a small amount of data from the book — inhibiting it from copying it. This would force the model to learn a general compression of its memory for many potential questions or reasoning tasks.

In conclusion, this thesis has not relied heavily on question answering simply because the right kinds of benchmarks and datasets were not present at its inception. However the pace of benchmark creation has picked up and question answering tasks now appear to be motivating lifelong reasoning architectures. Thus it has much future promise.

Chapter 3

Scaling Memory with Sparsity

Neural networks augmented with external memory have the ability to learn algorithmic solutions to complex tasks. These models appear promising for applications such as language modelling and machine translation. However when trained with soft attention over the entire memory, their compute scales linearly or even quadratically as the amount of memory grows. This in turn forces us to use small memories which cover only a small window of time from the past — limiting their applicability to real-world domains. In this chapter we propose an end-to-end differentiable memory access scheme, which we call Sparse Access Memory (SAM), that retains the representational power of the original approaches whilst training efficiently with very large memories.

We show that SAM achieves asymptotic lower bounds in space and time complexity, reaching logarithmic time to access memory and consuming only a constant amount of additional space per training step. Empirically, we find that an implementation runs 1,000× faster and with 3,000× less physical memory than non-

sparse models. SAM learns with comparable data efficiency to existing models on a range of synthetic tasks and few-shot Omniglot character recognition, and can scale to tasks requiring 100,000s of time steps and memories. As well, we show how our approach can be adapted for models that maintain temporal associations between memories, as with the recently introduced Differentiable Neural Computer (DNC), proposing another sparse model — the Sparse Differentiable Neural Computer (SDNC). We find the SDNC achieved state-of-the-art performance in the bAbI question answering task, outperforming the DNC, Memory Networks, and LSTM on this benchmark. Thus we conclude sparsity can not only help scale memory systems, but it can also improve learning.

3.1 Motivation

Recurrent neural networks, such as the Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997), have proven to be powerful sequence learning models (Graves et al., 2013; Sutskever et al., 2014). However, one limitation of the LSTM architecture is that the number of parameters grows proportionally to the square of the size of the memory, making them unsuitable for problems requiring large amounts of long-term memory. Recent approaches, such as Neural Turing Machines (NTMs) (Graves et al., 2014) and Memory Networks (Weston et al., 2014), have addressed this issue by decoupling the memory capacity from the number of model parameters. We refer to this class of models as memory augmented neural networks (MANNs). External memory allows MANNs to learn algorithmic solutions to problems that have eluded the capabilities of traditional LSTMs, and to generalise to longer sequence lengths. Nonetheless, MANNs have had limited

success in real world application.

A significant difficulty in training these models results from their smooth read and write operations, which incur linear computational overhead on the number of memories stored per time step of training. Even worse, they require duplication of the entire memory at each time step to perform backpropagation through time (BPTT). To deal with sufficiently complex problems, such as processing a book, or Wikipedia, this overhead becomes prohibitive. For example, to store 64 memories, a straightforward implementation of the NTM trained over a sequence of length 100 consumes $\approx 30\,\text{MiB}$ physical memory; to store 64,000 memories the overhead exceeds 29 GiB (see Figure 3.3).

In this chapter, we present a MANN named SAM (sparse access memory). By thresholding memory modifications to a sparse subset, and using efficient data structures for content-based read operations, our model is optimal in space and time with respect to memory size, while retaining end-to-end gradient based optimisation. To test whether the model is able to learn with this sparse approximation, we examined its performance on a selection of synthetic and natural tasks: algorithmic tasks from the NTM work (Graves et al., 2014), bAbI reasoning tasks used with Memory Networks (Sukhbaatar et al., 2015) and Omniglot few-shot classification (Santoro et al., 2016a; Lake et al., 2015). We also tested several of these tasks scaled to longer sequences via curriculum learning. For large external memories we observed improvements in empirical run-time and memory overhead by up to three orders magnitude over vanilla NTMs, while maintaining near-identical data efficiency and performance.

Further, in Section 3.5 we demonstrate the generality of our approach by de-

scribing how to construct a sparse version of the recently published Differentiable Neural Computer (Graves et al., 2016). This Sparse Differentiable Neural Computer (SDNC) is over 400× faster than the canonical dense variant for a memory size of 2,000 slots, and achieves the best reported result in the bAbI tasks without supervising the memory access.

3.2 Architecture

This section introduces *Sparse Access Memory (SAM)*, a new neural memory architecture with two innovations. Most importantly, all reads and writes to external memory are constrained to a sparse subset of the memory words, providing similar functionality as the NTM, while allowing computational and memory efficient operation. Secondly, we introduce a sparse memory management scheme that tracks memory usage and finds unused blocks of memory for recording new information.

For a memory containing N words, SAM executes a forward, backward step in $\Theta(\log N)$ time, initialises in $\Theta(N)$ space, and consumes $\Theta(1)$ space per time step. Under some reasonable assumptions, SAM is asymptotically optimal in time and space complexity (Section 3.3).

3.2.1 Read

The sparse read operation is defined to be a weighted average over a selection of words in memory:

$$\tilde{r}_t = \sum_{i=1}^{K} \tilde{a}_t^R(s_i)\mathbf{M}_t(s_i), \tag{3.1}$$

where $\tilde{a}_t^R \in \mathbb{R}^N$ contains K number of non-zero entries with indices $s_1, s_2, \ldots, s_K$; K is a small constant, independent of N, typically $K = 4$ or $K = 8$. We will refer to sparse analogues of weight vectors w as $\tilde{w}$, and when discussing operations that are used in both the sparse and dense versions of our model use w.

3.2.2 Sparse Attention

In general we would like to construct a sparse attention vector which preserves the read contents, i.e. $\tilde{r}_t \approx r_t$. For content-based reads where a_t^R is defined by (2.4), an effective approach is to keep the K largest non-zero entries and set the remaining entries to zero. We re-normalise the non-zero components to keep the attention sum to one.

The sparse attention operator is no longer uniformly continuous as it has a point of discontinuity for queries which lie on the voronoi boundary of their K-nearest neighbours. We could consider different ways of inducing sparsity, such as applying an L1 penalty to the attention weights such that many are pushed to zero naturally. We could also consider using the *SparseMax* (Martins and Astudillo, 2016) which projects the logits onto the closest point lying on the probability simplex. Both of these methods povide a differentiable and principled way of achieving sparsity

however they both have the drawback in being linear-time operations, with respect to the memory size N.

Sub-linear methods can be used to approximately find the K largest attention values. Whilst we can compute $\tilde{a}_t^R$ naively in $\mathcal{O}(N)$ time by calculating a_t^R and keeping the K largest values. Since the K largest values in a_t^R correspond to the K closest points to our query q_t, we can use an approximate nearest neighbour data-structure, described in Section 3.2.7, to calculate $\tilde{a}_t^R$ in $\mathcal{O}(\log N)$ time.

The sparse read can be considered a special case of the matrix-vector product defined in (2.3), with two key distinctions. The first is that we pass gradients for only a constant K number of rows of memory per time step, versus N, which results in a negligible fraction of non-zero error gradient per timestep when the memory is large. The second distinction is in implementation: by using an efficient sparse matrix format such as Compressed Sparse Rows (CSR), we can compute (3.1) and its gradients in constant time and space (see Section 3.3).

3.2.3 Write

The write operation is SAM is an instance of (2.5) where the write weights $\tilde{a}_t^{\mathbf{W}}$ are constrained to contain a constant number of non-zero entries.

$$M_t \leftarrow (1 - \tilde{a}_t^{\mathbf{W}} e_t^T) \odot M_{t-1} + \tilde{a}_t^{\mathbf{W}} w_t^T \ . \tag{3.2}$$

This controller writes either to the *previously read locations*, in order to update contextually relevant memories, or the *least recently accessed* location, in order to overwrite stale or unused memory slots with fresh content. This write scheme is

inherited from the Neural Turing Machine and the idea behind it is that we may want to update some recent information or allocate a new slot in memory. When we overwite a slot in memory the question arises, which slot should we garbage collect? Taking inspiration from cache data structures, this is chosen to be the slot which has been least 'used' where the usage will be defined formally in (3.4).

It is worth noting that in the case where the maximum number of slots is small in comparison to the sequence length, this *least usage* mechanism can counterract the learning of long-range dependencies for data that would not be read for long spans of time. This could be counter-acted by using a learned saliency that can decidedly store a class of input for longer stretches of time, however for the class of problems we consider we find this cache-inspired system works sufficiently well.

The introduction of sparsity could be achieved via other write schemes. For example, we could use a sparse content-based write scheme, where the controller chooses a query vector $q_t^{\mathbf{W}}$ and applies writes to similar words in memory. This would allow for direct memory updates, but would create problems when the memory is empty (and shift further complexity to the controller). We decided upon the previously read / least recently accessed addressing scheme for simplicity and flexibility.

The write weights are defined as

$$a_t^{\mathbf{W}} = \alpha_t \left(\gamma_t \, a_{t-1}^R + (1 - \gamma_t) \, \mathbb{I}_t^U \right) , \tag{3.3}$$

where the controller outputs the interpolation gate parameter γ_t and the write gate parameter α_t. The write to the previously read locations a_{t-1}^R is purely additive, while the least recently accessed word $\mathbb{I}_t^U$ is set to zero before being written to.

When the read operation is sparse (a^R_{t-1} has K non-zero entries), it follows the write operation is also sparse.

We define $\mathbb{I}^U_t$ to be an indicator over words in memory, with a value of 1 when the word minimises a usage measure U_t

$$\mathbb{I}^U_t(i) = \begin{cases} 1 \quad \text{if } U_t(i) = \min\limits_{j=1,\ldots,N} U_t(j) \\ 0 \text{ otherwise.} \end{cases} \tag{3.4}$$

If there are several words that minimise U_t then we choose arbitrarily between them. We tried two definitions of U_t. The first definition is a time-discounted sum of write weights $U^{(1)}_T(i) = \sum_{t=0}^{T} \lambda^{T-t} \left(a^{\mathbf{W}}_t(i) + a^R_t(i)\right)$ where λ is the discount factor.

This usage definition is incorporated within *Dense Access Memory* (DAM), a dense-approximation to SAM that is used for experimental comparison in Section 3.4.

The second usage definition, used by SAM, is simply the number of time-steps since a non-negligible memory access: $U^{(2)}_T(i) = T - \max\left\{\, t : a^{\mathbf{W}}_t(i) + a^R_t(i) > \delta \right\}$. Here, δ is a tuning parameter that we typically choose to be 0.005. We maintain this usage statistic in constant time using a custom data-structure (described in Section 3.3). Finally we also use the least recently accessed word to calculate the erase matrix. $\mathbf{R}_t = \mathbb{I}^U_t \mathbf{1}^T$ is defined to be the expansion of this usage indicator where $\mathbf{1}$ is a vector of ones. The total cost of the write is constant in time and space for both the forwards and backwards pass, which improves on the linear space and time dense write.

3.2.4 Setting the Memory Size N vs T

We denote N to be the maximum memory size, this value does not need to equal the sequence length T. In general the dense models which we compare to, such as DAM, the NTM, and DNC choose N to be much smaller than T due to computational and space limitations, however for SAM we can set N to be equal to T, and dynamically allocate memory slots as they are needed. The sparsity of the write scheme allows us to distinguish between unwritten memory slots and written memory slots, which means after ten time-steps the model knows to only attend to the first ten slots in memory, even if the total sequence length and memory capacity is in the thousands.

In this chapter we always consider episodic tasks where the memory is reset between episodes and we backpropagate-through-time over the whole sequence. If this were not the case, we would also posit that SAM would be well-suited to non-episodic tasks where memory is preserved across many episode boundaries as this would require a large memory slot budget.

3.2.5 Controller

We use a one layer LSTM for the controller throughout. At each time step, the LSTM receives a concatenation of the external input, x_t, the word, r_{t-1} read in the previous time step. The LSTM then produces a vector, $p_t = (q_t, w_t, \alpha_t, \gamma_t)$, of read and write parameters for memory access via a linear layer. The word read from memory for the current time step, r_t, is then concatenated with the output of the LSTM, and this vector is fed through a linear layer to form the final output, y_t. The full control flow is illustrated in Figure 3.1.

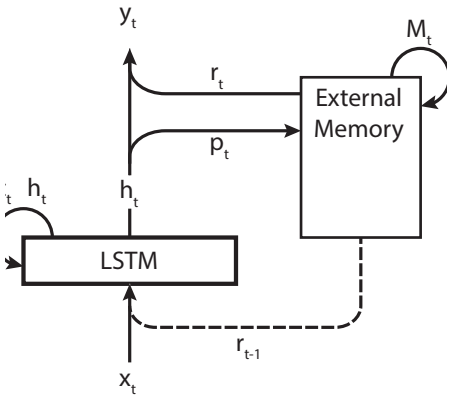


Figure 3.1: Schematic showing how the controller interfaces with the external memory in our experiments. The controller (LSTM) output h_t is used (through a linear projection, p_t) to read and write to the memory. The result of the read operation r_t is combined with h_t to produce output y_t, as well as being feed into the controller at the next timestep (r_{t-1}).

3.2.6 Efficient backpropagation through time

We have already demonstrated how the forward operations in SAM can be efficiently computed in $\mathcal{O}(T \log N)$ time. However, when considering space complexity of MANNs, there remains a dependence on $\mathbf{M_t}$ for the computation of the derivatives at the corresponding time step. A naive implementation requires the state of the memory to be cached at each time step, incurring a space overhead of $\mathcal{O}(NT)$, which severely limits memory size and sequence length.

Fortunately, this can be remedied. Since there are only $\mathcal{O}(1)$ words that are written at each time step, we instead track the sparse modifications made to the memory at each timestep, apply them in-place to compute $\mathbf{M_t}$ in $\mathcal{O}(1)$ time and $\mathcal{O}(T)$ space. During the backward pass, we can restore the state of M_t from M_{t+1} in $\mathcal{O}(1)$ time by reverting the sparse modifications applied at time step t. As such the memory is actually rolled back to previous states during backpropagation (Figure 3.2).

At the end of the backward pass, the memory ends rolled back to the start state. If required, such as when using truncating BPTT, the final memory state can be restored by making a copy of $\mathbf{M_T}$ prior to calling backwards in $\mathcal{O}(N)$ time, or by re-applying the T sparse updates in $\mathcal{O}(T)$ time.

3.2.7 Approximate nearest neighbours

When querying the memory, we can use an approximate nearest neighbour index (ANN) to search over the external memory for the K nearest words. Where a linear KNN search inspects every element in memory (taking $\mathcal{O}(N)$ time), an ANN index maintains a structure over the dataset to allow for fast inspection of nearby points in $\mathcal{O}(\log N)$ time.

In our case, the memory is still a dense tensor that the network directly operates on; however the ANN is a structured view of its contents. Both the memory and the ANN index are passed through the network and kept in sync during writes. However there are no gradients with respect to the ANN as its function is fixed.

We considered two types of ANN indexes: FLANN's randomised k-d tree implementation (Muja and Lowe, 2014) that arranges the data points in an ensemble of structured (randomised k-d) trees to search for nearby points via comparison-based search, and one that uses locality sensitive hash (LSH) functions that map points into buckets with distance-preserving guarantees. We used randomised k-d trees for small word sizes and LSHs for large word sizes. For both ANN implementations, there is an $\mathcal{O}(\log N)$ cost for insertion, deletion and query. We also rebuild the ANN from scratch every N insertions to ensure it does not become

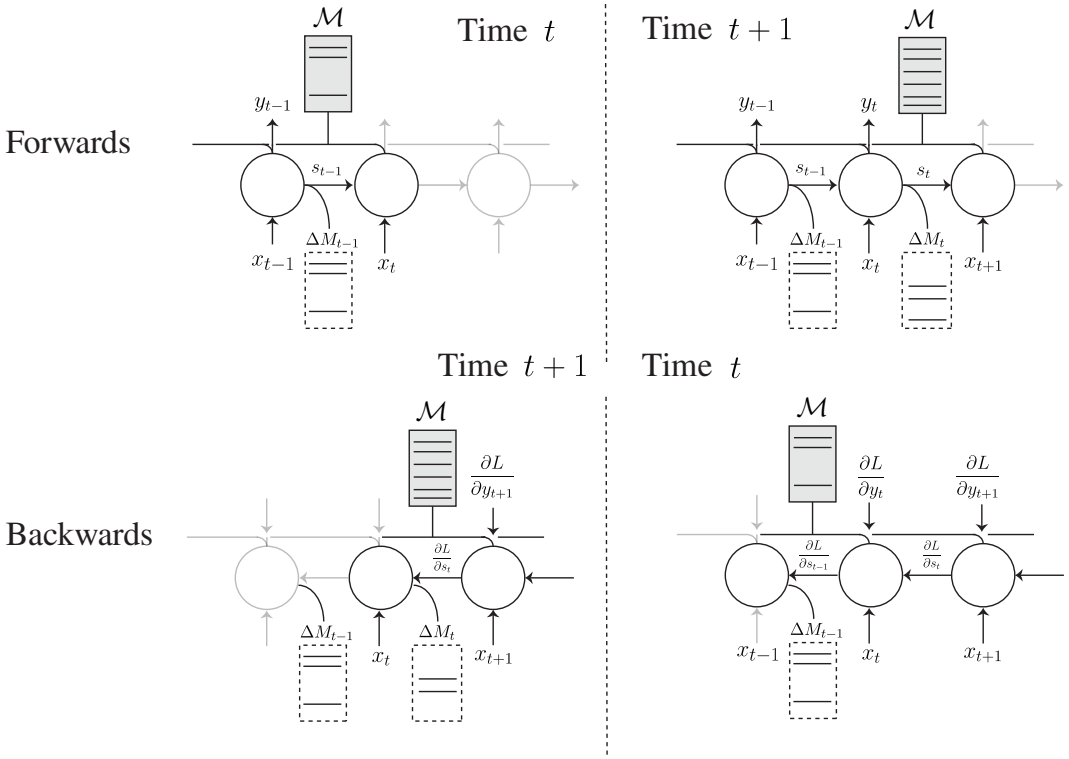


Figure 3.2: A schematic of the memory efficient backpropagation through time. Each circle represents an instance of the SAM core at a given time step. The grey box marks the dense memory. Each core holds a reference to the single instance of the memory, and this is represented by the solid connecting line above each core. We see during the forward pass, the memory's contents are modified sparsely, represented by the solid horizontal lines. Instead of caching the changing memory state, we store only the sparse modifications — represented by the dashed white boxes. During the backward pass, we "revert" the cached modifications to restore the memory to its prior state, which is crucial for correct gradient calculations.

imbalanced.

3.3 Time and space complexity

In this section we show that SAM requires $\mathcal{O}(\log n)$ time and $\mathcal{O}(1)$ space per training step. We argue that under a reasonable class of content addressable memory architectures, SAM is optimal in time and space complexity.

3.3.1 Initialisation

Upon initialisation, SAM consumes $\mathcal{O}(N)$ space and time to instantiate the memory and the memory Jacobian. Furthermore, it requires $\mathcal{O}(N)$ time and space to initialise auxiliary data structures which index the memory, such as the approximate nearest neighbour which provides a content-structured view of the memory, and the least accessed ring, which maintains the temporal ordering in which memory words are accessed. These initialisations represent an unavoidable one-off cost that does not recur per step of training, and ultimately has little effect on training speed. For the remainder of the analysis we will concentrate on the space and time cost per training step.

3.3.2 Complexity of read

Recall the sparse read operation,

$$\tilde{r}_t = \sum_{i=1}^{K} \tilde{w}_t^R(s_i)\mathbf{M}_t(s_i) \ . \tag{3.5}$$

As K is chosen to be a fixed constant, it is clear we can compute (3.5) in $\mathcal{O}(1)$ time despite having a much larger $\mathcal{O}(N)$ capacity. It's worth noting that N could be constrained to be of constant size, which means the read would also be constant-time, however the resulting memory would only be able to store a small number of inputs. During the backward pass, we see the gradients are sparse with only K non-zero terms,

$$\frac{\partial L}{\partial \tilde{a}_t^R}(i) = \begin{cases} \mathbf{M}_t(i) \cdot \frac{\partial L}{\partial \tilde{r}_t} & \text{if } i \in \{s_1, s_2, \ldots, s_K\} \\ 0 & \text{otherwise.} \end{cases}$$

and

$$\frac{\partial L}{\partial M_t}(i) = \begin{cases} \tilde{a}_t^R(i) \frac{\partial L}{\partial \tilde{r}_t} & \text{if } i \in \{s_1, s_2, \ldots, s_K\} \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

where $\mathbf{0}$ is a vector of M zeros. Thus they can both be computed in constant time by skipping the computation of zeros. Furthermore by using an efficient sparse matrix format to store these matrices and vectors, such as the CSR, we can represent them using at most $3K$ values. Since the read word $\tilde{r}_t$ and its respective error gradient is the size of a single word in memory (M elements), the overall space complexity is $\mathcal{O}(1)$ per time step for the read.

3.3.3 Complexity of write

Recall the write operation,

$$\mathbf{M}_t \leftarrow \mathbf{M}_{t-1} - \mathbf{E}_t + \mathbf{A}_t, \ , \tag{3.6}$$

where $\mathbf{A}_t = a_t^{\mathbf{W}} w_t^T$ is the add matrix, $\mathbf{E_t} = \mathbf{M}_{t-1} \odot \mathbf{R}_t$ is the erase matrix, and $\mathbf{R}_t = \mathbb{I}_t^U \mathbf{1}^T$ is defined to be the erase weight matrix. We chose the write weights to be an interpolation between the least recently accessed location and the previously read locations,

$$a_t^{\mathbf{W}} = \alpha_t \left(\gamma_t \, \tilde{a}_{t-1}^R + (1 - \gamma_t) \, \mathbb{I}_t^U \right) . \tag{3.7}$$

For sparse reads where $a_t^R = \tilde{a}_t^R$ is a sparse vector with K non-zeros, the write weights $a_t^{\mathbf{W}}$ is also sparse with $K + 1$ non-zeros: 1 for the least recently accessed location and K for the previously read locations. Thus the sparse-dense outer product $\mathbf{A_t} = a_t^{\mathbf{W}} w_t^T$ can be performed in $\mathcal{O}(1)$ time as K is a fixed constant.

Since $\mathbf{R}_t = \mathbb{I}_t^U \mathbf{1}^T$ can be represented as a sparse matrix with one single non-zero, the erase matrix $\mathbf{E}_t$ can also. As $\mathbf{A_t}$ and $\mathbf{E_t}$ are sparse matrices we can then add them component-wise to the dense $\mathbf{M_{t-1}}$ in $\mathcal{O}(1)$ time. By analogous arguments the backward pass can be computed in $\mathcal{O}(1)$ time and each sparse matrix can be represented in $\mathcal{O}(1)$ space.

We avoid caching the modified memory, and thus duplicating it, by applying the write directly to the memory. To restore its prior state during the backward pass, which is crucial to gradient calculations at earlier time steps, we roll the memory back by reverting the sparse modifications with an additional $\mathcal{O}(1)$ time overhead (displayed in Figure 3.2).

The location of the least recently accessed memory can be maintained in $\mathcal{O}(1)$ time by constructing a circular linked list that tracks the indices of words in memory, and preserves a strict ordering of relative temporal access. This uses $\mathcal{O}(1)$ space

per time-step. The first element in the ring is the least recently accessed word in memory, and the last element in the ring is the most recently modified. We keep a "head" pointer to the first element in the ring. When a memory word is randomly accessed, we can push its respective index to the back of the ring in $\mathcal{O}(1)$ time by redirecting a small number of pointers. When we pop the least recently accessed memory (and write to it) we move the head to the next element in the ring in $\mathcal{O}(1)$ time.

3.3.4 Content-based addressing in $\mathcal{O}(\log N)$ time

As discussed in Section 3.2.7 we can calculate the content-based attention, or read weights a_t^R, in $\mathcal{O}(\log N)$ time using an approximate nearest neighbour index that views the memory. We keep the ANN index synchronised with the memory by passing it through the network as a non-differentiable member of the network's state (so we do not pass gradients for it), and we update the index upon each write or erase to memory in $\mathcal{O}(\log N)$ time. Maintaining and querying the ANN index represents the most expensive part of the network, which is reasonable as content-based addressing is inherently expensive (Motwani et al., 2007; Arya et al., 1998).

For the backward pass computation, specifically calculating $\frac{\partial L}{\partial q_t}$ and $\frac{\partial L}{\partial \mathbf{M_t}}$ with respect to a_t^R, we can once again compute these using sparse matrix operations in $\mathcal{O}(1)$ time. This is because the K non-zero locations have been determined during the forward pass.

Thus to conclude, SAM consumes in total $\mathcal{O}(1)$ space for both the forward and backward step during training, $\mathcal{O}(\log N)$ time per forward step, and $\mathcal{O}(1)$ per

backward step.

3.3.5 Optimality

We remark that SAM is optimal in terms of space and time complexity, within the class of content addressable memory architectures that are able to retieve memories that are within some ϵ distance away from a given query.

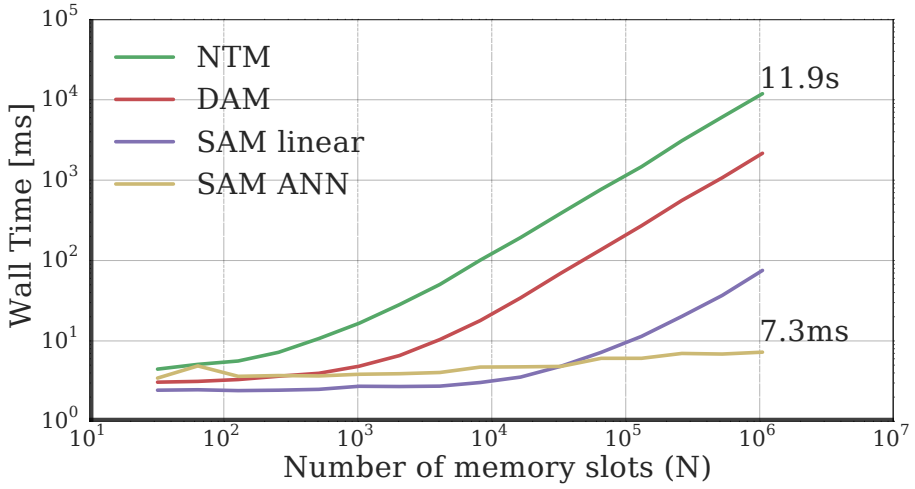
This follows from existing lower bounds, Motwani et al. (2007); Arya et al. (1998) assert that for any data structure which stores N elements and is able to return the memory that is within a factor of ϵ off from the truly nearest neighbour, it must require $\mathcal{O}(\log N)$ time per query. The optimality of SAM falls from the fact that it uses an asymptotically optimal approximate nearest neighbour search, and all other operations are constant space and time per unit memory.

What would it mean for an alternative memory-augmented neural network to improve upon this, and have constant-time per unit memory? This would imply it cannot find the approximate nearest neighbour in general. It is plausible this could be enough for a reasonable class of problems, i.e. auto-association would be possible if the query could only vary from a given memory in a small discrete number of ways that was independent of the memory capacity. The nework would learn a constant number of transformations over the query to determine if it exactly matches a known memory. Alternatively it would be a memory that retains only the recent $\mathcal{O}(1)$ memories in practice, however in that case the constant-time query mechanism is a misnomer, and the memory does not truly store $\mathcal{O}(N)$ memories.

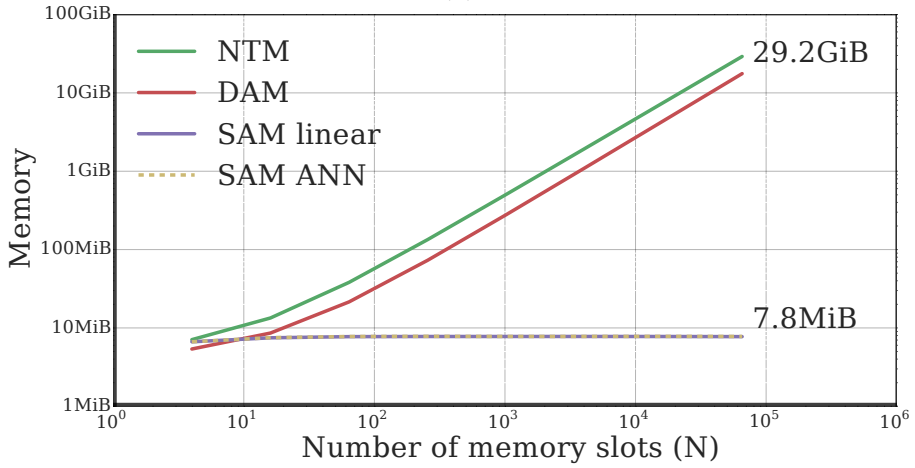
3.4 Results

3.4.1 Speed and memory benchmarks

We measured the forward and backward times of the SAM architecture versus the dense DAM variant and the original NTM and plot the results in Figure 3.3 when run on a multi-core CPU. The full details of the benchmark setup are detailed in Appendix A.2). We find SAM is over 100 times faster than the NTM when the memory contains one million words and an exact linear-index is used, and 1600 times faster with the k-d tree (Figure 3.3a). With an ANN the model runs in sub-linear time with respect to the memory size. SAM's memory usage per time step is independent of the number of memory words (Figure 3.3b), which empirically verifies the $\mathcal{O}(1)$ space claim from Section 3.3. For 64 K memory words SAM uses 53 MiB of physical memory to initialise the network and 7.8 MiB to run a 100 step forward and backward pass, compared with the NTM which consumes 29 GiB.



(a)



(b)

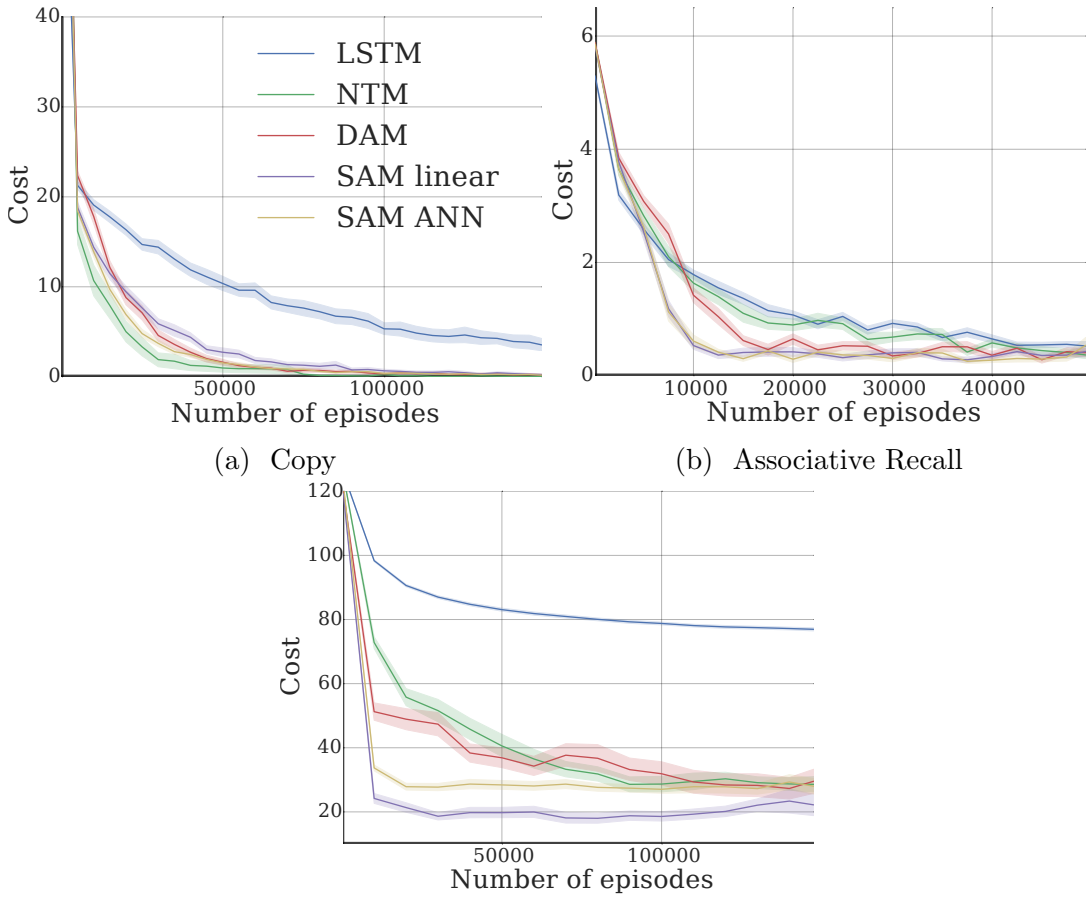
Figure 3.3: **(a)** Wall-clock time of a single forward and backward pass. The k-d tree is a FLANN randomised ensemble with 4 trees and 32 checks. For 1M memories a single forward and backward pass takes 12 s for the NTM and 7 ms for SAM, a speedup of 1600×. **(b)** Memory used to train over sequence of 100 time steps, excluding initialisation of external memory. The space overhead of SAM is independent of memory size, which we see by the flat line. When the memory contains 64,000 words the NTM consumes 29 GiB whereas SAM consumes only 7.8 MiB, a compression ratio of 3700.

3.4.2 Learning with sparse memory access

We have established that SAM reaps a huge computational and memory advantage of previous models, but can we really learn with SAM's sparse approximations? We investigated the learning cost of inducing sparsity, and the effect of placing an approximate nearest neighbour index within the network, by comparing SAM with its dense variant DAM and some established models, the NTM and the LSTM.

We trained each model on three of the original NTM tasks (Graves et al., 2014). **1. Copy**: copy a random input sequence of length 1–20, **2. Associative Recall**: given 3-6 random (key, value) pairs, and subsequently a cue key, return the associated value. **3. Priority Sort**: Given 20 random keys and priority values, return the top 16 keys in descending order of priority. We chose these tasks because the NTM is known to perform well on them. Full hyper-parameter details are in Appendix A.1.

Figure 3.4 shows that sparse models are able to learn with comparable efficiency to the dense models and, surprisingly, learn more effectively for some tasks — notably priority sort and associative recall. This shows that sparse reads and writes can actually benefit early-stage learning in some cases. This may be because sparsity can reduce the variance of precision of writes and reads, which makes it easier to precisely read back stored content. In contrast to soft writes and reads where every row is written to by every memory with some weight (potentially very close to zero), and every row is read from by every query with some weight, there is more interference and noise.



(a) Copy

(b) Associative Recall

(c) Priority Sort

Figure 3.4: Training curves for sparse (SAM) and dense (DAM, NTM) models. SAM trains comparably for the Copy task, and reaches asymptotic error significantly faster for Associative Recall and Priority Sort. Light colours indicate one standard deviation over 30 random seeds.

3.4.3 Scaling with a curriculum

The computational efficiency of SAM opens up the possibility of training on tasks that require storing a large amount of information over long sequences. Here we show this is possible in practice, by scaling tasks to a large scale via an exponentially increasing curriculum.

We parametrised three of the tasks described in Section 3.4.2: associative recall,

copy, and priority sort, with a progressively increasing difficulty level which characterises the length of the sequence and number of entries to store in memory. For example, level specifies the input sequence length for the copy task.

Since the time taken for a forward and backward pass scales $\mathcal{O}(T)$ with the sequence length T, following a standard linearly increasing curriculum could potentially take $\mathcal{O}(T^2)$, if the same amount of training was required at each step of the curriculum. As such we used an exponentially-increasing curriculum.

Specifically, h was doubled whenever the average training loss dropped below a threshold for a number of episodes. The level was sampled for each minibatch from the uniform distribution over integers $\mathcal{U}(0, h)$. We compared the dense models, NTM and DAM, with both SAM with an exact nearest neighbour index (SAM linear) and with locality sensitive hashing (SAM ANN). The dense models contained 64 memory words, while the sparse models had 2×10^6 words. These sizes were chosen to ensure all models use approximately the same amount of physical memory when trained over 100 steps.

For all tasks, SAM was able to advance further than the other models, and in the associative recall task, SAM was able to advance through the curriculum to sequences greater than 4000 (Figure 3.5). Note that we did not use truncated backpropagation, so this involved BPTT for over 4000 steps with a memory size in the millions of words.

To investigate whether SAM was able to learn algorithmic solutions to tasks, we investigated its ability to generalise to sequences that far exceeded those observed during training. We trained SAM on the associative recall task up to sequences

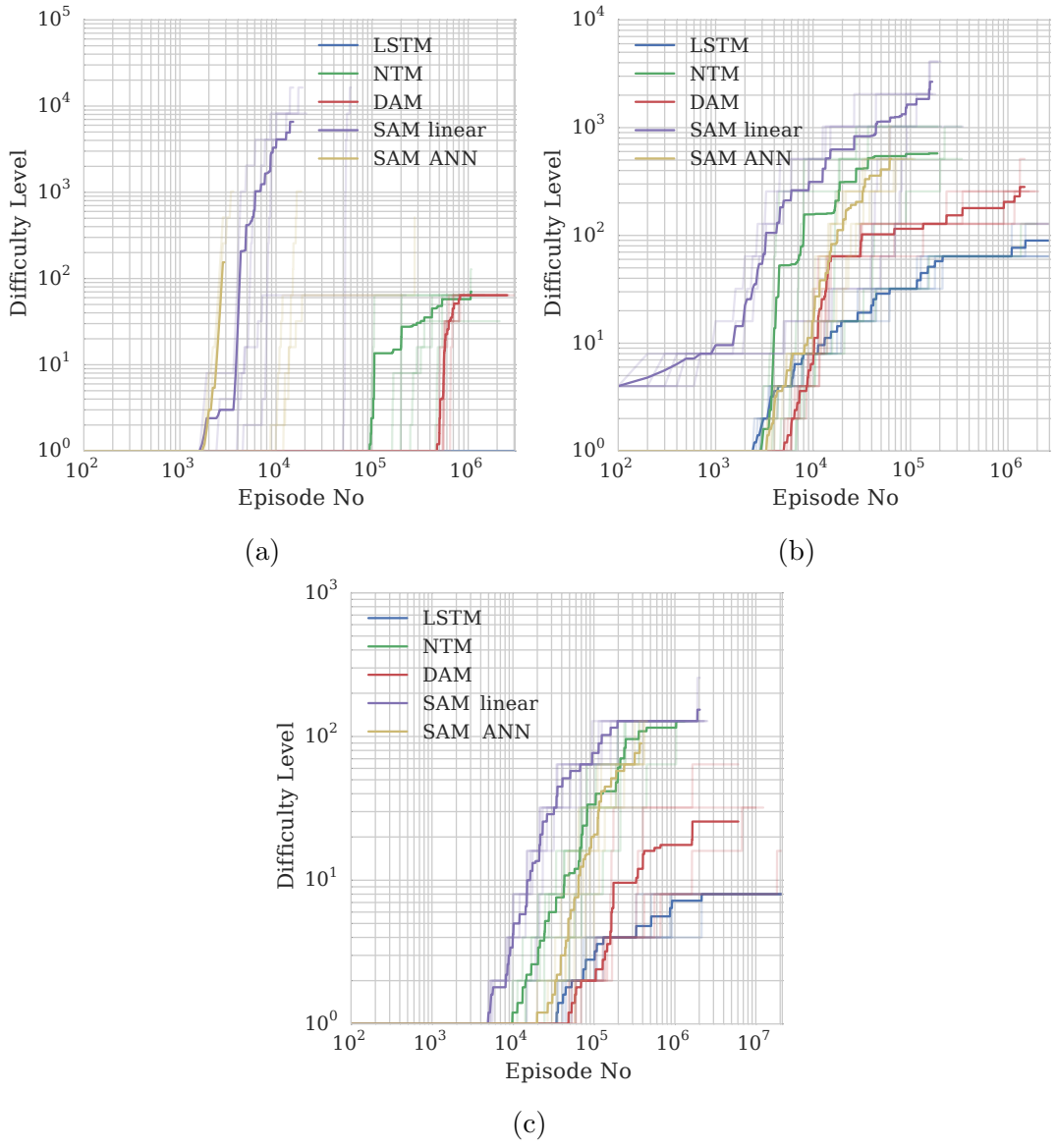


Figure 3.5: Curriculum training curves for sparse and dense models on (a) Associative recall, (b) Copy, and (c) Priority sort. Difficulty level indicates the task difficulty (e.g. the length of sequence for copy). We see SAM train (and backpropagate over) episodes with thousands of steps, and tasks which require thousands of words to be stored to memory. Each model is averaged across 5 replicas of identical hyper-parameters (light lines indicate individual runs).

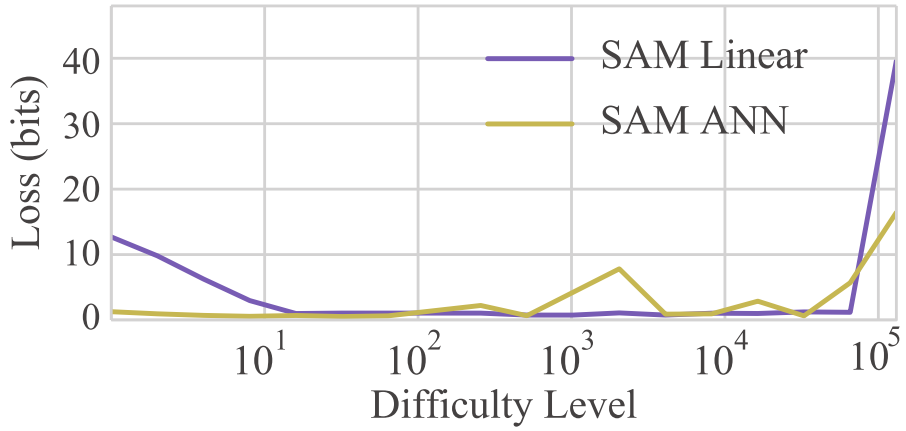


Figure 3.6: We tested the generalisation of SAM on the associative recall task. We train each model up to a difficulty level, which corresponds to the task's sequence length, of $10,000$, and evaluate on longer sequences. The SAM models (with and without ANN) are able to perform much better than chance (48 bits) on sequences of length $200,000$.

of length $10,000$, and found it was then able to generalise to sequences of length 200,000 (Figure 3.6).

3.4.4 Few-Shot Image Classification

We demonstrate that SAM is capable of learning in a non-synthetic dataset via a *few-shot learning* image classification task originally proposed by (Santoro et al., 2016a). We few-shot classify Omniglot (Lake et al., 2015) images; Omniglot is a dataset of 1623 characters taken from 50 different alphabets, with 20 examples of each character. This dataset is used to test rapid, or *one-shot* learning, since there are few examples of each character but many different character classes.

Following Santoro et al. (2016a), we generate episodes where the class labels are permuted, and a subset of characters are randomly selected from the dataset and presented sequentially with their permuted class label. For each episode we sam-

ple 50 character classes and we show ten images from each class, thus the total sequence length is 500. We can track the model's performance throughout the sequence as it observes more and more characters. In order to succeed at the task the model must learn to implicitly classify images by character and rapidly associate a novel character with the correct label, such that it can correctly classify subsequent examples of the same character class. Again, we used an exponential curriculum, doubling the number of additional characters provided to the model whenever the cost was reduced under a threshold. After training all MANNs for the same length of time, a validation task with 500 characters was used to select the best run, and this was then tested on a test set, containing all novel characters for different sequence lengths (Figure 3.7). All of the MANNs were able to perform much better than chance, even on sequences $\approx 4\times$ longer than seen during training. SAM outperformed other models, presumably due to its much larger memory capacity. Previous results on the Omniglot curriculum (Santoro et al., 2016a) task are not identical, since we used 1-hot labels throughout and the training curriculum scaled to longer sequences, but our results with the dense models are comparable (≈ 0.4 errors with 100 characters), while the SAM is significantly better ($0.2 <$ errors with 100 characters).

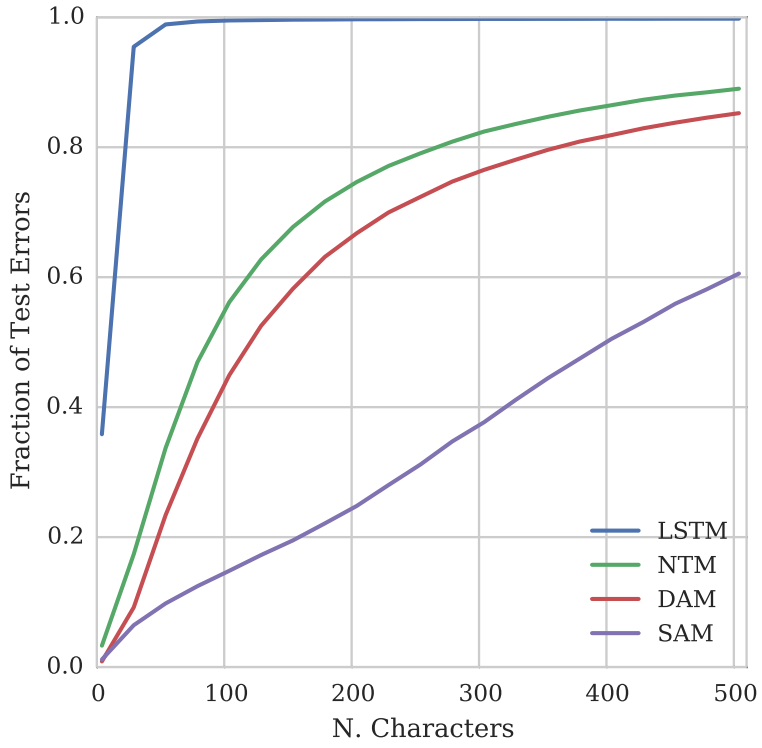


Figure 3.7: Test errors for few-shot image classification on Omniglot, for the best runs (as chosen by the validation set). The characters used in the test set were not used in validation or training. All of the MANNs were able to perform much better than chance with ≈ 500 characters (sequence lengths of ≈ 5000), even though they were trained, at most, on sequences of ≈ 130 (chance is 0.002 for 500 characters). This indicates they are learning solutions that generalise in the sequence length of observations. SAM is able to outperform other approaches, presumably because it can utilise a much larger memory.

3.5 Sparse Differentiable Neural Computer

Graves et al. (2016) proposed a novel MANN the Differentiable Neural Computer (DNC) which extends the Neural Turing Machine in two principal ways. The two innovations proposed by this model are a new approach to tracking memory freeness (dynamic memory allocation) and a mechanism for associating memories together (temporal memory linkage). We demonstrate here that the approaches enumerated in this chapter can be adapted to new models by outlining a sparse version of this model, the Sparse Differentiable Neural Computer (SDNC), which learns with similar data efficiency while retaining the computational advantages of sparsity.

3.5.1 Architecture

For brevity, we will only explain the sparse implementations of these two items, for the full model details refer to the original paper (Graves et al., 2016). The mechanism for sparse memory reads and writes was implemented identically to SAM.

It is possible to implement a scalable version of the dynamic memory allocation system of the DNC avoiding any $O(N)$ operations by using a heap. However, because it is practical to run the SDNC with many more memory words, reusing memory is less crucial so we did not implement this and used the same usage tracking as in SAM.

The temporal memory linkage in the DNC is a system for associating and recalling memory locations which were written in a temporal order, for exampling storing

and retrieving a list. In the DNC this is done by maintaining a temporal linkage matrix $\mathbf{L}_t \in [0,1]^{N \times N}$. $\mathbf{L}_t[i,j]$ represents the degree to which location i was written to after location j. This matrix is updated by tracking the precedence weighting p_t, where $p_t(i)$ represents the degree to which location i was written to.

$$p_0 = 0 \tag{3.8}$$

$$p_t = (1 - \sum_i w_t^W(i))\, p_{t-1} + w_t^W \tag{3.9}$$

The memory linkage is updated according to the following recurrence

$$\mathbf{L}_0 = 0 \tag{3.10}$$

$$\mathbf{L}_t(i,j) = \begin{cases} 0 & i = j \\ (1 - w_t^W(i) - w_t^W(j))\mathbf{L}_{t-1}(i,j) + w_t^W(i)p_{t-1}(j) & i \neq j \end{cases} \tag{3.11}$$

$$\tag{3.12}$$

The temporal linkage $\mathbf{L}_t$ can be used to compute read weights following the temporal links either forward

$$f_t^r = \mathbf{L}_t a_{t-1}^r \tag{3.13}$$

or backward

$$b_t^r = \mathbf{L}_t^T a_{t-1}^r \tag{3.14}$$

The read head then uses a 3-way softmax to select between a content-based read or following the forward or backward weighting.

Naively, the link matrix requires $O(N^2)$ memory and computation although Graves

et al. (2016) proposes a method to reduce the computational cost to $O(N \log N)$ and $O(N)$ memory cost.

In order to maintain the scaling properties of the SAM, we wish to avoid any computational dependence on N. We do this by maintaining two sparse matrices $\mathbf{N}_t, \mathbf{P}_t \in [0,1]^{N \times \mathsf{D_{KL}}}$ that approximate $\mathbf{L}_t$ and $\mathbf{L}_t^T$ respectively. We store these matrices in Compressed Sparse Row format. They are defined by the following updates:

$$\mathbf{N}_0 = 0 \tag{3.15}$$

$$\mathbf{P}_0 = 0 \tag{3.16}$$

$$\mathbf{N}_t(i,j) = (1 - w_t^W(i))\, \mathbf{N}_{t-1}(i,j) + w_t^W(i)\, p_{t-1}(j) \tag{3.17}$$

$$\mathbf{P}_t(i,j) = (1 - w_t^W(j))\, \mathbf{P}_{t-1}(i,j) + w_t^W(j)\, p_{t-1}(i) \tag{3.18}$$

Additionally, p_t is, as with the other weight vectors maintained as a sparse vector with at most K_L non-zero entries. This means that the outer product of $w_t p_{t-1}^T$ has at most K_L^2 non-zero entries. In addition to the updates specified above, we also constrain each row of the matrices $\mathbf{N}_t$ and $\mathbf{P}_t$ to have at most K_L non-zero entries — this constraint can be applied in $O(K_L^2)$ because at most K_L rows change in the matrix.

Once these matrices are applied the read weights following the temporal links can be computed similar to before:

$$f_t^r = \mathbf{N}_t a_{t-1}^r \tag{3.19}$$

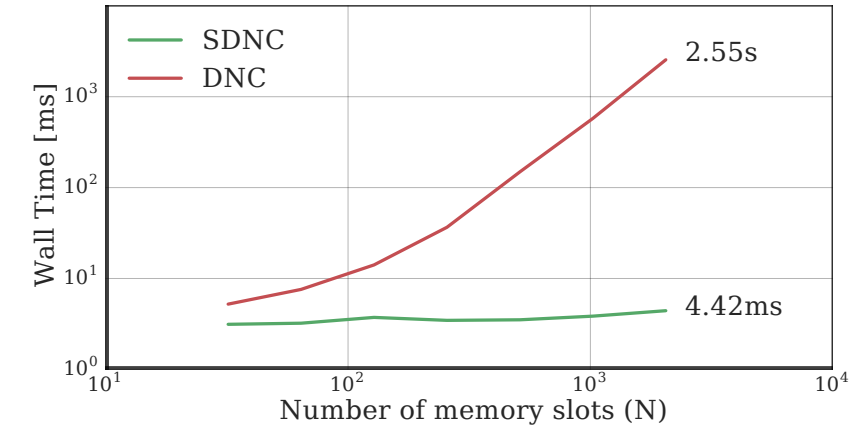
$$b_t^r = \mathbf{P}_t a_{t-1}^r \tag{3.20}$$

Note, the number of locations we read from, K, does not have to equal the number of outward and inward links we preserve, K_L. We typically choose $K_L = 8$ as this is still very fast to compute ($100\mu s$ in total to calculate $\mathbf{N}_t, \mathbf{P}_t, p_t, f_t^r, b_t^r$ on a single CPU thread) and we see no learning benefit with larger K_L. In order to compute the gradients, $\mathbf{N}_t$ and $\mathbf{P}_t$ need to be stored. This could be done by maintaining a sparse record of the updates applied and reversing them, similar to that performed with the memory as described in Section 3.2.6. However, for implementation simplicity we did not pass gradients through the temporal linkage matrices.

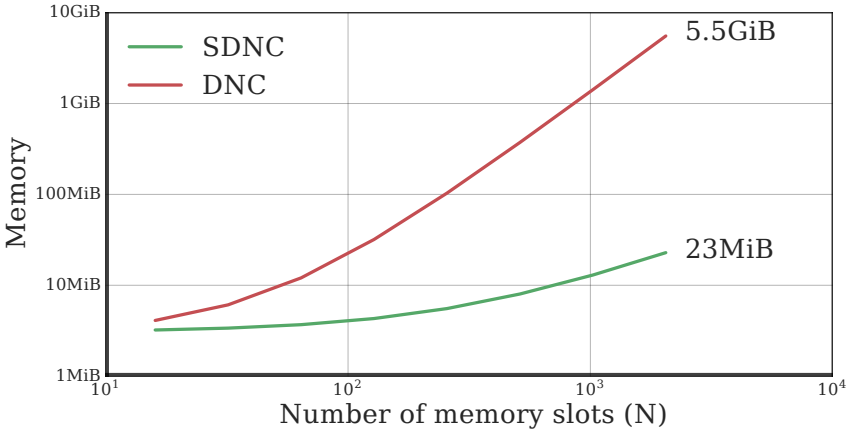
3.5.2 Results

We benchmarked the speed and memory performance of the SDNC versus a naive DNC implementation (details of setup in Appendix A.2). The results are displayed in Figure 3.8. Here, the computational benefits of sparsity are more pronounced due to the expensive (quadratic time and space) temporal transition table operations in the DNC. We were only able to run comparative benchmarks up to $N = 2048$, as the DNC quickly exceeded the machine's physical memory for larger values; however even at this modest memory size we see a speed increase of $\approx 440\times$ and physical memory reduction of $\approx 240\times$. Unlike the SAM memory benchmark in Section 3.4 we plot the total memory consumption, i.e. the memory overhead of the initial start state plus the memory overhead of unrolling the core over a sequence. This is because the SDNC and DNC do not have identical start states. The sparse temporal transition matrices $\mathbf{N}_0, \mathbf{P}_0 \in [0, 1]^{N \times N\{K\}}$ consume much less memory than the corresponding $\mathbf{L}_0 \in [0, 1]^{N \times N}$ in the DNC. In order to compare

the models on an externally-defined benchmark, we ran the DNC and SDNC on the bAbI task. The results are described in Section 3.5.3 and demonstrate the SDNC is capable of learning competitively. In particular, it achieves the best report result on the bAbI task at the time of its release (2016).



(a)



(b)

Figure 3.8: Performance benchmarks between the DNC and SDNC for small to medium memory sizes. Here the SDNC uses a linear KNN. **(a)** Wall-clock time of a single forward and backward pass. **(b)** Total memory usage (including initialisation) when trained over sequence of 10 time steps. When the memory contains 64,000 words the NTM consumes 29 GiB whereas SAM consumes only 7.8 MiB, a compression ratio of 3700.

3.5.3 Question answering on the bAbI tasks

Weston et al. (2015) introduced a set of toy tasks that are considered a prerequisite to agents which can reason and understand natural language as introduced in Section 2.4.4. The task was encoded using straightforward 1-hot word encodings for both the input and output. We trained a single model on all of the tasks, and used the 10,000 examples per task version of the training set (a small subset of which we used as a validation set for selecting the best run and hyperparameters). Previous work has reported best results (Table 3.1), which with only 15 runs is a noisy comparison, so we additionally report the mean and variance for all runs with the best selected hyperparameters (Table 3.2).

The MANNs, except the NTM, are able to learn solutions comparable to the previous best results, failing at only 2 of the tasks. The SDNC manages to solve all but 1 of the tasks, the best reported result on bAbI at the time of writing, when a model is trained jointly on all tasks or even trained singularly on each task in turn. The task that resisted being solved was *Task 16: Basic Induction*. Whilst this task is simple from the outset, we believe it was considered difficult by models because of an implementation bug that sometimes lead to ambiguous or incorrect result. A regular example and a 'buggy' example of this task is shown below:

Task 16: Basic Induction (regular):	**Task 16: Basic Induction (bug)**:
Lily is a swan.	Lily is a swan.
Lily is white.	Lily is white.
Bernhard is green.	Bernhard is green.
Greg is a swan.	Bernhard is a swan.
What color is Greg? A:white	What color is Bernhard? A:white

Essentially the bug was that the distractor statement sampled a person with re-

placement, so people could occasionally be directly associated with an attribute (e.g. colour) but also associated with a class object (e.g. swan) that was associated with a different colour via another person. The natural response to the buggy example above would be green, however the correct answer in the benchmark is white. After discussion with the benchmark creators it was decided to not fix the task, as there still was a uniformly correct strategy to solve the problem, which is to always follow the inductive path, even if the distractor statement is contradictory. Perhaps these models which rely on content-based attention find this almost impossible to discover, especially since they are jointly trained on other tasks such as *Task 15: Basic Deduction* which would encourage the answer to be 'green'.

Notably the best prior results have been obtained by using supervising the memory retrieval (during training the model is provided annotations which indicate which memories should be used to answer a query). More directly comparable previous work with end-to-end memory networks, which did not use supervision (Sukhbaatar et al., 2015), fails at 6 of the tasks. Both the sparse and dense perform comparably at this task, again indicating the sparse approximations do not impair learning. We believe the NTM may perform poorly since it lacks a mechanism which allows it to allocate memory effectively.

Table 3.1: Test results for the best run (chosen by validation set) on the bAbI task. The model was trained and tested jointly on all tasks. All tasks received approximately equal training resources. Both SAM and DAM pass all but 2 of the tasks, without any supervision of their memory accesses. SDNC achieves the best reported result on this task with unsupervised memory access, solving all but 1 task. We've included comparison with memory networks, both with supervision of memories (MemNet S) and, more directly comparable with our approach, learning end-to-end (MemNets U).

	LSTM	DNC	SDNC	DAM	SAM	NTM	MN S	MN U
1: 1 supporting fact	28.8	0.0	0.0	0.0	0.0	16.4	0.0	0.0
2: 2 supporting facts	57.3	3.2	0.6	0.2	0.2	56.3	0.0	1.0
3: 3 supporting facts	53.7	9.5	0.7	1.3	0.5	49.0	0.0	6.8
4: 2 argument relations	0.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0
5: 3 argument relations	3.5	1.7	0.3	0.4	0.7	2.5	0.3	6.1
6: yes/no questions	17.6	0.0	0.0	0.0	0.0	9.6	0.0	0.1
7: counting	18.5	5.3	0.2	0.4	1.9	12.0	3.3	6.6
8: lists/sets	20.9	2.0	0.2	0.0	0.4	6.5	1.0	2.7
9: simple negation	18.2	0.1	0.0	0.0	0.1	7.0	0.0	0.0
10: indefinite knowledge	34.0	0.6	0.2	0.0	0.2	7.6	0.0	0.5
11: basic coreference	9.0	0.0	0.0	0.0	0.0	2.5	0.0	0.0
12: conjunction	5.5	0.1	0.1	0.0	0.1	4.6	0.0	0.1
13: compound coreference	6.3	0.4	0.1	0.0	0.0	2.0	0.0	0.0
14: time reasoning	56.1	0.2	0.1	3.8	4.3	44.2	0.0	0.0
15: basic deduction	49.3	0.1	0.0	0.0	0.0	25.4	0.0	0.2
16: basic induction	53.2	51.9	54.1	52.8	53.1	52.2	0.0	0.2
17: positional reasoning	41.7	21.7	0.3	6.0	16.0	39.7	0.0	41.8
18: size reasoning	8.4	1.8	0.1	0.3	1.1	3.6	24.6	8.0
19: path finding	76.4	4.3	1.2	1.5	2.6	5.8	2.1	75.7
20: agent's motivations	1.9	0.1	0.0	0.1	0.0	2.2	31.9	0.0
Mean Error (%)	28.0	5.2	2.9	3.3	4.1	17.5	3.2	7.5
Failed tasks (err. $> 5\%$)	17	4	1	2	2	13	2	6

Table 3.2: Mean and variance of test errors for the best set of hyperparameters (chosen according the validation set). Statistics are generated from 15 runs.

	LSTM	DNC	SDNC	DAM	SAM	NTM
1: 1 supporting fact	30.9 ± 1.5	2.2 ± 5.6	0.0 ± 0.0	2.9 ± 10.7	4.7 ± 12.8	31.5 ± 15.3
2: 2 supporting facts	57.4 ± 1.2	23.9 ± 21.0	7.1 ± 14.6	12.1 ± 19.3	30.9 ± 25.1	57.0 ± 1.3
3: 3 supporting facts	53.0 ± 1.4	29.7 ± 15.8	9.4 ± 16.7	15.3 ± 17.4	31.4 ± 21.6	49.4 ± 1.3
4: 2 argument relations	0.7 ± 0.4	0.1 ± 0.1	0.1 ± 0.1	0.1 ± 0.1	0.2 ± 0.2	0.4 ± 0.3
5: 3 argument relations	4.9 ± 0.9	1.3 ± 0.3	0.9 ± 0.3	1.0 ± 0.4	1.0 ± 0.5	2.7 ± 1.2
6: yes/no questions	18.8 ± 1.0	2.8 ± 5.0	0.1 ± 0.2	1.9 ± 5.3	3.9 ± 6.7	18.6 ± 2.7
7: counting	18.2 ± 1.1	7.3 ± 5.9	1.6 ± 0.9	4.5 ± 6.1	7.3 ± 6.6	18.7 ± 3.2
8: lists/sets	20.9 ± 1.4	4.0 ± 4.1	0.5 ± 0.4	2.7 ± 5.4	3.6 ± 6.2	18.5 ± 5.9
9: simple negation	19.4 ± 1.5	3.0 ± 5.2	0.0 ± 0.1	2.1 ± 5.5	3.8 ± 6.7	17.6 ± 3.4
10: indefinite knowledge	33.0 ± 1.6	3.2 ± 5.9	0.3 ± 0.2	3.4 ± 8.1	5.7 ± 9.2	25.6 ± 6.9
11: basic coreference	15.9 ± 3.3	0.9 ± 3.0	0.0 ± 0.0	1.5 ± 5.5	2.6 ± 7.9	15.2 ± 9.4
12: conjunction	7.0 ± 1.3	1.5 ± 1.6	0.2 ± 0.3	1.8 ± 6.4	2.9 ± 7.9	14.7 ± 8.9
13: compound coreference	9.1 ± 1.4	1.5 ± 2.5	0.1 ± 0.1	0.6 ± 2.2	1.3 ± 2.4	6.8 ± 3.3
14: time reasoning	57.0 ± 1.6	10.6 ± 9.4	5.6 ± 2.9	11.5 ± 15.0	15.0 ± 12.6	52.6 ± 5.1
15: basic deduction	48.1 ± 1.3	31.3 ± 15.6	3.6 ± 10.3	17.2 ± 19.7	5.5 ± 13.8	42.0 ± 6.9
16: basic induction	53.8 ± 1.4	54.0 ± 1.9	53.0 ± 1.3	53.8 ± 1.0	53.6 ± 1.2	53.8 ± 2.1
17: positional reasoning	40.8 ± 1.8	27.7 ± 9.4	12.4 ± 5.9	16.9 ± 10.3	20.4 ± 8.6	40.1 ± 1.3
18: size reasoning	7.3 ± 1.9	3.5 ± 1.5	1.6 ± 1.1	1.8 ± 1.7	3.0 ± 1.8	5.0 ± 1.2
19: path finding	74.4 ± 1.3	44.9 ± 29.0	30.8 ± 24.2	23.0 ± 25.4	33.7 ± 27.8	60.8 ± 24.6
20: agent's motivations	1.7 ± 0.4	0.1 ± 0.2	0.0 ± 0.0	0.1 ± 0.5	0.0 ± 0.0	2.0 ± 0.3
Mean Error (%)	28.7 ± 0.5	12.8 ± 4.7	6.4 ± 2.5	8.7 ± 6.4	11.5 ± 5.9	26.6 ± 3.7
Failed tasks (err. $> 5\%$	17.1 ± 0.8	8.2 ± 2.5	4.1 ± 1.6	5.4 ± 3.4	7.1 ± 3.4	15.5 ± 1.7

3.6 Conclusion

We demonstrate that you can train neural networks with large memories via a sparse read and write scheme that makes use of efficient data structures within the network, and obtain significant speedups during training. Although we have focused on a specific MANN (SAM), which is closely related to the NTM, the approach taken here is general and can be applied to many differentiable memory architectures, such as Memory Networks (Weston et al., 2014).

It should be noted that there are multiple possible routes toward scalable memory architectures. For example, prior work aimed at scaling Neural Turing Machines (Zaremba and Sutskever, 2015) used reinforcement learning to train a discrete addressing policy. This approach also touches only a sparse set of memories at each time step, but relies on higher variance estimates of the gradient during optimisation. Though we can only guess at what class of memory models will become staple in machine learning systems of the future, we argue in Section 3.3 that they will be no more efficient than SAM in space and time complexity if they address memories based on content.

We have experimented with randomised k-d trees and LSH within the network to reduce the forward pass of training to sub-linear time, but there may be room for improvement here. K-d trees were not designed specifically for fully online scenarios, and can become imbalanced during training. Recent work in tree ensemble models, such as Mondrian forests (Lakshminarayanan et al., 2014), show promising results in maintaining balanced hierarchical set coverage in the online setting. An alternative approach which may be well-suited is LSH forests (Bawa et al., 2005), which adaptively modifies the number of hashes used. It would be an

interesting empirical investigation to more fully assess different ANN approaches in the challenging context of training a neural network.

One limitation of this study is that the experiments were run on CPU machines due to a lack of availability, at time of writing, to more powerful accelerators such as GPUs and TPUs. We see the asymptotic analysis translate to meaningful wall-clock speedups for our CPU benchmarks, where we see an $\mathcal{O}(\log N)$ time sparse operation quickly outperform brute force search. However for accelerated hardware specialised to high throughput matrix multiplications, the constant factors in these asymptotic statements will favour dense operations. It will be an interesting challenge to realise these same speed gains with sparsity structures that map well to GPUs. Some progress has been made in this area following this study with libraries such as FAISS (Johnson et al., 2017) which incorporate k-means LSH nearest neighbour search and contains implementations for various accelerators.

One key component to scaling towards very large sequences and memory sizes has been the use of a curriculum. It is unlikely one could learn these various algorithmic tasks such as copying and associative recall in the 99.9% sparse setting if one started with long sequences, because the model would frequently not attend to the right input. By starting small and then scaling up, the model frequently attends to the right inputs and roughly learns the algorithm. Then in the course of scaling up, the model already knows the types of inputs to attend to, and we see empirically the sparsity helps reduce noise. In this chapter the curriculum essentially helps the model learn what to attend to, but this is not guaranteed in real world tasks.

Humans are able to retain a large, task-dependent set of memories obtained in one pass with a surprising amount of fidelity (Brady et al., 2008). Here we have demonstrated architectures that may one day compete with humans at these kinds of tasks; progressing this on to more challenging real world problems is the clear future direction.

Chapter 4

Compressive Memory for Identifying Rare Classes

Neural networks trained with backpropagation often struggle to identify classes that have been observed a small number of times. In applications where most class labels are rare, such as language modelling, this can become a performance bottleneck. One potential remedy is to augment the network with a fast-learning non-parametric model which stores recent activations and class labels into an external memory. We explore a simplified architecture where we treat a subset of the model parameters as fast memory stores. This can help retain information over longer time intervals than a traditional memory, and does not require additional space or compute. In the case of image classification, we display faster binding of novel classes on an Omniglot image curriculum task. We also show improved performance for word-based language models on news reports (GigaWord), books (Project Gutenberg) and Wikipedia articles (WikiText-103) — the latter achieving

a state-of-the-art perplexity of 29.2.

4.1 Motivation

Neural networks can be trained to classify discrete outputs by appending a softmax output layer. This is a linear map projecting the d-dimensional hidden output of the network to m outputs, where m is the number of distinct classes. A softmax operator (Bridle, 1990) is then applied to produce a probability distribution over classes. The parameters in this softmax layer are typically optimised with the network's parameters by gradient descent.

We can think of the weights in the softmax layer $\theta \in \mathbb{R}^{m \times d}$ as a set of m vectors $\theta[i];\ i = 1, \ldots, m$ that each correspond to a given class. When trained with a supervised loss, such as cross-entropy, each step of gradient descent pulls the parameter $\theta[y]$, corresponding to the class label y, towards having a greater inner product with the network output h, and pushes all other parameters $\theta[j]$, $j \neq y$ towards having a smaller inner product with h.

One shortcoming of neural network classifiers trained with backpropagation is that they require many input examples for a given class in order to predict it with reasonable accuracy. That is, many positive class examples and optimisation steps are required to pull $\theta[i]$ towards a point in space where class i can then be recognised. While the learner will have many opportunities to organise $\theta[i]$ parameters associated with frequent classes, infrequent class parameters will be poorly estimated. In domains where new classes are frequently introduced, or large-scale classification problems where some classes are very infrequently observed, this estimation

problem is potentially quite serious.

One approach to speed up learning, which has received revived interest, is meta-learning. Here, meta-learning refers to algorithms which learn to produce or manipulate learning algorithms (Thrun, 1998; Hochreiter et al., 2001), and it operates by learning over a distribution of tasks or datasets. A meta-learner applies knowledge from the global distribution of tasks to produce or optimise algorithms which specialise to a given task instance. Meta-learning of neural networks has seen promising results for applications such as parameter optimisation (Andrychowicz et al., 2016; Ravi and Larochelle, 2016; Finn et al., 2017) and classification (Santoro et al., 2016a; Vinyals et al., 2016; Zhou et al., 2018a). For classification, the networks are augmented with a differentiable external memory, and are trained with many rounds of data — with class labels permuted between episodes.

Meta-learning can be very powerful for few-shot learning in cases where there is a set of similar prior data to meta-learn over, however it may not be practical for standalone datasets. For example, if one wants to model the grammar of computer code, it is unclear that a meta-learning system trained over natural language will be useful. Also memory-based meta-learning requires backpropagating from the read time to the original write time, which is not well suited to applications where writes and reads are separated by many time steps. In the case of modelling language, for example, infrequent words will not occur for large time intervals — rendering memory-based meta-learning challenging.

The task of statistical language modelling itself is interesting to investigate issues with binding new or infrequent classes, because most classes (words) are infrequent (Zipf, 1935) and new classes naturally emerge over time. Recent approaches

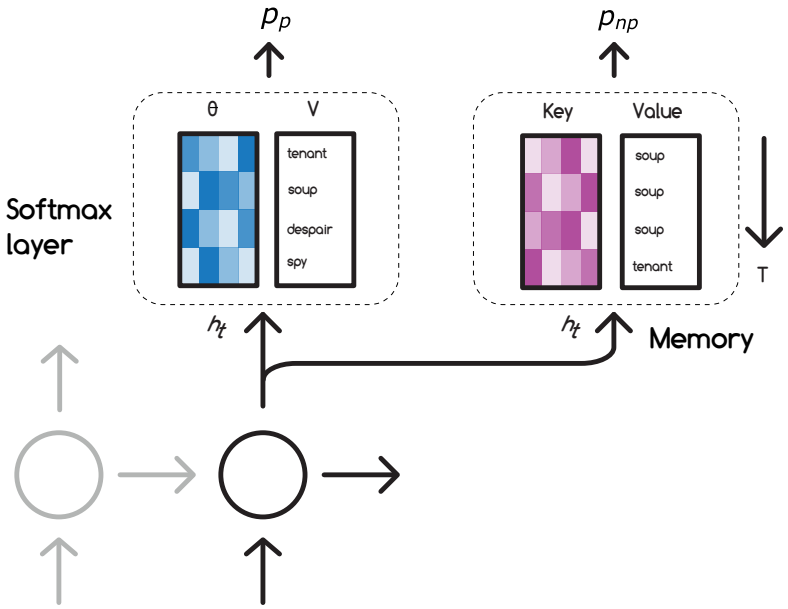


Figure 4.1: Mixture model of parametric and non-parametric classifiers connected to a recurrent language model. The non-parametric model (right hand side) stores a history of past activations and associated labels as key, value pairs. The parametric model (left hand side) contains learnable parameters θ for each class in the output vocabulary V. We can view both components as key, value memories — one slow-moving, optimised with gradient descent, and one rapidly updating but ephemeral.

to improve neural language models have involved augmenting the network with a non-parametric cache, which stores past hidden activations $h_{t-n}, \ldots, h_{t-1}$ and corresponding labels, $y_{t-n}, \ldots, y_{t-1}$ (Vinyals et al., 2015; Merity et al., 2016; Grave et al., 2016b; Kawakami et al., 2017; Grave et al., 2017). Attention over this cache provides better modelling of infrequent words that occur in a recent context, including previously unknown words (Gulcehre et al., 2016). However there is a diminishing return to increasing the cache size (Grave et al., 2016b), and once rare words fall outside the recent context the boost in predictive performance expires.

Motivated from these memory systems, we explore a very simple optimisation procedure where the network accumulates activations h_t directly into the softmax layer weights $\theta[y_t]$ when a class y_t has been seen a small number of times, and uses gradient descent otherwise. Accumulating or smoothing network activations into the weights actually corresponds to the well-known Hebbian learning update rule $W[i,j] \leftarrow \frac{1}{n}\sum_{t=1}^{n} x_t^i x_t^j$ (Hebb, 1949) in the special case of classification on the output layer, where W, x_t^i, x_t^j correspond to θ, h_t, y_t respectively. We see that mixing the two rules provides better initial representations and can also preserve these representations for much longer time spans. This is because memorised activations for one class are not competing for space with activations from other (more frequent, say) classes — unlike a conventional external memory. In this sense, the parameters become an instance of a quickly updated compressed memory, we explore this idea in Section 4.2.3

We demonstrate this model adapts quickly to novel classes in a simple image classification task using handwritten characters from Omniglot (Lake et al., 2015). We then show it improves overall test perplexity for two medium-scale language modelling corpora, WikiText103 (Wikipedia articles) from Merity et al. (2016) and Project Gutenberg[1] (books), alongside a large-scale corpus GigaWord v5 (news articles) from Parker et al. (2011). By splitting accuracy over word frequency buckets, we see improved perplexity for less frequent words.

[1]Project Gutenberg. (n.d.). Retrieved January 2, 2018, from www.gutenberg.org

4.2 Model

We propose the Hebbian Softmax , a modification of the traditional softmax layer with an updated learning rule. The Hebbian Softmax contains the same linear map from the hidden state to the output vocabulary, but learns by smoothing hidden activations into the weight parameters for novel classes whilst concurrently applying gradient descent. This is to facilitate faster binding of novel classes, and improve learning of infrequent classes. We note this corresponds to a learning rule that transitions from Hebbian learning to gradient descent, and we will show that the combination of the two learning rules works better than either one in isolation.

Many of the features of the Hebbian Softmax are motivated from memory systems, and the theory of *Complementary Learning Systems* in the brain McClelland et al. (1995). During training, the weights corresponding to a given class will initially correspond to a compressed[2] episodic memory store — with new activations memorised and older activations eventually forgotten.

The parameters of the softmax layer are treated both as regular slow-adapting network parameters through which gradients flow to the rest of the network, and fast-adapting memory slots which are updated sparsely without altering the rest of the network. In comparison to an external memory, the advantage of Hebbian Softmax is that it is simple to implement and requires almost no additional space or computation.

We will describe the learning rule in detail, and contrast the conditional proba-

[2]The memory is denoted 'compressed' because multiple activations corresponding to the same class are smoothed into one vector, instead of being stored separately.

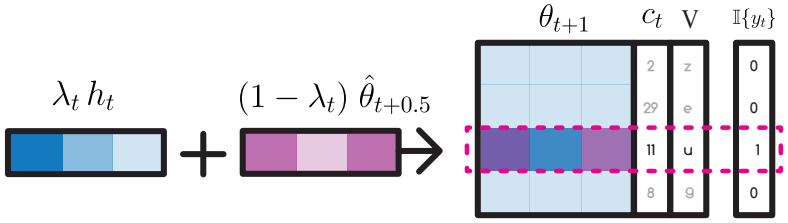


Figure 4.2: Update rule. Here the vector $\hat{\theta}_{t+0.5}$ denotes the parameters $\theta_t[y_t]$ of the final layer softmax corresponding to the active class y_t after one step of gradient descent. This is interpolated with the hidden activation at the time of class occurrence, h_t. The remaining parameters are optimised with gradient descent. Here, $\mathbb{I}\{y_t\}$ is the one-hot target vector, V denotes the vocabulary of classes, and c_t is defined to be a counter of class occurrences during training — which is used to anneal λ_t as described in (4.3).

bilities from Hebbian Softmax to those generated by a non-parametric cache. We also generalise the memorisation procedure in Section 4.2.4 as an instance of a secondary fast-learning overfitting procedure with respect to a Euclidean objective, and explore several promising variant objective functions.

4.2.1 Update Rule

Given the weights of a linear projection $\theta \in \mathbb{R}^{d \times m}$ in the final softmax layer of a network, we calculate the gradient descent update with respect to a cross-entropy loss,

$$\hat{\theta}_{t+0.5}[i] \leftarrow \begin{cases} \theta_t[i] - \alpha\,(p_i - 1)\,h_t & i = y_t \\ \theta_t[i] - \alpha\, p_i\, h_t & i \neq y_t \end{cases} \tag{4.1}$$

where $p_i = e^{h_t^T \theta_i} / \sum_{j=1}^{n} e^{h_t^T \theta_j}$ is the probability output from the softmax, and α is the learning rate. In practice the gradient descent update $\hat{\theta}_{t+0.5}$ can be calculated

with adaptive optimisers, such as RMSProp (Tieleman and Hinton, 2012). This is interpolated with the previous layer's hidden activation h_t for the active class y_t,

$$\theta_{t+1}[i] \leftarrow \begin{cases} \lambda_t \, h_t + (1 - \lambda_t) \, \hat{\theta}_{t+0.5}[i] & i = y_t \\ \hat{\theta}_{t+0.5}[i] & i \neq y_t \ , \end{cases} \tag{4.2}$$

as illustrated in Figure 4.2. When $\lambda_t = 1$ this corresponds to the rule $\theta_{t+1} \leftarrow h_t \cdot \mathbb{I}\{y_t\}$ where $\mathbb{I}\{y_t\} \in [0, 1]^m$ is a one-hot target vector. In this case Hebbian update rule, $W_{ij} \leftarrow x_i x_j$ for $x_i = h_t$ the hidden output and $x_j = \mathbb{I}\{y_t\}$ the target. Naturally when $\lambda = 0$ this is gradient descent, and so we see Hebbian Softmax is mixture of the two learning rules. All remaining parameters in the model are optimised with gradient descent as usual.

When mixing the two learning rules, we benefit from fast initial learning of classes that have not been seen many times, along with the eventual consolidation of frequently seen classes. As such we do not want λ_t to be constant, but instead something that is eventually annealed to zero. We add an additional counter array $c \in \mathbb{Z}^m$ which counts class occurrences, and propose an annealing function of

$$\lambda_t = \max(1 \, / \, c[y_t], \ \gamma) \ \cdot \ \mathbb{I}\{c[y_t] < T\} \tag{4.3}$$

where γ, T are tuning parameters. T is the number of class occurrences before switching completely to gradient descent and γ is the minimum activation mixing parameter. Although heuristic, we found this worked well in practice vs. a constant λ or pure annealing $\lambda_t = 1/c[y_t]$. If training from scratch, we suggest setting

Algorithm 1 Hebbian Softmax batched update

— At iteration 0
$\gamma \leftarrow$ min. discount (hyper-parameter)
$T \leftarrow$ smoothing limit (hyper-parameter)
$M \leftarrow$ num. classes
$B \leftarrow$ batch size
$\mathbf{c}_0[i] \leftarrow 0; \quad i = 1, \ldots, M$
— At iteration t
$\mathbf{h}_{t,1:B} \leftarrow$ softmax inputs
$\mathbf{p}_{t,1:B} \leftarrow$ softmax outputs
$\mathbf{y}_{t,1:B} \leftarrow$ target labels
$\hat{\theta}_{t+0.5} \leftarrow \texttt{SGD}(\theta_t, \mathbf{h}_{t,1:B}, \mathbf{p}_{t,1:B}, \mathbf{y}_{1:B})$
for $i = 1, \ldots, M$ **do**
  $n_{t,i} \leftarrow \sum_{j=1}^{B} \mathbb{I}\{y_{t,j} = i\}$
  if $n_{t,i} > 0$ **then**
    $\lambda_{t,i} \leftarrow \max(1/\mathbf{c}_t[i], \gamma)\, \mathbb{I}\{\mathbf{c}_t[i] < T\}$
    $\bar{h}_{t,i} \leftarrow \frac{1}{n_{t,i}} \sum_{j=1}^{B} h_{t,j} \mathbb{I}\{y_{t,j} = i\}$
    $\theta_{t+1} \leftarrow \lambda_{t,i} \bar{h}_{t,i} + (1 - \lambda_{t,i}) \hat{\theta}_{t+0.5}[i]$
  else
    $\theta_{t+1} \leftarrow \hat{\theta}_{t+0.5}[i]$
  end if
  $\mathbf{c}_{t+1}[i] \leftarrow \mathbf{c}_t[i] + n_{t,i}$
end for

$\gamma = 1/N_{min}$ and $T = N_{min} \times$ (# epochs until convergence) where N_{min} is the minimum number of occurrences of any class in a training epoch. This is to ensure we smooth over many class examples in a given epoch, and the memorisation of activations continues until the representation of h_t stabilises. We describe the full algorithm in Algorithm 1, including details for training with minibatches.

The final layer trains with a two-speed dynamic. For some training steps the full network will be optimised slowly via gradient descent as usual (when frequently-encountered classes are observed), and for other time steps a sparse subset of parameters will rapidly change. The remaining network parameters are optimised

with gradient descent.

It is worth noting that simply increasing the learning rate of the softmax layer, or running multiple steps of optimisation on rare class inputs, would not achieve the same effect. The value $\theta[y_t]$ would indeed be pulled towards a large inner product with h_t, however neighbouring parameters $\theta[i];\ i \neq y_t$ would be pushed towards a large negative inner product with h_t and this could lead to catastrophic forgetting of previously consolidated classes. Instead we allow gradient descent to slowly push neighbouring parameters away, and thus disambiguate similar classes in a gradual fashion.

4.2.2 Relation to weight decay

We can combine the updates (4.1) and (4.2) for the parameter $\theta[y_t]$ relating to the observed class:

$$\theta_{t+1}[y_t] = \lambda_t h_t + (1-\lambda_t)\hat{\theta}_{t+0.5} \tag{4.4}$$

$$= \lambda_t h_t + (1-\lambda_t)(\theta_t[y_t] - \alpha(p_i - 1)h_t) \tag{4.5}$$

$$= \theta_t[y_t] + h_t\left(\lambda_t + (1-\lambda_t)\alpha(1-p_i)\right) - \lambda_t\theta_t[y_t]\ . \tag{4.6}$$

This takes the form of stochastic gradient descent with weight decay, with a different step size towards h_t. Instead of the $\alpha(1-p_i)$ multiplier on h_t we have $\lambda_t + (1-\lambda_t)\alpha(1-p_i)$. For a learning rate less than one ($0 \leq \alpha \leq 1$), $\lambda_t + (1-\lambda_t)\alpha(1-p_i) \geq \alpha(1-p_i)$ so can consider this step size to be at least as large as s.g.d for all values of $\lambda_t \in [0,1]$. Thus for $0 < \lambda_t \leq 1$ we have an update that resembles a faster-learning s.g.d. with weight decay, however for $\lambda_t = 0$ we reduce to exactly

s.g.d. We comment next on how we can interpret the update as a cache memory store.

4.2.3 Relation to cache models

We can consider the weights constructed from the above optimisation procedure as a compressed memory, storing historic activations. We contrast the output probabilities of Hebbian Softmax with those produced from a non-parametric cache model.

Recall the conditional probability of a class, w, given a cache of previous activations (2.7). If we set $I_w(j)$ to be the time step of j-th most recent occurrence of w, then we can re-write the cache probability,

$$\begin{aligned} p_c(w \mid h_t) &\propto \sum_{i=t-n}^{t-1} e^{h_t^T h_i} \mathbb{I}\{y_i = w\} \\ &= \sum_{j=1}^{N_w} e^{g(j)\, h_t^T h_{I_w(j)}} \end{aligned} \tag{4.7}$$

where $g(j) = -\infty$ if $j < t-n$ and 1 otherwise, is a weighting function which places uniform weight to the attention over classes in the past n time steps. However if we wish to characterise infrequent classes, we may want a weighting scheme with a larger time horizon that has a smooth decay.

If we modified the cache to have infinite memory capacity and used a geometric weighting scheme to decay the contribution of the j-th most recent activation corresponding to the given class, e.g. $g(j) = \lambda\,(1-\lambda)^{j-1}$, then the resulting

conditional probability is,

$$\tilde{p}_c(w \mid h_t) \propto \sum_{j=1}^{N_w} e^{\lambda\,(1-\lambda)^{j-1}\,h_t^T h_{I_w(j)}} \tag{4.8}$$

where N_w is the total number of occurrences of class w. Let us now consider the conditional probability from Hebbian Softmax for class w, where w has been observed less than T times. If the gradient with respect to θ_i is zero and we fix $\lambda_t = \lambda$ over time, then (4.2) gives

$$\theta_i \approx \sum_{j=1}^{N_w} \lambda\,(1-\lambda)^{j-1} h_{I_w(j)}\ ,$$

plugging this into our softmax conditional probability,

$$\begin{aligned} p_\theta(w \mid h_t) \propto e^{h_t^T \theta_w} &\approx e^{h_t^T \sum_{j=1}^{N_w} \lambda\,(1-\lambda)^{j-1} h_{I_w(j)}} \\ &= \prod_{j=1}^{N_w} e^{\lambda\,(1-\lambda)^{j-1} h_t^T h_{I_w(j)}}\ . \end{aligned}$$

we see the parametric Hebbian Softmax actually becomes a proxy for the conditional probability output by the non-parametric infinite cache model $\tilde{p}_c$. Past activations now have a geometric contribution to the probability, versus the cache's arithmetic reduction (4.8). This form is useful because we can compute p_{sm} much more efficiently than $\tilde{p}_c$ and it does not require storing the entire history of past activations.

4.2.4 Alternate Objective Functions

We briefly discuss a generalisation of the Hebbian Softmax update by casting it as an overfitting procedure to an inner objective function. Recall equation (4.2) for parameters corresponding to the active class,

$$\theta_{t+1}[i] \leftarrow \lambda_t \, h_t + (1 - \lambda_t) \, \hat{\theta}_{t+0.5}[i].$$

We can re-phrase this as smoothing $\hat{\theta}_{t+0.5}[i]$ with the trivial solution to a Euclidean objective function, which we overfit to.

$$\theta_{t+1}[i] \leftarrow \lambda w^* + (1 - \lambda) \, \hat{\theta}_{t+0.5}[i]$$
$$w^* \leftarrow \arg\max_w -||w - h_t||_2$$

From this perspective we are performing a two-level optimisation procedure. The outer optimisation loop is the mixture of gradient descent and exponential smoothing, and the inner optimisation loop determines a good value for w^* based on the activation h_t and the current parameters.

We consider several other objective functions that are more expensive to compute, but may be preferable to a simple Euclidean distance. Notably, switching to inner product similarity (`IP`), and also incorporating a cost to parameter similarity (`SVM`, `Smax`) to push w^* towards h_t but away from neighbouring parameters — to avoid confusion or interference with other classes. Whilst the optimizer update in (4.1) would also push the parameter away from neighbouring classes, in the setting where we are purely relying on the inner objective loop, i.e. when $\lambda_t = 1$ then this does not have an effect. As we keep neighbouring parameters fixed, we hope to

avoid the catastrophic forgetting typically associated with model overfitting. We list the set of objectives considered,

$$w^* \leftarrow \arg\max_w \; g(w)$$

$$g_{\text{L2}}(w) = -||w - h_t||_2 \tag{4.9}$$

$$g_{\text{IP}}(w) = w^T h_t \tag{4.10}$$

$$g_{\text{SVM}}(w) = w^T h_t - \sum_{\theta_j \in \mathcal{N}_k(h_t)} \xi \, w^T \theta_j \cdot \mathbb{I}(w^T \theta_j > \epsilon) \tag{4.11}$$

$$g_{\text{Smax}}(w) = e^{w^T h_t} / \sum_{\theta_j \in \mathcal{N}_k(h_t)} e^{w^T \theta_j} \tag{4.12}$$

where $\mathcal{N}_k(h_t)$ refers to the k nearest parameters to the activation h_t that do not correspond to y_t, the class label. Including all M parameters in θ_t would make the inner optimisation loop very slow, so we choose a sparse subset $k \ll M$. These are all optimised under the hard norm constraint $||w||_2 < 10$ with gradient descent for multiple steps, typically 20, at a given point in training.

4.3 Results

4.3.1 Image Curriculum

We apply Hebbian Softmax to the problem of image classification. We create a simple curriculum task using Omniglot data (Lake et al., 2015), where a subset of classes (30) are initially provided, and 5 new classes are added when test performance exceeds a threshold (60%). Although this is a toy setup, it allows us to investigate the basic properties of fast class binding without other confounding

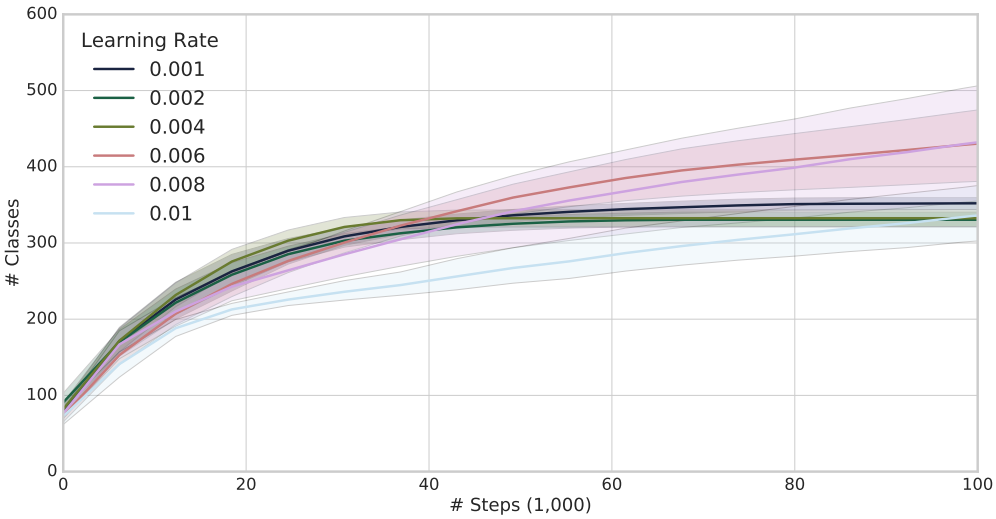


Figure 4.3: Learning rate comparison. Omniglot curriculum performance versus learning rate for a regular softmax architecture using RMSProp. Values of 0.006 to 0.008 are similarly fast to learn and are stable. Stability breaks down for larger values.

factors, found in real-world problems.

Omniglot contains handwritten characters from 50 alphabets, totalling 1623 unique character classes. There are 20 examples per class. We partition the first 5 examples per class to a test set, and assign the rest for training. We use the same architectural setup as *Matching Networks* (Vinyals et al., 2016) where the images are re-sized to 28×28 and a 4 layer convolutional neural network is used. Each layer has 64 filters, 3×3 convolutions, batch normalisation, ReLU activations, and 2×2 max pooling. Each channel maps the input to a scalar, so the resulting hidden size is 64. All weight parameter in the softmax are initialised with Glorot initialisation (Glorot and Bengio, 2010). Models were trained with 20% dropout on the final layer and a small amount of data augmentation was applied to training examples (rotation $\in [-30, 30]$, translation) to avoid over-fitting. Otherwise the models quickly plateau on a low level. For the Hebbian Softmax update, we

store the pristine hidden activation pre-dropout. Unlike many one-shot Omniglot studies, we do not train in a meta-learning setup — namely, labels are not shuffled between episodes. We do this to emulate the truly low-data regime, where we cannot pre-train on many similar instances of the task. However it could be an interesting extension to evaluate the Hebbian Softmax in the meta-learning setting where one would back-propagate through both the hidden activations h_t and the learning update.

We trained the convnet classifier with RMSProp (Tieleman and Hinton, 2012), Adam (Kingma and Ba, 2014), and AdaGrad (Duchi et al., 2011). We swept over learning rates to find the fastest-learning baseline softmax model, as displayed in Figure 4.3. We then compared the regular softmax layer with the Hebbian Softmax , both placed on top of the convnet encoder. It is worth notin that when we use optimisers with trailing statistics such as RMSProp, Adam and AdaGrad we maintain gradient statistics that do not correspond to the originally-defined update, which could potentially create instability. If we inspect the number of steps spent on each level averaged over 10 seeds, focusing on RMSProp for simplicity, we see in Figure 4.4 that the model is noticeably more data efficient after 80 total classes. In Figure 4.5 we see this faster curriculum progression is consistent across RMSProp, Adam, and AdaGrad. Although the models are far from one-shot, there is a $1 - 2X$ data efficiency gain on average.

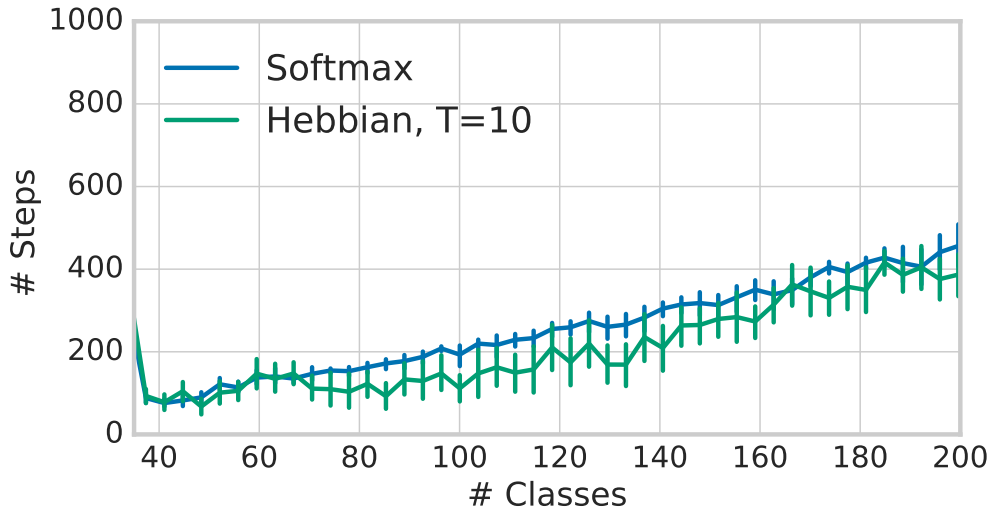


Figure 4.4: Number of training steps taken to complete each level on the Omniglot curriculum task. Comparisons between the Hebbian Softmax and softmax baseline are averaged over 10 independent seeds. As classes are sampled uniformly, we expect the number of steps taken to level completion to rise linearly with the number of classes.

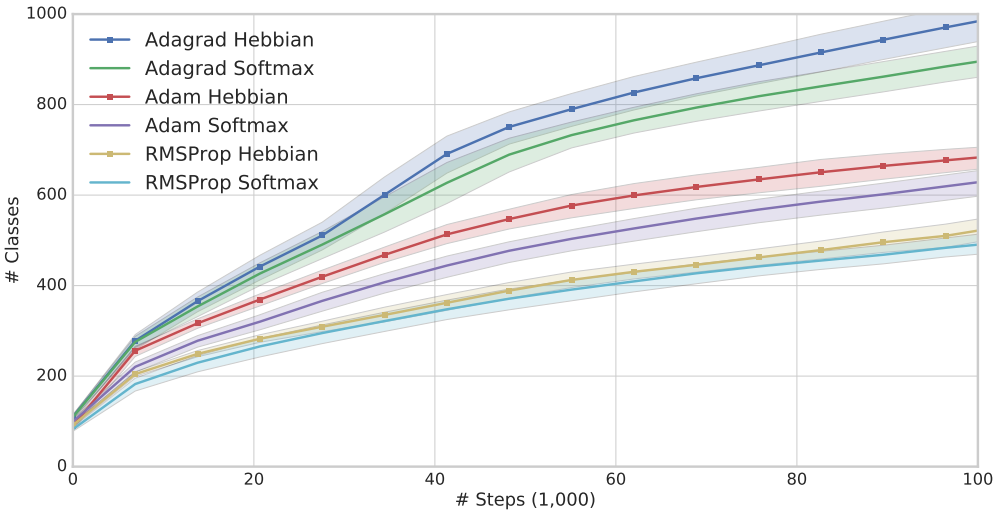


Figure 4.5: Omniglot curriculum task. Starting from 30 classes, 5 new classes are added when total test error exceeds 60%. Each line shows a 2-σ confidence band obtained from 10 independent seed runs. The Hebbian Softmax uses hyper-parameters $T = 10$ and $\gamma = 0.1$. The learning rate chosen for AdaGrad was 0.08, and 0.006 for RMSProp and Adam — these were obtained from a prior sweep with the baseline softmax model.

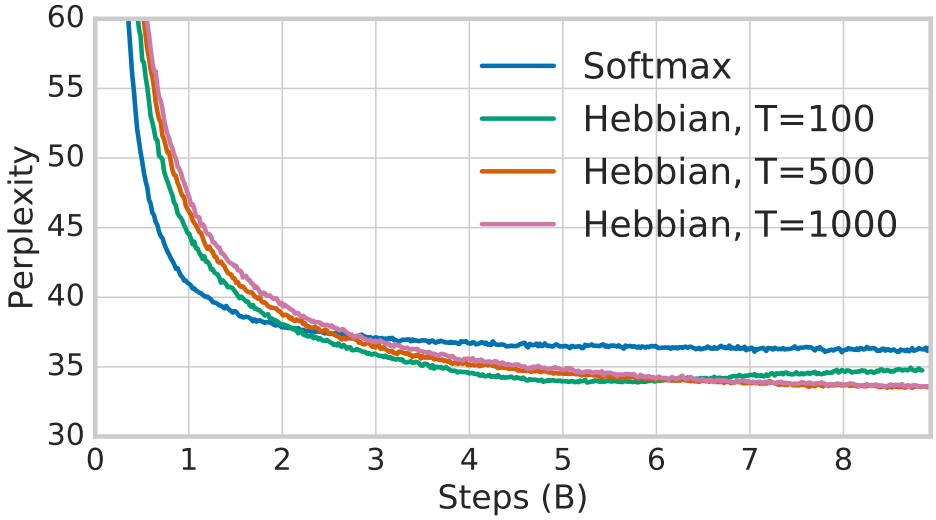


Figure 4.6: Validation perplexity for WikiText-103 over 9 billion words of training ($\approx$ 90 epochs). The LSTM drops to a perplexity of 36.4 with a regular softmax layer, and 34.3 with the Hebbian Softmax , $T = 500$, when representations from the LSTM begin to settle. For tuning parameter T; $T = 100$ converges quicker, but begins to overfit after 5.5B training words (coinciding when all classes have been observed at least 100 times).

4.3.2 Language Modelling

We evaluate the Hebbian Softmax in the context of a large-scale classification task, where some classes are infrequently observed. Word-level language modelling is an ideal fit because it satisfies both criteria, and there are established performance benchmarks. Some large-scale language modelling corpora require the use of efficient softmax approximations, such as the adaptive softmax (Grave et al., 2016a) or hierarchical softmax (Goodman, 2001) due to the very large vocabulary size. To reduce confounding factors, we restrict ourselves to applications where the full softmax can be used. We investigate two medium-sized corpora, WikiText-103 which contains just over $100M$ tokens derived from Wikipedia articles (Merity et al., 2016), and Gutenberg which contains a subset of open-access texts from

Project Gutenberg. The idea is that Wikipedia articles should cover factual information, where the style of writing is somewhat consistent and named entities may appear across many articles; whereas books should be more self-contained (unique named entities) and stylistically different. We also consider a very large corpus, GigaWord v5, which is a collection of articles from eight press associations exceeding a decade's worth of global news.

We selected the baseline model to be a single-layer LSTM with 2048 units, tied input/output embedding parameters, and an embedding dropout rate of 0.3. These were selected from a baseline sweep on WikiText-103. Hyper-parameters and further training details are described in Appendix B.1.

WikiText-103

The WikiText-103 corpus contains $267,735$ unique words and each word occurs at least three times in the training set. We take the best LSTM parameter configuration (described above) as a baseline, and compare it to an identical model where the final layer is replaced with Hebbian Softmax . We swept over the insertion limit parameter $T \in \{100, 500, 1000\}$ and discount factor $\gamma \in \{0.05, 0.1, 0.25\}$ using the validation set. We found $T = 500$, $\gamma = 0.25$ worked best, achieving a test perplexity of 34.3 on this dataset (Table 4.1). Inspecting the validation curves in Figure 4.6 we see the Hebbian Softmax initially hampers validation performance, until around 2–3B training tokens have been consumed. This makes sense, as storing activations from prior layers of the network is only an effective strategy once the network has rich intermediate representations of its inputs. Table 4.2 shows the test perplexity broken down by word frequency, we see the gain in overall

Table 4.1: Validation and test perplexities on WikiText-103.

	Valid.	Test
LSTM (Grave et al., 2016b)	-	48.7
Temporal CNN (Bai et al., 2018a)	-	45.2
Gated CNN (Dauphin et al., 2016)	-	37.2
LSTM (ours)	36.0	36.4
LSTM + Cache	34.5	34.8
LSTM + Hebbian	34.1	34.3
LSTM + Hebbian + Cache	29.7	29.9
LSTM + Hebbian + Cache + MbPA	**29.0**	**29.2**

performance is obtained from less frequent vocabulary.

We also investigate the model evaluated dynamically on the test using (a) a Neural Cache (Grave et al., 2016b) and (b) Memory-based Parameter Adaptation (MbPA) (Sprechmann et al., 2018). Hyper-parameter details for these models are detailed in Appendix B.2. The cache reduces the test perplexity by 1.6 for the LSTM and 4.4 for LSTM + Hebbian Softmax . The addition of MbPA reaches a test perplexity of 29.2 which is, to the authors' knowledge, state-of-the-art at time of writing.

We inspected whether this gain was optimisation-specific by also training a model using the AdaGrad optimiser, however we found this degraded performance, obtaining a perplexity of 60 after 20B steps.

Gutenberg

Books provide several different linguistic challenges to articles. The style of writing is intentionally varied between authors, and named entities can be wholly fictional — confined to a single text. We extract a subset of English-language books from

Table 4.2: Test perplexity versus training word frequency. Hebbian Softmax models less frequent words with better accuracy. Note the training set size of WikiText is smaller than Gutenberg, which is itself much smaller than GigaWord; so the $>$ 10K bucket includes an increasing number of unique words. This explains GigaWord's larger perplexity in this bucket. Furthermore there were no words observed $<$ 100 times within the GigaWord 250K vocabulary. A random model would have a perplexity of $|V| \approx 2.5e5$ for all frequency buckets.

	$>$ 10K	1K-10K	100-1K	$<$ 100	All
WikiText-103					
softmax	12.1	2.2e2	1.2e3	9.7e3	36.4
Hebbian Softmax	12.1	**1.8e2**	**7.6e2**	**5.2e3**	**34.3**
Gutenberg					
softmax	19.0	9.8e2	6.9e3	8.6e4	47.9
Hebbian Softmax	**18.1**	**9.4e2**	**6.6e3**	**5.9e4**	**45.5**
GigaWord					
softmax	39.4	6.5e3	3.7e4	-	53.5
Hebbian Softmax	**33.2**	**3.2e3**	**1.6e4**	-	**43.7**

the corpus, strip the Gutenberg headers and tokenise the text. We select a dataset of comparable size to WikiText-103; 2,042 books in total with 2,017 training books (175,181,505 tokens), 12 validation books (609,545 tokens), and 13 test books (526,646 tokens). We select all words that occur at least five times in the training set, a total vocabulary of 242,621 and map the remainder to an 'unk' token.

We use the same LSTM hyper-parameters as those chosen from the Wikipedia sweep, and compare against Hebbian Softmax with $T = 100$, $T = 500$ and $\gamma = 0.1$. Figure 4.7 shows the validation performance after 15B steps of training, equating to roughly 80 epochs and 6 days of training with 8 P100s training synchronously. After approximately 4B steps of training the softmax performance is surpassed, and this gap widens even up to 15B steps to a gap of 2-3 points in

perplexity. Similar to WikiText-103, we see in Table 4.2 the gain in perplexity is more pronounced over less frequent words.

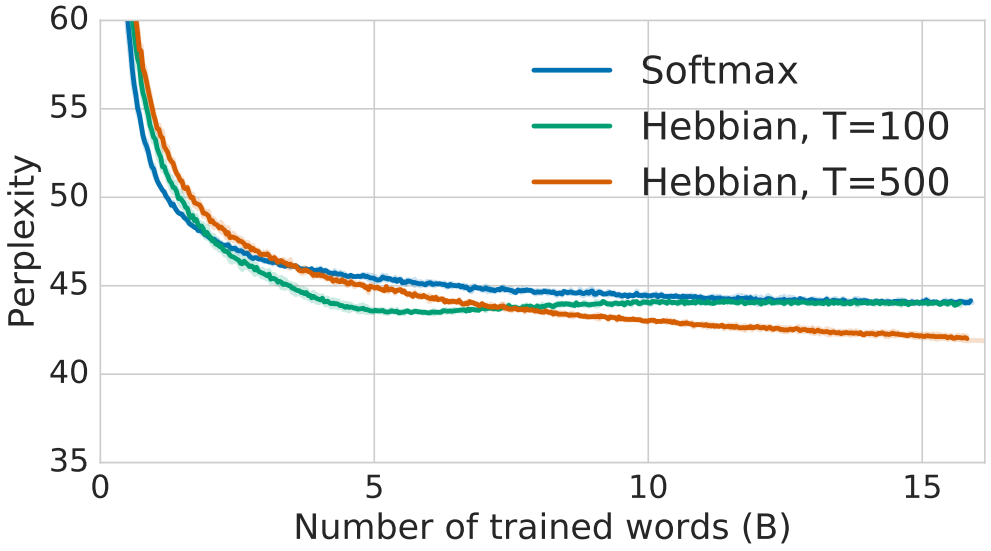


Figure 4.7: Validation perplexity on Gutenberg. All word classes have been observed after around 4B training tokens and we observe the performance of Hebbian Softmax return to that of the vanilla LSTM thereafter, as all parameters are optimised by gradient descent.

GigaWord v5

We evaluate Hebbian Softmax on a large-scale language modelling corpus. GigaWord is interesting because it is a vast collection of news articles, and there is a natural temporal order. We pre-process the dataset, select all articles from 2000-2009 for the training set, and test on all articles from 2010. The total number of training tokens is 4.0B and the total number of test tokens is 260M. The total unique tokens (after pre-processing) for the training set reaches 6M, however for parity with the other experiments we choose a vocabulary size of 250K. We use the same LSTM hyper-parameters and Hebbian Softmax hyper-parameters, and train

the model for 6B steps, after which the models plateau in evaluation performance. We observe a 9.8-point drop in perplexity, from 53.5 to 43.7, illustrated in Table 4.2.

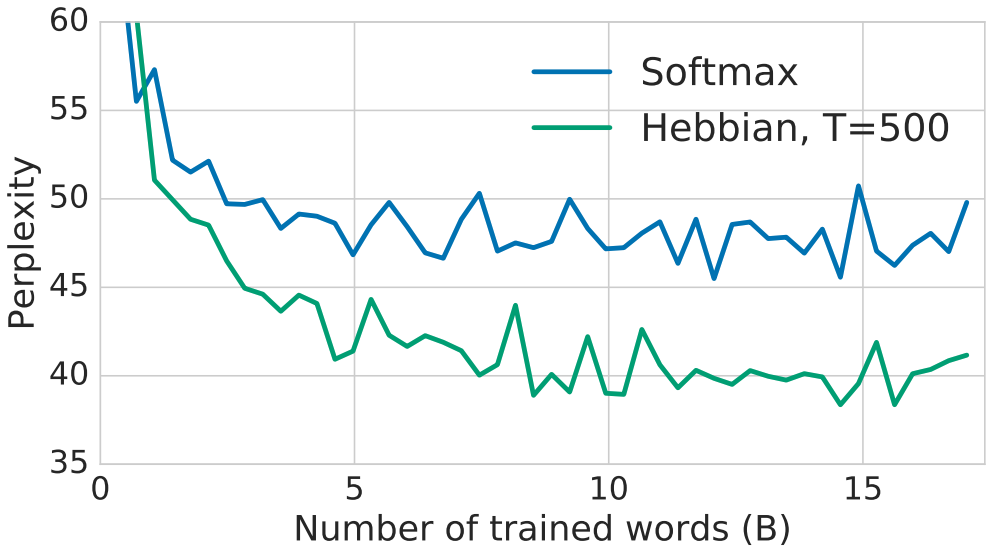


Figure 4.8: Test perplexity on GigaWord v5. Each model is trained on all articles from $2000 - 2009$ and tested on 2010. Because the test set is very large, a random subsample of articles are used per evaluation cycle. For this reason, the measurements are more noisy.

4.3.3 Softmax Approximations

So far we have always used the full softmax as a baseline. This is to make experimental comparisons straightforward, however in many applications the full softmax is too expensive to compute. We now consider the interaction between the Hebbian Softmax update rule and computationally efficient softmax approximations, namely the *sampled softmax* (Jean et al., 2014). When the baseline language model is trained on WikiText-103 with a sampled softmax (using 8192 samples) we see in Figure 4.9 that the learning update from Hebbian Softmax improves

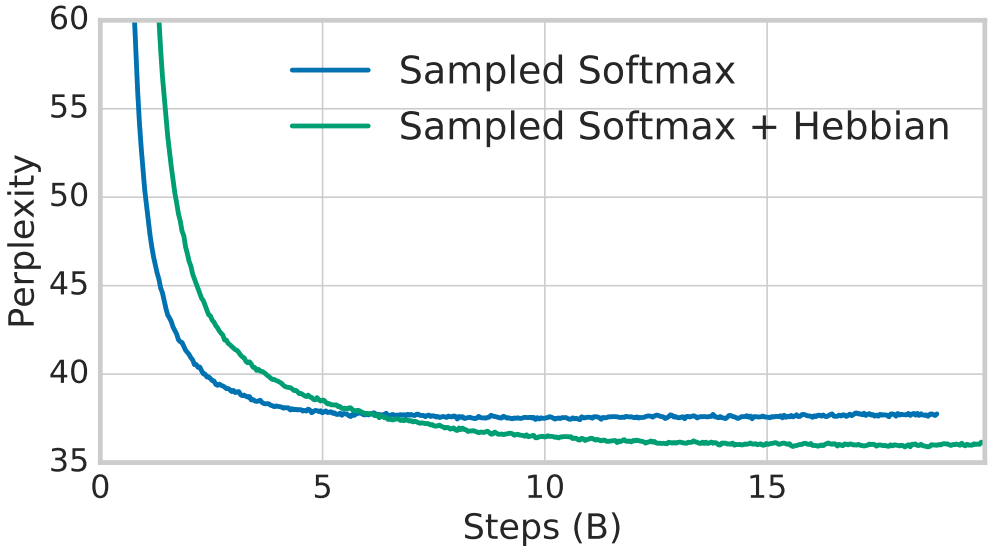


Figure 4.9: Interaction with Sampled Softmax. Validation curve on WikiText-103 when using a sampled softmax (Jean et al., 2014) with 8192 samples. Due to the smaller memory overhead, we trained with a batch size of 256 (vs 64 when using the full softmax) using 2 P100 GPUs instead of 8. The total batch size of 512 is kept the same, however training wall time is reduced from 6 days to 2. We see an improvement when using the Hebbian Softmax however both models plateau at $2-3$ perplexity points higher than the exact softmax.

upon the sampled softmax by approximately 2 perplexity points, however both models plateau $2-3$ perplexity points higher than the exact softmax models from Section 4.3.2.

4.3.4 Alternate Objective Functions

We test out some of the alternate inner objective functions described in (4.9) from Section 4.2.4. The inner objective functions include *Euclidean, Inner Product, SVM, (sparse) Softmax*. These could be applied to any of the described experiments, we chose the WikiText-103 language modelling task because it is more comparable to prior work.

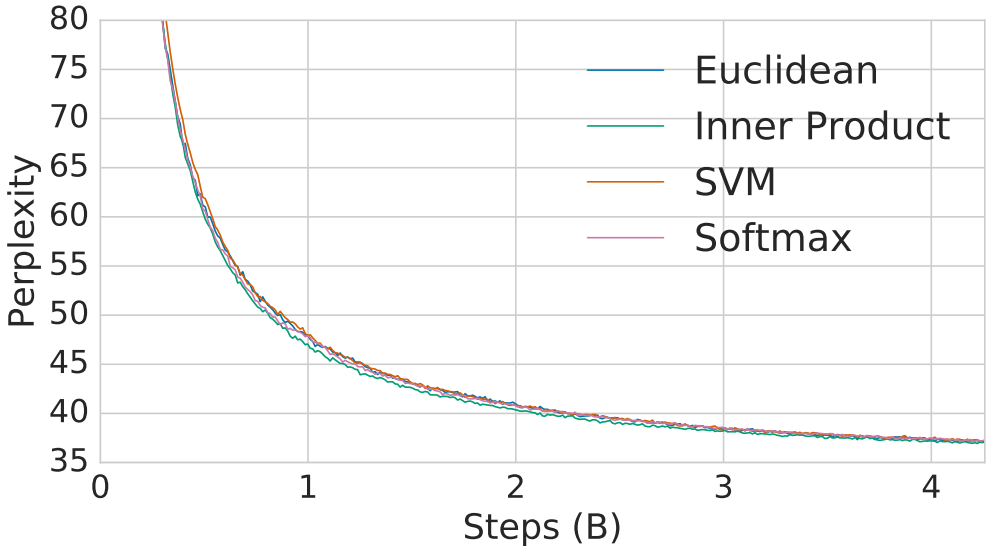


Figure 4.10: Objective Function Comparison. Validation learning curves for WikiText-103 comparing different overfitting objectives as illustrated in (4.9). We observe there is not a significant improvement in performance by choosing inner objectives which relate to the overall training objective, e.g. Softmax, vs $L2$.

Although more expressive objective functions appear promising, in practice we find that validation performance is roughly equivalent between all inner objective functions (Figure 4.10). This suggests the network activation h_t naturally do not land too close to other class parameters, and the norm of activations is not too large or small, in comparison to the model parameters θ. The latter may be due to the use of layer normalisation from the LSTM.

4.4 Related Work

Few-shot classification has been investigated in a meta-learning setup with a mixture model of a parametric neural network and a non-parametric memory (Santoro et al., 2016a; Vinyals et al., 2016). Here, a subset of classes are used with per-

muted labels per episode, activations are stored to memory, and gradients are passed through the memory. This allows the network to shape its activations to be conducive to accurate retrieval and classification. In this study we do not meta-learn the activations stored into network parameters and instead rely on their representation being rich enough from regular training. We do this to avoid back-propagating through time to the point of memory write, which is impractical when memories are stored millions of time steps ago, such as in the case of modelling rare words.

In natural language processing memory-augmented models have been shown to improve the modelling of unknown words and adaptation to new domains (Grave et al., 2016b; Merity et al., 2016; Kawakami et al., 2017). However in these works the memory is typically small and models the recent past. During evaluation the test activations and corresponding labels are stored in memory, and the model is evaluated dynamically — adapting to the test data on the fly. Whilst dynamic evaluation provides insights into domain transfer, it is limited in applicability as the model may not receive ground-truth labels when launched into production.

More recent work has investigated methods of memorising and searching over the training set to enhance performance (Kaiser et al., 2017; Grave et al., 2017; Gu et al., 2017). These approaches typically require complex engineering to efficiently index this memory store. Part of the benefit of the Hebbian Softmax is implementation simplicity.

Prior literature on the softmax operator for language modelling computational efficiency (Chen et al., 2015; Grave et al., 2016a) or tricks such as smoothing across many softmax layers (Yang et al., 2017). However these do not focus on

increasing the data-efficiency or faster learning of infrequent classes.

Other architectures have been considered for fast learning, such as the 'fast weights' auto-associative memory (Ba et al., 2016a). This focuses on fast adaptation to recent information that persists over a short window of time. The LEABRA architecture (O'Reilly, 1996a) contains a mixture of contrastive Hebbian learning (GENEREC) (O'Reilly, 1996b) and gradient descent for fast and slow learning, however this cognitively-inspired model has not been shown to scale to large-scale classification problems.

4.5 Conclusion

This chapter explores one way in which we can achieve fast parametric learning in neural networks, and preserve this knowledge over time. We show that activation memorisation is useful for vision in the binding of newly introduced classes, beating well tuned adaptive learning rate optimisers, RMSProp and AdaGrad.

For language we show improvement in the modelling of text with an extensive vocabulary. In the latter we show the model beats a very strong LSTM benchmark on three stylistically different corpora, and achieves state of the art on WikiText-103. This is achieved with effectively no additional compute or memory resources. Breaking down perplexity over word frequency bucket, we see that less frequent words are better modelled, as hypothesised. We suggest that the Hebbian Softmax could be applied to any classification domain with infrequent classes, or non-stationary data. It may also be useful in quickly adapting a pre-trained classifier to a new task / set of classes — however this is beyond the scope

of our initial investigation.

It would also be interesting to explore activation memorisation deeper within the network, and thus in more general scenarios to classification. More recent work that has occurred since this study has investigated Hebbian update rules throughout the entire network (Miconi et al., 2018) and this has been incorporated for networks to solve sequence learning tasks where memory is required. However this has not yet been demonstrated to improve fast continual learning or better transfer, thus there is an exciting opportunity to create networks which mix slow learning with fast activation memorisation across all layers towards this goal.

Chapter 5

Compressive Memory for Data Structures

There has been a recent trend in training neural networks to replace data structures that have been crafted by hand, with an aim for faster execution, better accuracy, or greater compression. In this setting, a neural data structure is instantiated by training a network over many epochs of its inputs until convergence. In applications where inputs arrive at high throughput, or are ephemeral, training a network from scratch is not practical. This motivates the need for few-shot neural data structures. In this chapter we explore the learning of approximate set membership over a set of data in one-shot via meta-learning. We propose a novel memory architecture, the Neural Bloom Filter, which is able to achieve significant compression gains over classical Bloom Filters and existing memory-augmented neural networks.

5.1 Motivation

One of the simplest questions one can ask of a set of data is whether or not a given query is contained within it. Is q, our query, a member of S, our chosen set of observations? This *set membership* query arises across many computing domains; from databases, network routing, and firewalls. One could query set membership by storing S in its entirety and comparing q against each element. However, more space-efficient solutions exist.

The original and most widely implemented *approximate set membership* data-structure is the Bloom Filter (Bloom, 1970). It works by storing sparse distributed codes, produced from randomized hash functions, within a binary vector. The Bloom-filter trades off space for an allowed false positive rate, which arises due to hash collisions. However its error is one-sided; if an element q is contained in S then it will always be recognized. It never emits false negatives. One can find Bloom Filters embedded within a wide range of production systems; from *network security* (Geravand and Ahmadi, 2013), to block malicious IP addresses; *databases*, such as Google's Bigtable (Chang et al., 2008), to avoid unnecessary disk lookups; *cryptocurrency* (Hearn and Corallo, 2012), to allow clients to filter irrelevant transactions; *search*, such as Facebook's typeahead search (Adams, 2010), to filter pages which do not contain query prefixes; and *program verification* (Dillinger and Manolios, 2004), to avoid recomputation over previously observed states.

While the main appeal of Bloom Filters is favourable compression, another important quality is the support for dynamic updates. New elements can be inserted in $\mathcal{O}(1)$ time. This is not the case for all approximate set membership data structures. For example, perfect hashing saves $\approx 40\%$ space over Bloom Filters but

requires a pre-processing stage that is polynomial-time in the number of elements to store (Dietzfelbinger and Pagh, 2008). Whilst the static set membership problem is interesting, it limits the applicability of the algorithm. For example, in a database application that is serving a high throughput of write operations, it may be intractable to regenerate the full data-structure upon each batch of writes.

We thus focus on the data stream computation model (Muthukrishnan et al., 2005), where input observations are assumed to be ephemeral and can only be inspected a constant number of times — usually once. This captures many real-world applications: network traffic analysis, database query serving, and reinforcement learning in complex domains. Devising an approximate set membership data structure that is not only more compressive than Bloom Filters, but can be applied to either dynamic or static sets, could have a significant performance impact on modern computing applications. In this chapter we investigate this problem using memory-augmented neural networks and meta-learning.

We build upon the recently growing literature on using neural networks to replace algorithms that are configured by heuristics, or do not take advantage of the data distribution. For example, Bloom Filters are indifferent to the data distribution. They have near-optimal space efficiency when data is drawn uniformly from a universe set (Carter et al., 1978) (maximal-entropy case) but (as we shall show) are sub-optimal when there is more structure. Prior studies on this theme have investigated compiler optimization (Cummins et al., 2017), computation graph placement (Mirhoseini et al., 2017), and data index structures such as b-trees (Kraska et al., 2018). In the latter work, Kraska et al. (2018) explicitly consider the problem of static set membership. By training a neural network over a fixed S (in their

case, string inputs) along with held-out negative examples, they observe 36% space reduction over a conventional Bloom Filter.[1] Crucially this requires iterating over the storage set S a large number of times to embed its salient information into the weights of a neural network classifier. For a new S this process would have to be repeated from scratch.

Instead of learning from scratch, we draw inspiration from the few-shot learning advances obtained by meta-learning memory-augmented neural networks (Santoro et al., 2016b; Vinyals et al., 2016). In this setup, tasks are sampled from a common distribution and a network learns to specialize to (learn) a given task with few examples. This matches very well to applications where many Bloom Filters are instantiated over different subsets of a common data distribution. For example, a Bigtable database usually contains one Bloom Filter per SSTable file. For a large table that contains Petabytes of data, say, there can be over $100,000$ separate instantiated data-structures which share a common row-key format and query distribution. Meta-learning allows us to exploit this common redundancy. We design a database task with similar redundancy to investigate this exact application in Section 5.6.4.

The main contributions of this chapter are (1) A new memory-augmented neural network architecture, the *Neural Bloom Filter*, which learns to write to memory using a distributed write scheme, and (2) An empirical evaluation of the Neural Bloom Filter meta-learned on one-shot approximate set membership problems of varying structure. We compare with the classical Bloom Filter along-

[1]The space saving increases to 41% when an additional trick is incorporated, in discretizing and re-scaling the classifier outputs and treating the resulting function as a hash function to a bit-map.

side other memory-augmented neural networks such as the Differentiable Neural Computer (Graves et al., 2016) and Memory Networks (Sukhbaatar et al., 2015). We find when there is no structure, that differentiates the query set elements and queries, the Neural Bloom Filter learns a solution similar to a Bloom Filter derivative — a Bloom-g filter (Qiao et al., 2011) — but when there is a lot of structure the solution can be considerably more compressive (e.g. 30× for a database task).

5.2 Compression in Memory-Augmented Neural Networks

Recurrent neural networks such as LSTMs retain a small amount of memory via the recurrent state. However this is usually tied to the number of trainable parameters in the model. There has been recent interest in augmenting neural networks with a larger external memory. The method for doing so, via a differentiable write and read interface, was first popularized by the Neural Turing Machine (NTM) (Graves et al., 2014) and its successor the Differentiable Neural Computer (DNC) (Graves et al., 2016) in the context of learning algorithms, and by Memory Networks (Sukhbaatar et al., 2015) in the context of question answering. Memory Networks store embeddings of the input in separate rows of a memory matrix M. Reads are performed via a differentiable *content-based addressing* operation. The NTM and DNC use the same content-based read mechanism, but also learns to write based upon content (content-based writes), temporally ordered locations, or unused memory. The soft write schemes do distribute write words across all slots

in memory however in practice they have a very peaked distribution around certain slots. The sparse extensions of these models that we explore in Chapter 3 have even more selective allocations of write words to memory slots. Memory Networks allocate a single memory slot to a given memory — the most selective scheme across all models. Thus there is very little emphasis on distributing write words across many slots, and having a joint compression of past memories across the different rows in M our memory matrix.

When it comes to capacity, there has been consideration to scaling both the DNC and Memory Networks to very large sizes using sparse read and write operations, as covered in Chapter 3. However another way to increase the capacity is to increase the amount of compression which occurs in memory. The Kanerva Machine (Wu et al., 2018a,b) tackles memory-wide compression using a distributed write scheme to jointly compose and compress its memory contents. The model uses content-based addressing over a separate learnable addressing matrix A, instead of the memory M, and thus learns *where* to write. Alongside the general motivation from data-structures, we take inspiration from this Kanerva Machine write scheme.

5.3 Model

One approach to learning set membership in one-shot would be to use a recurrent neural network, such as an LSTM or DNC. Here, the model sequentially ingests the N elements to store, answers a set of queries using the final state, and is trained by BPTT. Whilst this is a general training approach, and the model may learn a compressive solution, it does not scale well to larger number of elements.

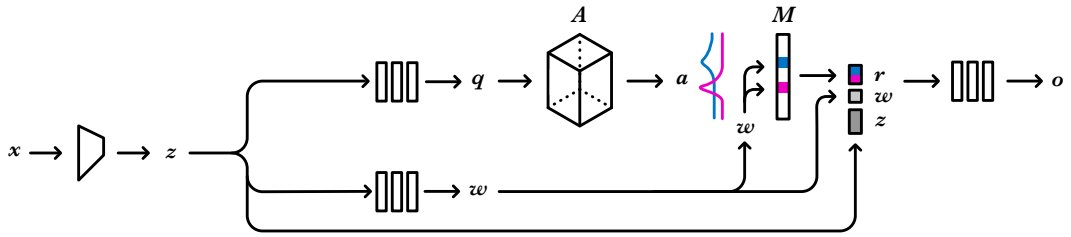


Figure 5.1: Overview of the Neural Bloom Filter architecture.

Even when $N = 1000$, backpropagating over a sequence of this length induces computational and optimization challenges. For larger values this quickly becomes intractable. Alternatively one could store an embedding of each element $x_i \in S$ in a slot-based Memory Network. This is more scalable as it avoids BPTT, because the gradients of each input can be calculated in parallel. However Memory Networks are not a space efficient solution (as shown in in Section 5.6.2) because there is no joint compression of inputs.

Algorithm 2 Neural Bloom Filter

def controller(x):
    $z \leftarrow f_{enc}(x)$ {Input embedding}
    $q \leftarrow f_q(z)$ {Query word}
    $a \leftarrow \sigma(q^T A)$ {Memory address}
    $w \leftarrow f_w(z)$ {Write word}

def write(x):
    $a, w \leftarrow$ controller(x)
    $M_{t+1} \leftarrow M_t + wa^T$ {Additive write}

def read(x):
    $a, w, z \leftarrow$ controller(x)
    $r \leftarrow$ flatten$(M \odot$ a$)$ {Read words}
    $o \leftarrow f_{out}([r, w, z])$ {Output logit}

This motivates the proposed memory model, the Neural Bloom Filter. Briefly, the network is augmented with a real-valued memory matrix. The network *addresses*

memory by classifying which memory slots to read or write to via a softmax, conditioned on the input. We can think of this as a continuous analogue to the Bloom Filter's hash function; because it is learned the network can co-locate or separate inputs to improve performance. The network updates memory with a simple *additive write* operation — i.e. no multiplicative gating or squashing — to the addressed locations. An additive write operation can be seen as a continuous analogue to the the Bloom Filter's logical OR write operation.

Crucially, the additive write scheme allows us to train the model without BPTT — this is because gradients with respect to the write words $\partial L/\partial w = (\partial L/\partial M)^T a$ can be computed in parallel. Reads involve a component-wise multiplication of address and memory (analogous to the selection of locations in the Bloom Filter via hashing), but instead of projecting this down to a scalar with a fixed function, we pass this through an MLP to obtain a scalar familiarity logit. The network is fully differentiable, allows memories to be stored in a distributed fashion across slots, and is quite simple e.g. in comparison to DNCs.

The full architecture depicted in Figure 5.1 consists of a *controller network* which encodes the input to an embedding $z \leftarrow f_{enc}(x)$ and transforms this to a write word $w \leftarrow f_w(z)$ and a query $q \leftarrow f_q(z)$. The address over memory is computed via a softmax $a \leftarrow \sigma(q^T A)$ over the content-based attention between q and a learnable address matrix A. Here, σ denotes a softmax. All components of the network are differentiable and are trained end-to-end. The network thus learns where to place elements or overlap elements based on their content, we can think of this as a soft and differentiable relaxation of the uniform hashing families incorporated by the Bloom Filter (see Section 5.4.1 for further discussion).

A *write* is performed by running the controller to obtain a write word w and address a, and then additively writing w to M, weighted by the address a, $M_{t+1} \leftarrow M_t + wa^T$. The simple additive write ensures that the resulting memory is invariant to input ordering (as addition is commutative) and we do not have to backpropagate-through-time (BPTT) over sequential writes — gradients can be computed in parallel.

A *read* is performed by also running the controller network to obtain z, w, and a and component-wise multiplying the address a with M, $r \leftarrow M \odot a$. The read words r are fed through an MLP along with the residual inputs w and z, and are projected to a single scalar logit, indicating the familiarity signal. These write words are used during the read operation to provide signal for familiarity, as they tell the network whether a retrieved word bears resemblance to what would have been written from a given input. We found this read scheme to be more powerful than the conventional read operation $r \leftarrow a^T M$ used by the DNC and Memory Networks, as it allows for non-linear interactions between rows in memory at the time of read. See Algorithm 2 for an overview of the operations.

To give an example network configuration, we chose f_{enc} to be a 3-layer CNN in the case of image inputs, and a 128-hidden-unit LSTM in the case of text inputs. We chose f_w and f_q to be an MLP with a single hidden layer of size 128, followed by layer normalization, and f_{out} to be a 3-layer MLP with residual connections. We used a leaky ReLU as the non-linearity. Although the described model uses dense operations that scale linearly with the memory size m, we discuss how the model could be implemented for $\mathcal{O}(\log m)$ time reads and writes using sparse attention and read/write operations, in Section 5.9. Furthermore the model's relation to

uniform hashing is discussed in Section 5.4.1.

5.4 Space Complexity

In this section we discuss space lower bounds for the approximate set membership problem when there is some structure to the storage or query set. This can help us formalise why and where neural networks may be able to beat classical lower bounds to this problem.

The $n \log_2 (1/\epsilon)$ lower bound from Carter et al. (1978) assumes that all subsets $S \subset U$ of size n, and all queries $q \in U$ have equal probability. Whilst it is instructive to bound this maximum-entropy scenario, which we can think of as 'worst case', most applications of approximate set membership e.g. web cache sharing, querying databases, or spell-checking, involve sets and queries that are not sampled uniformly. For example, the elements within a given set may be highly dependent, there may be a power-law distribution over queries, or the queries and sets themselves may not be sampled independently.

A more general space lower bound can be defined by an information theoretic argument from communication complexity (Yao, 1979). Namely, approximate set membership can be framed as a two-party communication problem between Alice, who observes the set S and Bob, who observes a query q. They can agree on a shared policy Π in which to communicate. For given inputs S, q they can produce a transcript $A_{S,q} = \Pi(S, q) \in \mathcal{Z}$ which can be processed $g : \mathcal{Z} \to 0, 1$ such that $\mathbb{P}\left(g(A_{S,q}) = 1 | q \notin S\right) \leq \epsilon$. This transcript can be thought of as some piece of communication between Alice and Bob which is optimally space efficient, and can

be used to make a decision over whether $q \in S$. Bar-Yossef et al. (2004) shows that the maximum transcript size, over all possible sets and queries, is greater than the mutual information between the inputs and transcript: $\max_{S,q} |A_{S,q}| \geq I(S, q; A_{S,q}) = H(S, q) - H(S, q|A_{S,q})$. Thus we note problems where we may be able to use less space than the classical lower bound are cases where the entropy $H(S, q)$ is small, e.g. our sets are highly non-uniform, or cases where $H(S, q|A_{S,q})$ is large, which signifies that many query and set pairs can be solved with the same transcript.

5.4.1 Relation to uniform hashing

We can think of the decorrelation of s, along with the sparse content-based attention with A, as a hash function that maps s to several indices in M. For moderate dimension sizes of s (256, say) we note that the Gaussian samples in A lie close to the surface of a sphere, uniformly scattered across it. If q, the decorrelated query, were to be Gaussian then the marginal distribution of nearest neighbours rows in A will be uniform. If we chose the number of nearest neighbours $k = 1$ then this implies the slots in M are selected independently with uniform probability. This is the exact hash function specification that Bloom Filters assume. Instead we use a continuous (as we choose $k > 1$) approximation (as we decorrelate $s \to q$ vs Gaussianize) to this uniform hashing scheme, so it is differentiable and the network can learn to shape query representations.

5.5 Backup Bloom Filters

For each task we compare the model's memory size, in bits, at a given false positive rate — usually chosen to be 1%. This memory size is measured as the size of the network's memory state. For our neural networks which output a probability $p = f(x)$ one could select an operating point τ_ϵ such that the false positive rate is ϵ. In all of our experiments the neural network outputs a memory (state) s which characterizes the storage set. Let us say `SPACE(f, ϵ)` is the minimum size of s, in bits, for the network to achieve an average false positive rate of ϵ. We could compare `SPACE(f,ϵ)` with `SPACE(Bloom Filter,ϵ)` directly, but this would not be a fair comparison as our network f can emit false negatives.

To remedy this, we employ the same scheme as Kraska et al. (2018) where we use a 'backup' Bloom Filter with false positive rate δ to store all false negatives. When $f(x) < \tau_\epsilon$ we query the backup Bloom Filter. Because the overall false positive rate is $\epsilon + (1 - \epsilon)\delta$, to achieve a false positive rate of at most α (say 1%) we can set $\epsilon = \delta = \alpha/2$. Thus the total space can be calculated,

$$\texttt{TOTAL\_SPACE}(f, \alpha) = \texttt{SPACE}(f, \frac{\alpha}{2}) + n_{fn} * \texttt{SPACE(Bloom Filter,}\frac{\alpha}{2})$$

Where n_{fn} is the number of false negatives, i.e. the number of elements stored in the backup bloom filter. We compare this quantity for different storage set sizes.

Algorithm 3 Meta-Learning Training

Let S^{train} denote the distribution over sets to store.
Let Q^{train} denote the distribution over queries.
for $i = 1$ **to** max train steps **do**
  Sample task:
    Sample set to store: $S \sim \mathcal{S}^{train}$
    Sample t queries: $x_1, \ldots, x_t \sim Q^{train}$
    Targets: $y_j = 1$ if $x_j \in S$ else 0; $j = 1, \ldots, t$
  Write entries to memory: $M \leftarrow f_\theta^{write}(S)$
  Calculate logits: $o_j = f_\theta^{read}(M, x_j)$; $j = 1, \ldots, t$
  XE loss: $L = \sum_{j=1}^{t} y_j \log o_j + (1 - y_j)(1 - \log o_j)$
  Backprop through queries and writes: $dL/d\theta$
  Update parameters: $\theta_{i+1} \leftarrow \text{Optimizer}(\theta_i, dL/d\theta)$
end for

5.6 Experiments

Our experiments explore scenarios where set membership can be learned in one-shot with improved compression over the classical Bloom Filter. We consider tasks with varying levels of structure in the storage sets S and queries q. We compare the Neural Bloom Filter with three memory-augmented neural networks, the LSTM, DNC, and Memory Network, that are all able to write storage sets in one-shot.

For the Bloom Filter baselines we consider the optimal storage space of Bloom Filters instead of relying on a specific implementation, which provides it a very slight advantage. However if we were to rely on a particular implementation, it's worth noting the data-structure receives and hashes the raw input instead of processing the input from an encoder network. For the neural network models, the training setup follows the memory-augmented meta-learning training scheme of Vinyals et al. (2016), only here the task is familiarity classification versus image

classification. The network samples tasks which involve classifying familiarity for a given storage set. Meta-learning occurs as a two-speed process, where the model quickly learns to recognize a given storage set S within a training episode via writing to a memory or state, and the model slowly learns to improve this fast-learning process by optimizing the model parameters θ over multiple tasks. We detail the training routine in Algorithm 3.

For the RNN baselines (LSTM and DNC) the write operation corresponds to unrolling the network over the inputs and outputting the final state. For these models, the query network is simply an MLP classifier which receives the concatenated final state and query, and outputs a scalar logit. For the Memory Network, inputs are stored in individual slots and the familiarity signal is computed from the maximum content-based attention value. The Neural Bloom Filter read and write operations are defined in Algorithm 2.

5.6.1 Method of Space Comparison

Comparing models in terms of the size of their memory and their resulting performance requires careful thinking. We compared the space (in bits) of the model's memory (or state) to a Bloom Filter at a given false positive rate and 0% false negative rate. The false positive rate is measured empirically over a sample of $50,000$ queries for the learned models; for the Bloom Filter we employ the analytical false positive rate. Beating a Bloom Filter's space usage with the analytical false positive rate implies better performance for any given Bloom Filter library version (as actual Bloom Filter hash functions are not uniform), thus the comparison is reasonable.

For each model we sweep over hyper-parameters relating to model size to obtain their smallest operating size at the desired false positive rate (for the full set, see Appendix C.2). Because the neural models can emit false negatives, we store these in a (ideally small) backup Bloom Filter, as proposed by Kraska et al. (2018); Mitzenmacher (2018a). We account for the space of this backup Bloom Filter, and add it to the space usage of the model's memory for parity (See Section 5.5 for further discussion).

The neural network must learn to output a small state in one-shot that can serve set membership queries at a given false positive rate, and emit a small enough number of false negatives such that the backup filter is also small, and the total size is considerably less than a Bloom Filter.

5.6.2 Sampling Strategies on MNIST

To understand what kinds of scenarios neural networks may be more (or less) compressive than classical Bloom Filters, we consider three simple set membership tasks that have a graded level of structure to the storage sets and queries. Concretely, they differ in the sampling distribution of storage sets $\mathcal{S}^{train}$ and queries $\mathcal{Q}^{train}$. However all problems are approximate set membership tasks that can be solved by a Bloom Filter.

The tasks are *(1) Class-based familiarity*, a highly structured task where each set of images is sampled with the constraint that they arise from the same randomly-selected class. This task emulates when there are implicit class attributes that set elements share in common. *(2) Non-uniform instance-based familiarity*, a moderately structured task where the images are sampled non-uniformly without re-

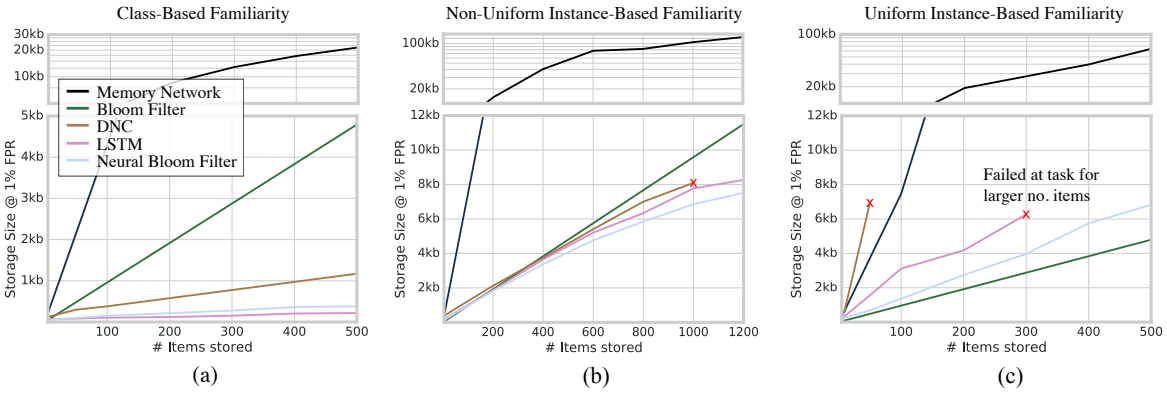


Figure 5.2: Sampling strategies on MNIST. Space consumption at 1% FPR.

placement. This task represents settings where heavy hitters exist in the data, that are more likely to be present. *(3) Uniform instance-based familiarity*, a completely unstructured task where each subset contains images sampled uniformly without replacement. This represents the adversarial setting when there is no structure in the storage sets.

For each task we varied the size of the sample set to store, and calculated the space (in bits) of each model's state at a fixed false positive rate of 1% and a false negative rate of 0%. We used relatively small storage set sizes (e.g. $100 - 1000$) to start with, as this highlights that some RNN-based approaches struggle to train over larger set sizes, before progressing to larger sets in subsequent sections. See Appendix C.3 for further details on the task setup and Appendix C.1 for further details on the convnet image encoder.

In the *class-based sampling* task we see in Figure 5.2a that the DNC, LSTM and Neural Bloom Filter are able to significantly outperform the classical Bloom Filter when images are sampled by class. The Memory Network is able to solve the task with a word size of only 2, however this corresponds to a far greater number of

bits per element, 64 versus the Bloom Filter's 9.8 (to a total size of 4.8kb), and so the overall size was prohibitive. The DNC, LSTM, and Neural Bloom Filter are able to solve the task with a storage set size of 500 at 1.1kb , 217b, and 382b; a 4.3×, 22×, and 12× saving respectively. One could remark, in this setting the Bloom Filter would perform much more strongly if it were given a hash function that received the class label instead of the image. The idea behind this task, is that it represents a setting where there exists an implicit class label that perhaps is not known. The Neural Bloom Filter can find this implicit label and exploit it to perform strongly with a small amount of space, where the Bloom Filter is fixed in its approach to solving the problem.

For the *non-uniform sampling task* in Figure 5.2b we see the Bloom Filter is preferable for less than 500 stored elements, but is overtaken thereafter. At 1000 elements the DNC, LSTM, and Neural Bloom Filter consume 7.9kb, 7.7kb, and 6.8kb respectively which corresponds to a 17.6%, 19.7%, and 28.6% reduction over the 9.6kb Bloom Filter. In the *uniform sampling task* shown in Figure 5.2c, there is no structure to the sampling of S.

The two architectures which rely on BPTT essentially fail to solve the task at some threshold of storage size. The Neural Bloom Filter solves it with 6.8kb (using a memory size of 50 and word size of 2). The overall conclusion from these sets of experiments is that the classical Bloom Filter works best when there is no structure to the data, however when there is (e.g. skewed data, or highly dependent sets that share common attributes) we do see significant space savings.

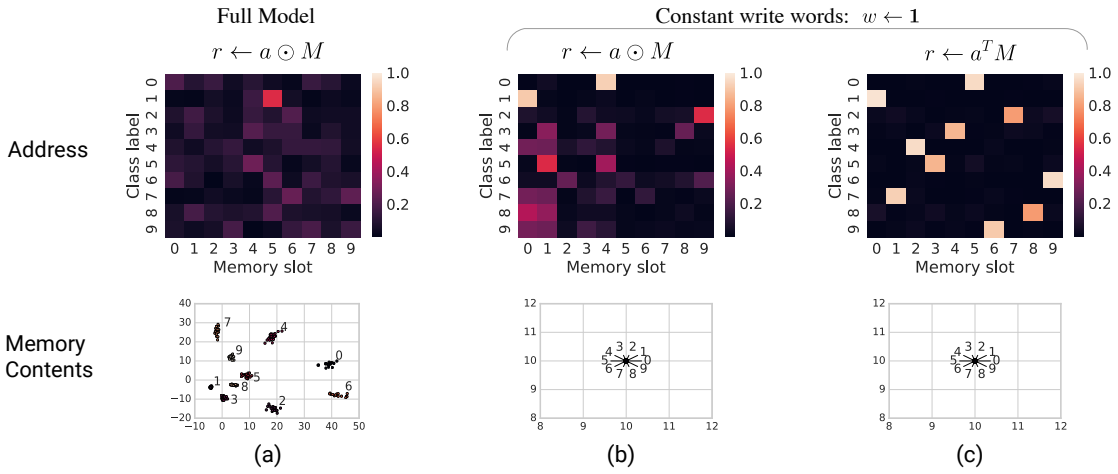


Figure 5.3: Memory access analysis. Three different learned solutions to class-based familiarity. We train three Neural Bloom Filter variants, with a succession of simplified read and write mechanisms. Each model contains 10 memory slots and the memory addressing weights a and contents $\bar{M}$ are visualised, broken down by class. Solutions share broad correspondence to known algorithms: (a) Bloom-g filters, (b) Bloom Filters, (c) Perfect hashing.

5.6.3 Memory Access Analysis

We wanted to understand how the Neural Bloom Filter uses its memory, and in particular how its learned solutions may correspond to classical algorithms. We inspected the memory contents (what was stored to memory) and addressing weights (where it was stored) for a small model of 10 memory slots and a word size of 2, trained on the MNIST class-based familiarity task.

We plot this for each class label, and compare the pattern of memory usage to two other models that use increasingly simpler read and write operations: (1) an ablated model with constant write words $w \leftarrow \mathbf{1}$, and (2) an ablated model with $w \leftarrow \mathbf{1}$ *and* a linear read operator $r \leftarrow a^T M$.

The full model, shown in Figure 5.3a learns to place some classes in particular slots, e.g. class 1 $\rightarrow$ slot 5, however most are distributed. Inspecting the memory

contents, it is clear the write word encodes a unique 2D token for each class. This solution bears resemblance with Bloom-g Filters (Qiao et al., 2011) where elements are spread across a smaller memory with the same hashing scheme as Bloom Filters, but a unique token is stored in each slot instead of a constant 1-bit value. With the model ablated to store only 1s in Figure 5.3b we see it uses semantic addressing codes for some classes (e.g. 0 and 1) and distributed addresses for other classes. E.g. for class 3 the model prefers to uniformly spread its writes across memory slot 1, 4, and 8.

The model solution is similar to that of Bloom Filters, with distributed addressing codes as a solution — but no information in the written words themselves. When we force the read operation to be linear in Figure 5.3c, the network maps each input class to a unique slot in memory. This solution has a correspondence with perfect hashing. In conclusion, with small changes to the read/write operations we see the Neural Bloom Filter learn different algorithmic solutions.

5.6.4 Database Queries

We look at a task inspired by database interactions. NoSQL databases, such as Bigtable and Cassandra, use a single string-valued row-key, which is used to index the data. The database is comprised of a union of files (e.g. SSTables) storing contiguous row-key chunks. Bloom Filters are used to determine whether a given query q lies within the stored set. We emulate this setup by constructing a universe of strings, that is alphabetically ordered, and by sampling contiguous ranges (to represent a given SSTable). Queries are sampled uniformly from the universe set of strings. We choose the $2.5M$ unique tokens in the GigaWord v5 news corpus to be

Table 5.1: Database task. Storing 5000 row-key strings for a target false positive rate.

	5%	1%	0.1%
Neural Bloom Filter	871b	1.5kb	24.5kb
Bloom Filter	31.2kb	47.9kb	72.2kb
Cuckoo Filter	33.1kb	45.3kb	62.6kb

our universe as this consists of structured natural data and some noisy or irregular strings. We consider the task of storing sorted string sets of size 5000. We train the Neural Bloom Filter to several desired false positive rates $(5\%, 1\%, 0.1\%)$ and used a backup Bloom Filter to guarantee 0% false negative rate. We also trained LSTMs and DNCs for comparison, but they failed to learn a solution to the task after several days of training; optimizing insertions via BPTT over a sequence of length 5000 did not result in a remotely usable solution. The Neural Bloom Filter avoids BPTT via its simple additive write scheme, and so it learned to solve the task quite naturally. As such, we compare the Neural Bloom Filter solely to classical data structures: Bloom Filters and Cuckoo Filters. In Table 5.1 we see a significant space reduction of $3 - 40\times$, where the margin grows with increasing permitted false positive rates. Since memory is an expensive component within production databases (in contrast to disk, say), this memory space saving could translate to a non-trivial cost reduction. We note that a storage size of 5000 may appear small, but is relevant to the NOSQL database scenario where disk files (e.g. SSTables) are typically sharded to be several megabytes in size, to avoid issues with compaction. E.g. if the stored values were of size 10kB per row, we would expect 5000 unique keys or less in an average Bigtable SSTable.

5.7 Database Extrapolation Task

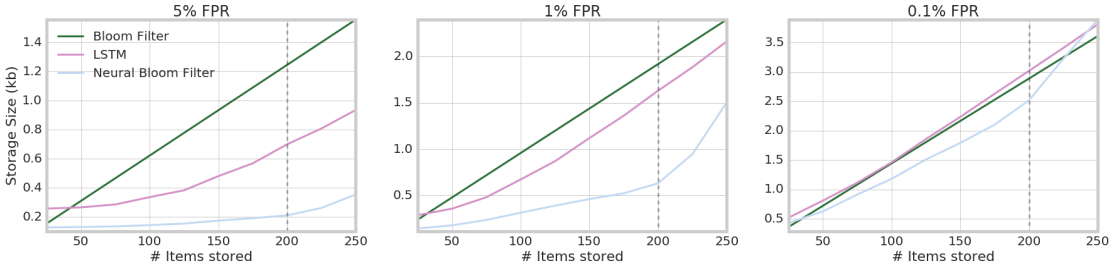


Figure 5.4: Database extrapolation task. Models are trained up to sets of size 200 (dashed line). We see extrapolation to larger set sizes on test set, but performance degrades. Neural architectures perform best for larger allowed false positive rates.

One further consideration for production deployment is the ability to extrapolate to larger storage set sizes during evaluation. Whilst a classical Bloom Filter has guarantees on performance as a function of storage set size, and thus can be trusted to extrapolate, the same does not hold true for neural networks. Crucially, larger sets can create out-of-distribution mismatch which degrades performance. investigate whether neural models are able to extrapolate to larger test sizes using the database task setup. Each set contains a contiguous set of sorted strings; we train both the Neural Bloom Filter and LSTM on sets of sizes 2 - 200. We then evaluate on sets up to 250, i.e. a 25% increase over what is observed during training. This is to emulate the scenario that we train on a selection of database tablets, but during evaluation we may observe some tablets that are slightly larger than those in the training set. We display the results in Figure 5.4.

Both the LSTM and Neural Bloom Filter are able to solve the task, with the Neural Bloom Filter using significantly less space for the larger allowed false positive rate of 5% and 1%. We do see the models' error increase as it surpasses the maximum training set size, however it is not catastrophic. Another interesting

Table 5.2: Latency for a single query, and throughput for a batch of 10,000 queries. *Query-efficient Bloom Filter from Chen et al. (2007), †Learned Index from Kraska et al. (2018).

	Query + Insert Latency		Query Throughput(/s)		Insert Throughput(/s)	
	CPU	GPU	CPU	GPU	CPU	GPU
Bloom Filter*	**0.02ms**	-	61K	-	61K	-
Neural Bloom Filter	5.1ms	13ms	3.5K	105K	3.2K	**101K**
LSTM	5.0ms	13ms	3.1K	**107K**	2.4K	4.6K
Learned Index†	780ms	1.36s	3.1K	**107K**	25	816

trend is noticeable; the neural models have higher utility for larger allowed false positive rates. This may be because of the difficulty in training the models to an extremely low accuracy.

5.7.1 Timing benchmark

We have principally focused on space comparisons, we now consider speed for the database task described in the prior section. We measure latency as the wall-clock time to complete a single insertion or query of a row-key string of length 64. We also measure throughput as the reciprocal wall-clock time of inserting or querying $10,000$ strings. We use a common encoder architecture for the neural models, a 128-hidden-unit character LSTM. We benchmark the models on the CPU (Intel(R) Xeon(R) CPU E5-1650 v2 @ 3.50GHz) and on the GPU (NVIDIA Quadro P6000) with models implemented in TensorFlow without any model-specific optimizations.

We compare to empirical timing results published in a query-optimized Bloom Filter variant (Chen et al., 2007). We include the Learned Index from Kraska et al. (2018) to contrast timings with a model that is not one-shot. The architecture is simply the LSTM character encoder; inserts are performed via gradient descent.

The number of gradient-descent steps to obtain convergence is domain-dependent, we chose 50 steps in our timing benchmarks. The Learned Index queries are obtained by running the character LSTM over the input and classifying familiarity — and thus query metrics are identical to the LSTM baseline.

We see in Table 5.2. that the combined query and insert latency of the Neural Bloom Filter and LSTM sits at 5ms on the CPU, around 400× slower than the classical Bloom Filter. The Learned Index contains a much larger latency of 780ms due to the sequential application of gradients. For all neural models, latency is not improved when operations are run on the GPU.

However for production databases serving thousands of queries per second, the batched performance is of more importance than individual latency. When multiple queries are received, the throughput of GPU-based neural models surpasses the classical Bloom Filter due to efficient concurrency of the dense linear algebra operations. This leads to the conclusion that a Neural Bloom Filter could be deployed in scenarios with high query load without a catastrophic decrease in throughput, if GPU devices are available. This is not to claim that Bloom Filters could not have an improved throughput on the GPU. The claim is that the CPU-based throughput for Bloom Filters is sufficient in all known database applications and thus a hardware-accelerated Neural Bloom Filter which matches this throughput could be feasible in the same setting.

For insertions we see a bigger separation between the one-shot models: the LSTM and Neural Bloom Filter. The Neural Bloom Filter surpasses the Bloom Filter's insertion throughput when placed on the GPU, with $101K$ insertions per second (IPS). The LSTM runs at $4.6K$ IPS, one order of magnitude slower, because writes

are serial, and the Learned Index structure is two orders of magnitude slower at 816 IPS due to sequential gradient computations. The benefits of the Neural Bloom Filter's simple write scheme are apparent here.

It is worth noting, in several Bloom Filter applications, the actual query latency is not in the critical path of computation. For example, for a distributed database, the network latency and disk access latency for one tablet can be orders of magnitude greater than the in-memory latency of a Bloom Filter query. For this reason, we have not made run-time a point of focus in this study, and it is implicitly assumed that the neural network is trading off greater latency for less space.

5.8 Related Work

There have been a large number of Bloom Filter variants published; from *Counting Bloom Filters* which support deletions (Fan et al., 2000), *Bloomier Filters* which store functions vs sets (Chazelle et al., 2004), *Compressed Bloom Filters* which use arithmetic encoding to compress the storage set (Mitzenmacher, 2002), and *Cuckoo Filters* which use cuckoo hashing to reduce redundancy within the storage vector (Fan et al., 2014). Although some of these variants focus on better compression, they do not achieve this by specializing to the data distribution.

One of the few works which address data-dependence are *Weighted Bloom Filters* (Bruck et al., 2006; Wang et al., 2015). They work by modulating the number of hash functions used to store or query each input, dependent on its storage and query frequency. This requires estimating a large number of separate storage and query frequencies. This approach can be useful for imbalanced data distributions,

such as the non-uniform instance-based MNIST familiarity task. However it cannot take advantage of dependent sets, such as the class-based MNIST familiarity task, or the database query task. We see the Neural Bloom Filter is more compressive in all settings.

Sterne (2012) proposes a neurally-inspired set membership data-structure that works by replacing the randomized hash functions with a randomly-wired computation graph of *OR* and *AND* gates. The false positive rate is controlled analytically by modulating the number of gates and the overall memory size. However there is no learning or specialization to the data with this setup. Bogacz and Brown (2003) investigates a learnable neural familiarity module, which serves as a biologically plausible model of familiarity mechanisms in the brain, namely within the perirhinal cortex. However this has not shown to be empirically effective at exact matching.

Kraska et al. (2018) consider the use of a neural network to classify the membership of queries to a fixed set S. Here the network itself is more akin to a perfect hashing setup where multiple epochs are required to find a succinct holistic representation of the set, which is embedded into the weights of the network. In their case this search is performed by gradient-based optimization. We emulate their experimental comparison approach but instead propose a memory architecture that represents the set as activations in memory, versus weights in a network.

Mitzenmacher (2018a) discusses the benefits and draw-backs of a learned Bloom Filter; distinguishing the empirical false positive rate over the distribution of sets S versus the conditional false positive rate of the model given a particular set S. In this chapter we focus on the empirical false positive rate because we wish

to exploit redundancy in the data and query distribution. Mitzenmacher (2018b) also considers an alternate way to combine classical and learned Bloom Filters by 'sandwiching' the learned model with pre-filter and post-filter classical Bloom Filters to further reduce space.

5.9 Efficient addressing

We discuss some implementation tricks that could be employed for a production system.

Firstly the original model description defines the addressing matrix A to be trainable. This ties the number of parameters in the network to the memory size. It may be preferable to train the model at a given memory size and evaluate for larger memory sizes. One way to achieve this is by allowing the addressing matrix A to be non-trainable. We experiment with this, allowing $A \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ to be a fixed sample of Gaussian random variables. We can think of these as point on a sphere in high dimensional space, the controller network must learn to organize inputs into separate buckets across the surface of the sphere.

To make the addressing more efficient for larger memory sizes, we experiment with sparsification of the addressing softmax by preserving only the top k components. We denote this sparse softmax $\sigma_k(\cdot)$. When using a sparse address, we find the network can fixate on a subset of rows. This observation is common to prior sparse addressing work (Shazeer et al., 2017). We find **sphering** the query vector, often dubbed whitening, remedies this. This is done by maintaining a moving average of the principal components of the query q. see Section 5.9.2 for an ablation). The

modified sparse architecture variant is illustrated in Algorithm 4.

Algorithm 4 Sparse Neural Bloom Filter

def sparse_controller(x):
    $z \leftarrow f_{enc}(x)$
    $s \leftarrow f_q(z)$ {Raw query word}
    $q \leftarrow moving\_zca(q)$ {Spherical query}
    $a \leftarrow \sigma_k(q^T A)$ {Sparse address}
    $w \leftarrow f_w(z)$

def sparse_write(x):
    $a, w \leftarrow$ sparse_controller(x)
    $M_{t+1}[a_{idx}] \leftarrow M_t[a_{idx}] + w a_{val}^T$

def sparse_read(x):
    $a, w, z \leftarrow$ sparse_controller(x)
    $r \leftarrow M[a_{idx}] \odot a_{val}$
    $o \leftarrow f_{out}([r, w, z])$

One can avoid the linear-time distance computation $q^T A$ in the addressing operation $\sigma_k(q^T A)$ by using an approximate k-nearest neighbour index, such as locality-sensitive hashing (Datar et al., 2004), to extract the nearest neighbours from A in $\mathcal{O}(\log m)$ time. The use of an approximate nearest neighbour index has been empirically considered for scaling memory-augmented neural networks in Chapter 3 and also in (Kaiser et al., 2017) however this was used for attention on M directly. As M is dynamic the knn requires frequent re-building as memories are stored or modified. This architecture is simpler — A is fixed and so the approximate knn can be built once.

To ensure the serialized size of the network (which can be shared across many memory instantiations) is independent of the number of slots in memory m we can avoid storing A. In the instance that it is not trainable, and is simply a

fixed sample of random variables that are generated from a deterministic random number generator — we can instead store a set of integer seeds that can be used to re-generate the rows of A. We can let the i-th seed c_i, say represented as a 16-bit integer, correspond to the set of 16 rows with indices $16i, 16i + 1, \ldots, 16i + 15$. If these rows need to be accessed, they can be regenerated on-the-fly by c_i. The total memory cost of A is thus m bits, where m is the number of memory slots. If there are more than one million slots, the same argument applies by replacing 16 with 32.

Putting these two together it is possible to query and write to a Neural Bloom Filter with m memory slots in $\mathcal{O}(\log m)$ time, where the network consumes $\mathcal{O}(1)$ space. It is worth noting, however, the Neural Bloom Filter's memory is often much smaller than the corresponding classical Bloom Filter's memory, and in many of our experiments is even smaller than the number of unique elements to store. Thus dense matrix multiplication can still be preferable - especially due to its acceleration on GPUs and TPUs (Jouppi et al., 2017) - and a dense representation of A is not inhibitory. As model optimization can become application-specific, we do not focus on these implementation details and use the model in its simplest setting with dense matrix operations.

5.9.1 Moving ZCA

We found sphering the query helped avoid mode collapse where the same $\mathcal{O}(1)$ slots were always written to and read from. This was done by efficiently computing the ZCA transform (Bell and Sejnowski, 1996) in an online setting. ZCA was computed by taking moving averages over the first and second moments re-

spectively, calculating the ZCA matrix and updating a moving average projection matrix θ_{zca}. This is only done during training, at evaluation time θ_{zca} is fixed. We describe the update below for completeness.

$$\text{Input: } s \leftarrow f_q(z) \tag{5.1}$$

$$\mu_{t+1} \leftarrow \gamma\mu_t + (1-\gamma)\bar{s} \qquad \text{1st moment EMA} \tag{5.2}$$

$$\Sigma_{t+1} \leftarrow \gamma\Sigma_t + (1-\gamma)\ s^T s \qquad \text{2nd moment EMA} \tag{5.3}$$

$$U, s, \_ \leftarrow \texttt{svd}(\Sigma - \mu^2) \qquad \text{Singular values} \tag{5.4}$$

$$W \leftarrow UU^T/\sqrt{(}s) \qquad \text{ZCA matrix} \tag{5.5}$$

$$\theta_{zca} \leftarrow \eta\theta_{zca} + (1-\eta)W \qquad \text{ZCA EMA} \tag{5.6}$$

$$q \leftarrow s\ \theta_{zca} \qquad \text{Projected query} \tag{5.7}$$

In practice we do not compute the singular value decomposition at each time step to save computational resources, but instead calculate it and update θ every T steps. We scale the discount in this case $\eta' = \eta/T$.

5.9.2 Query Sphering

We see the benefit of sphering sparse addressing mechanisms in Figure 5.5 where the converged validation performance ends up at a higher state. Investigating the proportion of memory filled after all elements have been written in Figure 5.6, we see the model uses quite a small proportion of its memory slots. This is likely due to the network fixating on rows it has accessed with sparse addressing, and ignoring rows it has otherwise never touched — a phenomena noted in Shazeer et al. (2017). The model finds a local minima in continually storing and accessing

the same rows in memory. The effect of sphering is that the query now appears to be Gaussian (up to the first two moments) and so the nearest neighbour in the address matrix A (which is initialized to Gaussian random variables) will be close to uniform. This results in a more uniform memory access (as seen in Figure 5.6) which significantly aids performance (as seen in Figure 5.5).

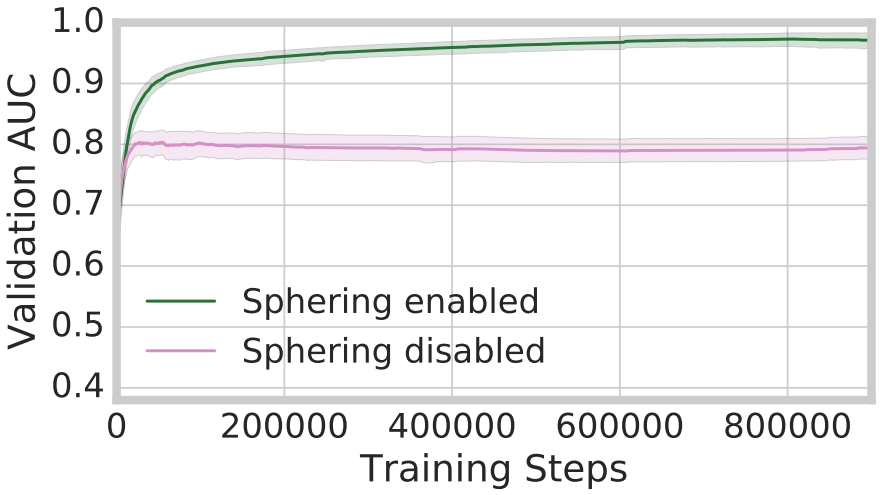


Figure 5.5: For sparse addresses, sphering enables the model to learn the task of set membership to high accuracy.

5.10 Conclusion

In many situations neural networks are not a suitable replacement to Bloom Filters and their variants. The Bloom Filter is robust to changes in data distribution because it delivers a bounded false positive rate for any sampled subset. However in this chapter we consider the questions, "When might a single-shot neural network provide better compression than a Bloom Filter?". We see that a model which uses an external memory with an adaptable capacity, avoids BPTT with a feed-forward write scheme, and learns to address its memory, is the most promising option in

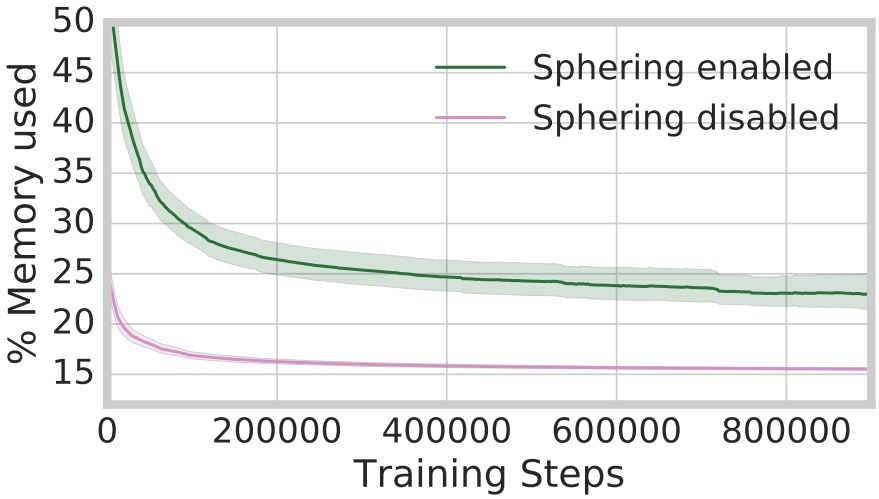


Figure 5.6: For sparse addresses, sphering the query vector leads to fewer collisions across memory slots and thus a higher utilization of memory.

contrast to popular memory models such as DNCs and LSTMs. We term this model the Neural Bloom Filter due to the analogous incorporation of a hashing scheme, commutative write scheme, and multiplicative read mechanism.

The Neural Bloom Filter relies on settings where we have an off-line dataset (both of stored elements and queries) that we can meta-learn over. In the case of a large database we think this is warranted, a database with 100K separate set membership data structures will benefit from a single (or periodic) meta-learning training routine that can run on a single machine and sample from the currently stored data, generating a large number of efficient data-structures. We envisage the space cost of the network to be amortized by sharing it across many neural Bloom Filters, and the time-cost of executing the network to be offset by the continuous acceleration of dense linear algebra on modern hardware, and the ability to batch writes and queries efficiently. A promising future direction would be to investigate the feasibility of this approach in a production system.

Chapter 6

Compressive Memory for Sequence Modelling

We present the Compressive Transformer, an attentive sequence model which compresses past memories for long-range sequence learning. We find the Compressive Transformer obtains state-of-the-art language modelling results in the WikiText-103 and Enwik8 benchmarks, achieving 17.1 ppl and 0.97 bpc respectively. We also find it can model high-frequency speech effectively and can be used as a memory mechanism for RL, demonstrated on an object matching task. To promote the domain of long-range sequence learning, we propose a new open-vocabulary language modelling benchmark derived from books, PG-19.

6.1 Motivation

Humans have a remarkable ability to remember information over long time horizons. When reading a book, we build up a compressed representation of the past narrative, such as the characters and events that have built up the story so far. We can do this even if they are separated by thousands of words from the current text, or long stretches of time between readings. During daily life, we make use of memories at varying time-scales: from locating the car keys, placed in the morning, to recalling the name of an old friend from decades ago. These feats of memorisation are not achieved by storing every sensory glimpse throughout one's lifetime, but via lossy compression. We aggressively select, filter, or integrate input stimuli based on factors of surprise, perceived danger, or repetition — amongst other signals (Richards and Frankland, 2017).

Memory systems in artificial neural networks began with very compact representations of the past. Recurrent neural networks (RNNs, Rumelhart et al. (1986)) learn to represent the history of observations in a compressed state vector. The state is *compressed* because it uses far less space than the history of observations — the model only preserving information that is pertinent to the optimisation of the loss. The LSTM (Hochreiter and Schmidhuber, 1997) is perhaps the most ubiquitous RNN variant; it uses learned gates on its state vector to determine what information is stored or forgotten from memory.

However since the LSTM, there has been great benefit discovered in *not* bottlenecking all historical information in the state, but instead in keeping past activations around in an external memory and *attending* to them. The Transformer (Vaswani et al., 2017) is a sequence model which stores the hidden activation of every time-

step, and integrates this information using an attention operator (Bahdanau et al., 2014). The Transformer will thus represent the past with a tensor (depth $\times$ memory size $\times$ dimension) of past observations that is, in practice, an order of magnitude larger than an LSTM's hidden state. With this granular memory, the Transformer has brought about a step-change in state-of-the-art performance, within machine translation (Vaswani et al., 2017), language modelling (Dai et al., 2019; Shoeybi et al., 2019), video captioning (Zhou et al., 2018b), and a multitude of language understanding benchmarks (Devlin et al., 2019; Yang et al., 2019) amongst others.

One drawback in storing everything is the computational cost of attending to every time-step and the storage cost of preserving this large memory. Sparse attention mechanisms, such as SAM presented in Chapter 3 address this, along with more recent contemporary works applying sparsity to transformers (Child et al., 2019; Sukhbaatar et al., 2019; Lample et al., 2019). However sparse attention does not solve the storage problem, and often requires custom sparse kernels for efficient implementation. Instead we look back to the notion of compactly representing the past. We show this can be built with simple dense linear-algebra components, such as convolutions, and can reduce both the space and compute cost of our models.

We propose the Compressive Transformer, a simple extension to the Transformer which maps past hidden activations (memories) to a smaller set of compressed representations (compressed memories). The Compressive Transformer uses the same attention mechanism over its set of memories and compressed memories, learning to query both its short-term granular memory and longer-term coarse memory.

We observe this improves the modelling of text, achieving state-of-the-art results in character-based language modelling — 0.97 bpc on Enwik8 from the Hutter Prize (Hutter, 2012) — and word-level language modelling — 17.1 perplexity on WikiText-103 (Merity et al., 2016). Specifically, we see the Compressive Transformer improves the modelling of rare words.

We show the Compressive Transformer works not only for language, but can also model the waveform of high-frequency speech with a trend of lower likelihood than the TransformerXL and Wavenet (Oord et al., 2016) when trained over 400,000 steps. We also show the Compressive Transformer can be used as a memory component within an RL agent, IMPALA (Espeholt et al., 2018), and can successfully compress and make use of past observations.

Furthermore we present a new book-level language-modelling benchmark PG-19, extracted from texts in Project Gutenberg,[1] to further promote the direction of long-context sequence modelling. This is over double the size of existing LM benchmarks and contains text with much longer contexts.

6.2 Related Work

For character-level neural machine translation, Lee et al. (2017) propose a hybrid model containing compressed long-range memory of the source sentence, and granular short-range memory. Namely the source sentence is encoded at the character-level using a stack of convolutional neural networks with pooling, to temporally compress the input from characters to segments, followed by an

[1] `https://www.gutenberg.org/`

attention-augmented GRU. The authors find this setup outperforms translation over byte-pair-encoded BPE text, where BPE typically reduces the sequence length by 4x. These experiments were confined to relatively short-range sentence-level translation, however they show the promise in learned temporal compression for modelling natural language.

There have been a variety of recent attempts to extend the range of attention with Transformer models, or to replace the attention operation with something less expensive. Wu et al. (2019) show that a convolution-like operator that runs in linear time can actually exceed the performance of the quadratic-time self-attention layer in the Transformer at sentence-to-sentence translation and sentence-level language modelling. However such a mechanism inhibits the flow of information across a large number of time-steps for a given layer, and has not shown to be beneficial for long-range sequence modelling.

Dai et al. (2019) propose the TransformerXL, which keeps past activations around in memory. They also propose a novel relative positional embedding scheme which they see outperforms the Transformer's original absolute positional system. Our model incorporates both of these ideas, the use of a memory to preserve prior activations and their relative positional embedding scheme.

The Sparse Transformer (Child et al., 2019) uses fixed sparse attention masks to attend to roughly $\sqrt{n}$ locations in memory. This approach still requires keeping all memories around during training, however with careful re-materialization of activations and custom kernels, the authors are able to train the model with a reasonable budget of memory and compute. When run on Enwik8, the much larger attention window of $8,000$ improves model performance, but overall it does not

significantly outperform a simpler TransformerXL with a much smaller attention window.

The use of dynamic attention spans is explored in Sukhbaatar et al. (2019). Different attention heads can learn to have shorter or longer spans of attention — and they observe this achieves state-of-the-art in character-based language modelling. This idea could easily be combined with our contribution — a compressive memory. However an efficient implementation is not possible on current dense-linear-algebra accelerators, such as Google's TPUs, due to the need for dynamic and sparse computation. Our approach builds on simple dense linear algebra components, such as convolutions.

The most similar model which has been published slightly after this study is the Funnel Transformer (Dai et al., 2020) which also considers temporal compression via convolutions, pooling, and maximum-attention selection in a sequence-to-sequence setup. This specifically compresses information in the encoder of an encoder-decoder transformer for tasks such as machine translation. The key difference is the encoder processes the whole input bi-directionally, however we consider the case of auto-regressive modelling where causality must be preserved. We can think of the Funnel Transformer as an instance of the Compressive Transformer if we applied the encoder to past memories, and the decoder to the current sequence of activations to be modelled.

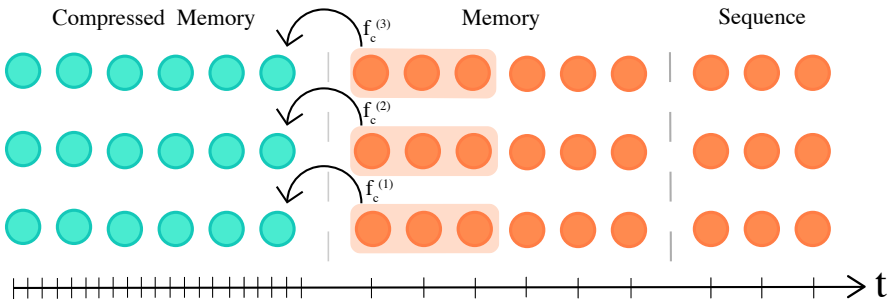


Figure 6.1: The Compressive Transformer keeps a fine-grained memory of past activations, which are then compressed into coarser *compressed* memories. The above model has three layers, a sequence length $n_s = 3$, memory size $n_m = 6$, compressed memory size $n_{cm} = 6$. The highlighted memories are compacted, with a compression function f_c per layer, to a single compressed memory — instead of being discarded at the next sequence. In this example, the rate of compression $c = 3$.

6.3 Model

We present the Compressive Transformer, a long-range sequence model which compacts past activations into a compressed memory. The Compressive Transformer is a variant of the Transformer (Vaswani et al., 2017), a deep residual network which only uses attention to propagate information over time (namely *multi-head attention*). We build on the ideas of the TransformerXL (Dai et al., 2019) which maintains a memory of past activations at each layer to preserve a longer history of context. The TransformerXL discards past activations when they become sufficiently old (controlled by the size of the memory). The key principle of the Compressive Transformer is to compress these old memories, instead of discarding them, and store them in an additional *compressed memory*.

6.3.1 Description

We define n_m and n_{cm} to be the number of respective memory and compressive memory slots in the model per layer. The overall input sequence $\mathcal{S} = x_1, x_2, \ldots, x_{|s|}$ represents input observations (e.g. tokens from a book). These are split into fixed-size windows of size n_s for the model to process in parallel. The model observes $\mathbf{x} = x_t, \ldots, x_{t+n_s}$ at time t, which we refer to as the *sequence* (e.g. in Figure 6.1). As the model moves to the next sequence, its n_s hidden activations are pushed into a fixed-sized FIFO memory (like the TransformerXL). The oldest n_s activations in memory are evicted, but unlike the TransformerXL we do not discard them. Instead we apply a *compression operation*, $f_c : \mathbf{R}^{n_s \times d} \rightarrow \mathbf{R}^{\lfloor \frac{n_s}{c} \rfloor \times d}$, mapping the n_s oldest memories to $\lfloor \frac{n_s}{c} \rfloor$ compressed memories which we then store in a secondary FIFO *compressed memory*. d denotes the hidden size of activations and c refers to the compression rate, a higher value indicates more coarse-grained compressed memories. The full architecture is described in Algorithm 5.

6.3.2 Compression Functions and Losses

For choices of compression functions f_c we consider **(1) max/mean pooling**, where the kernel and stride is set to the compression rate c; **(2) 1D convolution** also with kernel & stride set to c; **(3) dilated convolutions**; **(4) most-used** where the memories are sorted by their average attention (usage) and the most-used are preserved. The pooling is used as a fast and simple baseline. The *most-used* compression scheme is inspired from the garbage collection mechanism in the Differentiable Neural Computer (Graves et al., 2016) where low-usage memories are erased. The convolutional compression functions contain parameters which

Algorithm 5 Compressive Transformer

At time zero

$\mathbf{m_0} \leftarrow \mathbf{0}$ {Initialise memory to zeros $(l \times n_m \times d)$}
$\mathbf{cm_0} \leftarrow \mathbf{0}$ {Initialise compressed memory to zeros $(l \times n_{cm} \times d)$}

At time t

$\mathbf{h}^{(1)} \leftarrow \mathbf{x}\mathbf{W_{emb}}$ {Embed input sequence$(n_s \times d)$}
for layer $i = 1, 2, \ldots, l$ **do**
  $\mathbf{mem^{(i)}} \leftarrow \text{concat}(\mathbf{cm_t^{(i)}}, \mathbf{m_t^{(i)}})$ {$((n_{cm} + n_m) \times d)$}
  $\tilde{\mathbf{a}}^{(\mathbf{i})} \leftarrow \text{multihead\_attention}^{(i)}(\mathbf{h^{(i)}}, \mathbf{mem_t^{(i)}})$ {MHA over both mem types} $(n_s \times d)$
  $\mathbf{a^{(i)}} \leftarrow \text{layer\_norm}(\tilde{\mathbf{a}}^{(\mathbf{i})} + \mathbf{h^{(i)}})$ {Regular skip + layernorm $(n_{cm} \times d)$}
  $\mathbf{old\_mem^{(i)}} \leftarrow \mathbf{m_t^{(i)}}[: n_s]$ {Oldest memories to be forgotten $(n_s \times d)$}
  $\mathbf{new\_cm^{(i)}} \leftarrow f_c^{(i)}(\mathbf{old\_mem^{(i)}})$ {Compress oldest memories by factor c $(\lfloor \frac{n_s}{c} \rfloor \times d)$}
  $\mathbf{m_{t+1}^{(i)}} \leftarrow \text{concat}(\mathbf{m_t^{(i)}}, \mathbf{h^{(i)}})[-n_m :]$ {Update memory $(n_m \times d)$}
  $\mathbf{cm_t^{(i)}} \leftarrow \text{concat}(\mathbf{cm_t^{(i)}}, \mathbf{new\_cm^{(i)}})[-n_{cm} :]$ {Update compressed memory $(n_{cm} \times d)$}
  $\mathbf{h^{(i+1)}} \leftarrow \text{layer\_norm}(\text{mlp}^{(i)}(\mathbf{a^{(i)}}) + \mathbf{a^{(i)}})$ {Mixing MLP $(n_s \times d)$}
end for

Algorithm 6 Attention-Reconstruction Loss

$L^{attn} \leftarrow 0$
for layer $i = 1, 2, \ldots, l$ **do**
  $\mathbf{h^{(i)}} \leftarrow \text{stop\_gradient}(\mathbf{h^{(i)}})$ {Stop compression grads from passing... }
  $\mathbf{old\_mem^{(i)}} \leftarrow \text{stop\_gradient}(\mathbf{old\_mem^{(i)}})$ {...into transformer network.}
  $\mathbf{Q}, \mathbf{K}, \mathbf{V} \leftarrow \text{stop\_gradient}(\text{attention params at layer i})$ {Re-use attention weight matrices.}
  def $\text{attn}(\mathbf{h}, \mathbf{m}) \leftarrow \text{sigm}((\mathbf{hQ})\,(\mathbf{mK}))(\mathbf{mV})$ {Use content-based attention (no relative).}
  $\mathbf{new\_cm^{(i)}} \leftarrow f_c^{(i)}(\mathbf{old\_mem^{(i)}})$ {Compression network (to be optimised).}
  $L^{attn} \leftarrow L^{attn} + ||\text{attn}(\mathbf{h^{(i)}}, \mathbf{old\_mem^{(i)}}) - \text{attn}(\mathbf{h^{(i)}}, \mathbf{new\_cm^{(i)}})||_2$
end for

require training.

One can train the compression network using gradients from the loss; however for very old memories this requires backpropagating-through-time (**BPTT**) over long unrolls. As such we also consider some local auxiliary compression losses. We consider an **auto-encoding** loss where we reconstruct the original memories from the compressed memories $\mathcal{L}^{ae} = ||\mathbf{old\_mem^{(i)}} - g(\mathbf{new\_cm^{(i)}})||_2$, where $g : \mathbb{R}^{\frac{n_s}{c} \times d} \rightarrow \mathbb{R}^{n_s \times d}$ is learned. This is a lossless compression objective — it attempts to retain all information in memory. We also consider an **attention-reconstruction** loss described in Algorithm 6 which reconstructs the content-based attention over memory, with content-based attention over the compressed memories. This is a lossy objective, as information that is no longer attended to can be discarded, and we found this worked best. We stop compression loss gradients from passing into the main network as this prevents learning. Instead the Transformer optimises the task objective and the compression network optimises the compression objective conditioned on task-relevant representations; there is no need to mix the losses with a tuning constant.

6.3.3 Temporal Range

The TransformerXL with a memory of size n has a maximum temporal range of $l \times n$ with an attention cost of $\mathcal{O}(n_s^2 + n_s n)$ (see Dai et al. (2019) for a detailed discussion). The Compressive Transformer now has a maximum temporal range of $l \times (n_m + c * n_{cm})$ with an attention cost of $\mathcal{O}(n_s^2 + n_s(n_m + n_{cm}))$. For example, setting $n_{cm} = n_m = n/2$ and $c = 3$ we obtain a maximum temporal range that is two times greater than the TransformerXL with an identical attention cost. Thus

if we can learn in the $c > 1$ compressed setting, the temporal range of the model can be significantly increased.

6.4 PG-19 Benchmark

As models begin to incorporate longer-range memories, it is important to train and benchmark them on data containing larger contexts. Natural language in the form of text provides us with a vast repository of data containing long-range dependencies, that is easily accessible. We propose a new language modelling benchmark, **PG-19**, using text from books extracted from Project Gutenberg.[2] We select Project Gutenberg books which were published over 100 years old, i.e. before 1919 (hence the name PG-19) to avoid complications with international copyright, and remove short texts. The dataset contains $28,752$ books, or $11GB$ of text — which makes it over double the size of BookCorpus and Billion Word Benchmark.

6.4.1 Preprocessing

The raw texts from the Gutenberg project were minimally pre-processed by removing boilerplate license text. We then also replaced discriminatory words with a unique ⟨DWx⟩ token using the Ofcom list of discriminatory words. [3]

[2] `https://github.com/deepmind/pg19`

[3] `https://www.ofcom.org.uk/__data/assets/pdf_file/0023/91625/OfcomQRG-AOC.pdf`

6.4.2 Related Datasets

The two most benchmarked word-level language modelling datasets either stress the modelling of stand-alone sentences (Billion Word Benchmark from Chelba et al. (2013)) or the modelling of a small selection of short news articles (Penn Treebank processed by Mikolov et al. (2010)). Merity et al. (2016) proposed the WikiText-103 dataset, which contains text from a high quality subset of English-language wikipedia articles. These articles are on average $3,600$ words long. This dataset has been a popular recent LM benchmark due to the potential to exploit longer-range dependencies in text, such as those explored in Chapter 4 and contemporary works (Grave et al., 2016b; Bai et al., 2018b). However recent Transformer models, such as the TransformerXL (Dai et al., 2019) appear to be able to exploit temporal dependencies on the order of several thousand words. This motivates a larger dataset with longer contexts.

Books are a natural choice of long-form text, and provide us with stylistically rich and varied natural language. Texts extracted from books have been used for prior NLP benchmarks; such as the Children's Book Test (Hill et al., 2015) and LAMBADA (Paperno et al., 2016). These benchmarks use text from Project Gutenberg, an online repository of books with expired US copyright, and BookCorpus (Zhu et al., 2015), a prior dataset of $11K$ unpublished (at time of authorship) books. CBT and LAMBADA contain extracts from books, with a specific task of predicting held-out words. In the case of LAMBADA the held-out word is specifically designed to be predictable for humans with access to the full textual context — but difficult to guess with only a local context.

CBT and LAMBADA are useful for probing the linguistic intelligence of models,

Table 6.1: Comparison to existing popular language modelling benchmarks.

	Avg. length (words)	**Train Size**	**Vocab**	**Type**
1B Word	27	4.15GB	793K	News (sentences)
Penn Treebank	355	5.1MB	10K	News (articles)
WikiText-103	3.6K	515MB	267K	Wikipedia (articles)
PG-19	69K	10.9GB	(open)	Books

but are not ideal for training long-range language models from scratch as they truncate text extracts to at most a couple of paragraphs, and discard a lot of the books' text. There has been prior work on training models on book data using BookCorpus directly (e.g. BERT from Devlin et al. (2019)) however BookCorpus is no longer distributed due to licensing issues, and the source of data is dynamically changing — which makes exact benchmarking difficult over time.

The NarrativeQA Book Comprehension Task (Kočiskỳ et al., 2018) uses Project Gutenberg texts paired with Wikipedia articles, which can be used as summaries. Due to the requirement of needing a corresponding summary, NarrativeQA contains a smaller selection of books: 1,527 versus the 28,752 books in PG-19. However it is reasonable that PG-19 may be useful for pre-training book summarisation models.

6.4.3 Statistics

A brief comparison of PG-19 to other LM datasets can be found in Table 6.1. We intentionally do not limit the vocabulary by *unk-ing* rare words, and release the dataset as an open-vocabulary benchmark. To compare models we propose to continue measuring the word-level perplexity. This can still be computed for any chosen character-based, byte-based or subword-based scheme. To do this, one

calculates the total cross-entropy loss $L = -\sum_t \log(p_t|p_{<t})$ over the given validation or test subset using a chosen tokenisation scheme, and then one normalises this value by the number of words: L/n_{words} where n_{words} is the total number of words in the given subset, taken from Table 6.3. The word-level perplexity is thus $e^{L/n_{words}}$.

For sake of model comparisons, it is important to use the exact number of words computed in Table 6.3 as the normalisation constant. It's worth noting only open-vocabulary approaches should be compared, that is it would be considered cheating on this benchmark to restrict the support of the model to include only the train and test set. It is also worth noting most open-vocabulary approaches, such as BPE, use a greedy segmentation of the text and thus the model's log likelihood is actually an under-estimate of the true likelihood that would be obtained by marginalising over all segmentations. Nevertheless comparing lower bounds is still a valid methodology, and is common in the vision literature for Variational Auto-Encoders, for example, where exact likelihoods cannot be computed.

6.4.4 Topics

We present top-words for some of the topics on the PG-19 corpus. These were generated with LDA topic model (Blei et al., 2003). These topics include art, education, naval exploration, geographical description, war, ancient civilisations, and more poetic topics concerning the human condition — love, society, religion, virtue etc. This contrasts to the more objective domains of Wikipedia and news corpora.

Table 6.2: Examples of top topics on PG-19 corpus.

Geography	War	Civilisations	Human Condition	Naval	Education	Art
water	people	roman	love	island	work	poet
river	emperor	rome	religion	ship	school	music
feet	war	greek	religious	sea	life	one
miles	army	city	life	men	children	poetry
north	death	gods	moral	captain	may	work
south	battle	king	human	coast	social	literature
mountains	city	first	society	land	child	art
sea	soldiers	caesar	man	great	education	great
lake	power	great	virtue	found	conditions	poem
rock	thousand	romans	nature	islands	well	written
mountain	arms	athens	marriage	shore	study	english
country	empire	greece	women	voyage	best	author
valley	upon	temple	christian	vessels	years	play
ice	country	son	age	time	possible	genius
west	time	egypt	law	english	class	style

6.5 Experiments

We optimised all models with Adam (Kingma and Ba, 2014). We used a learning rate schedule with a linear warmup from 1e-6 to 3e-4 and a cosine decay back down to 1e-6. For character-based LM we used $4,000$ warmup steps with $100,000$ decay steps, and for word-based LM we used $16,000$ warmup steps with $500,000$ decay steps. We found that decreasing the optimisation update frequency helped (see Section 6.5.5). We only applied parameter updates every 4 steps after $60,000$ iterations. However we found the models would optimise well for a range of warmup/warm-down values. We clipped the gradients to have a norm of at most 0.1, which was crucial to successful optimisation.

6.5.1 PG-19

We benchmark the Compressive Transformer against the TransformerXL on the newly proposed PG-19 books dataset. Because it is open-vocabulary, we train a subword vocabulary of size 32000 with SubwordTextEncoder from TensorFlow and

Table 6.3: PG-19 statistics split by subsets.

	Train	**Valid.**	**Test**
# books	28,602	50	100
# words	1,973,136,207	3,007,061	6,966,499

Table 6.4: Text perplexity on PG-19.

	Valid.	Test
36L TransformerXL	45.5	36.3
36L Compressive Transformer	43.4	33.6

use the dataset statistics to compute word-level perplexity, as described in Section 6.4.3. We train a 36 layer Compressive Transformer with a window size of 512, both memory and compressed memory size of 512, and compression rate $C = 2$. We compare this to a 36 layer TransformerXL trained with window size 512 and attention window 1024. The model was trained on 256 TPUv3 cores with a total batch size of 512 and converged after processing around 100 billion subword tokens. We display the results in Table 6.4 where we see the Compressive Transformer obtains a test perplexity of 33.6 versus the TransformerXL's 36.3. Despite the dataset size, it is clearly a challenging domain. This can suit as a first baseline on the proposed long-range language modelling benchmark. We show samples from this model in Section 6.5.6. The model is able to generate long-form samples of varying styles: character dialogue, first person diary entries, and third-person narrative.

Table 6.5: State-of-the-art results on Enwik8.

Model	**Bits per Character**
7L LSTM (Graves, 2013)	1.67
HyperNetworks Ha et al. (2016)	1.34
HM-LSTM Chung et al. (2016)	1.32
ByteNet (Kalchbrenner et al., 2016)	1.31
RHN Zilly et al. (2017)	1.27
mLSTM Krause et al. (2016)	1.24
64L Transformer Al-Rfou et al. (2019)	1.06
24L TransformerXL (Dai et al., 2019)	0.99
Sparse Transformer (Child et al., 2019)	0.991
Adaptive Transformer (Sukhbaatar et al., 2019)	0.98
24L TransformerXL (ours)	0.98
24L Compressive Transformer	**0.97**

6.5.2 Enwik8

We compare the TransformerXL and the Compressive Transformer on the standard character-level language modelling benchmark Enwik8 taken from the Hutter Prize (Hutter, 2012), which contains 100M bytes of unprocessed Wikipedia text. We select the first 90MB for training, 5MB for validation, and the latter 5MB for testing — as per convention. We train 24-layer models with a sequence window size of 768. During training, we set the TransformerXL's memory size to 2304, and for the Compressive Transformer we use memory of size 768 and compressed memory of size 1152 with compression rate $C = 3$. During evaluation, we increased the TransformerXL memory size to 4096 and the compressed memory in our model to 3072 (after sweeping over the validation set), obtaining the numbers reported in Table 6.5. The proposed model achieves the new state-of-the-art on this dataset with 0.97 bits-per-character.

Memory Size

We compare the best test perplexity obtained for the Compressive Transformer trained on WikiText-103 and Enwik8 across a range of compressed memory sizes. For both models, the best model used a 1D convolution compression network with a compression rate of 3. The Enwik8 model was trained with an embedding size of 1024, 8 attention heads, 24 layers, an mlp hidden size of 3072, a sequence window size of 768, and a memory size of 768. We see the best compressed memory size is $3,072$ in this sweep, facilitating a total attention window of 3840. The WikiText-103 model was trained with an embedding size of 1024, adaptive inputs using the same parameters as (Sukhbaatar et al., 2019), 16 attention heads, 18 layers, an mlp hidden size of 4096, a sequence window of size 512 and a memory of size 512. The best compressed memory size is 1536 resulting in a total attention window of c. 2048.

Compressed Memory Size	512	1024	2048	3072	4096
Enwik8 BPC	1.01	0.99	0.98	0.97	1.00

Table 6.6: Compressed memory size vs test performance for Enwik8

Compressed Memory Size	256	512	1024	1536	2048
WikiText-103 Perplexity	18.2	17.9	17.6	17.1	17.7

Table 6.7: Compressed memory size vs test performance for WikiText-103

Compression Functions

We compare compression functions and the use of auxiliary losses in Table 6.8. We sweep over compression rates of 2, 3, and 4 and report results with the best performing value for each row. BPTT signifies that no auxiliary compression loss

Table 6.8: Compression approaches on Enwik8.

Compression fn	**Compression loss**	**BPC**
Convolution	BPTT	0.996
Max Pooling	N/A	0.986
Convolution	Auto-encoding	0.984
Mean Pooling	N/A	0.982
Most-used	N/A	0.980
Dilated convolution	Attention	0.977
Convolution	Attention	**0.973**

was used to train the network other than the overall training loss. To feed gradients into the compression function we unrolled the model over double the sequence length and halved the batch size to fit the larger unroll into memory.

6.5.3 Wikitext-103

We train an eighteen-layered Compressive Transformer on the closed-vocabulary word-level language modelling benchmark WikiText-103, which contains articles from Wikipedia. We train the model with a compressed memory size, memory size, and a sequence window size all equal to 512. We trained the model over 64 Tensor Processing Units (TPU) v3 with a batch size of 2 per core — making for a total batch size of 128. The model converged in a little over 12 hours. We found the single-layer convolution worked best, with a compression rate of $c = 4$. This model obtained 17.6 perplexity on the test set. By tuning the memory size over the validation set — setting the memory size to 500, and compressed memory size to $1,500$ — we obtain 17.1 perplexity. This is 1.2 perplexity points over prior state of the art, and means the model places a $\approx 5\%$ higher probability on the correct word over the prior SotA TransformerXL. Whilst 5% may seem like a small improvement which may not be statistically significant, it is worth noting that the

language models trained with different initialisation seeds and different shuffled orderings of the training data obtain the same best perplexity to a single decimal place. Thus these experiments are highly reproducible and the difference is not due to noise.

It is worth noting that in Table 6.9 we do not list methods that use additional training data, or that make use of test-time labels to continue training the model on the test set (known as dynamic evaluation (Graves, 2013)). If we incorporate a very naive dynamic evaluation approach of loading a model checkpoint and continuing training over one epoch of the test set, then we obtain a test perplexity of **16.1**. This is slightly better than the published 16.4 from Krause et al. (2019) — which uses a more sophisticated dynamic evaluation approach on top of the TransformerXL. However in most settings, one does not have access to test-time labels — and thus we do not focus on this setting. Furthermore there has been great progress in showing that more data equates to much better language modelling; Shoeybi et al. (2019) find a large transformer 8B-parameter transformer trained on 170GB of text obtains 10.7 word-level perplexity on WikiText-103. However it is not clear to what extent the WikiText-103 test set may be leaked inside these larger training corpora. For clarity of model comparisons, we compare to published results trained on the WikiText-103 training set. Certainly the direction of larger scale and more data appear to bring immediate gains to the quality of existing language models. Both data scale and quality alongside intelligent model design are complementary lines of research towards better sequence modelling.

We break perplexity down by word frequency in Table 6.10 and see the Compressive Transformer makes only a small modelling improvement for frequent words (2.6%

Table 6.9: Validation and test perplexities on WikiText-103.

	Valid.	Test
LSTM (Graves et al., 2014)	-	48.7
Temporal CNN (Bai et al., 2018a)	-	45.2
GCNN-14 (Dauphin et al., 2016)	-	37.2
Quasi-RNN (Bradbury et al., 2016)	32	33
RMC (Santoro et al., 2018)	30.8	31.9
LSTM+Hebb. (Rae et al., 2018)	29.0	29.2
Transformer (Baevski and Auli, 2019)	-	18.7
18L TransformerXL, M=384 (Dai et al., 2019)	-	18.3
18L TransformerXL, M=1024 (ours)	-	18.1
18L Compressive Transformer, M=1024	**16.0**	**17.1**

Table 6.10: WikiText-103 test perplexity broken down by word frequency buckets. The most frequent bucket is words which appear in the training set more than $10,000$ times, displayed on the left. For reference, a uniform model would have perplexity $|V| = 2.6e5$ for all frequency buckets. *LSTM comparison obtained from Chapter 4 Table 4.1

	> 10K	1K$-$10K	$100 - 1$K	< 100	All
LSTM*	12.1	219	1,197	9,725	36.4
TransformerXL (ours)	7.8	61.2	188	1,123	18.1
Compressive Transformer	**7.6**	**55.9**	**158**	**937**	**17.1**
Relative gain over TXL	2.6%	9.5%	21%	19.9%	5.8%

over the TransformerXL baseline) but obtains a much larger improvement of $\approx 20\%$ for infrequent words. Furthermore, we see **10X** improvement in modelling rare words over the prior state-of-the-art LSTM language model published in 2018 — which demonstrates the rate of progress in this area.

6.5.4 Compressibility of layers

We can use compression to better understand the model's mode of operation. We inspect how compressible Transformer's activations are as they progress through higher layers in the network.

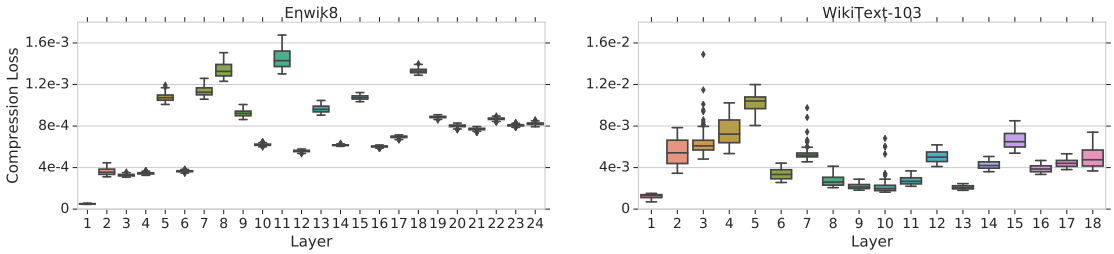


Figure 6.2: **Model analysis.** Compression loss, via attention reconstruction, broken down by layer.

We monitor the compression loss at each layer of our best-performing Compressive Transformer models trained on Enwik8 and WikiText-103 and display these in Figure 6.2. The compression loss here refers to the attention-reconstruction attention loss. We plot this for a 24 layer trained model on Enwik8, and an 18 layer model trained on WikiText-103. The compression loss for character-based language modelling is about one order of magnitude lower than that of word-level language modelling. The first layer's representations are highly compressible, however from then on there is no fixed trend. Some non-contiguous layers have a very similar compression loss (e.g. 4 & 6, 5 & 7) which suggests information is being routed from these layer pairs via the skip connection.

We note that the compression loss is about one order of magnitude higher for word-level language modelling (WikiText-103) over character-level language modelling (Enwik8). Furthermore the first layer of the Transformer is highly compressible. However there is not a clear trend of compression cost increasing with layer depth.

6.5.5 Attention

We inspect where the network is attending to on average, to determine whether it is using its compressed memory. We average the attention weight over a sample of $20,000$ sequences from a trained model on Enwik8. We aggregate the attention into eighteen buckets, six for each of the compressed memory, memory, and sequence respectively. We set the size of the sequence, memory and compressed memory all to be 768. We plot this average attention weight per bucket in Figure 6.3 with a 1σ standard error. We see most of the attention is placed on the current sequence; with a greater weight placed on earlier elements of the sequence due to the causal self-attention mechanism which masks future attention weights. We also observe there is an *increase* in attention from the oldest activations stored in the regular memory, to the activations stored in the compressed memory. **This goes against the trend of older memories being accessed less frequently — and gives evidence that the network is learning to preserve salient information**.

Optimisation Schedule

We make an observation about an interesting but undesirable meta-learning phenomenon during long-context training. When the learning rate is tuned to be much smaller (or set to zero) during training, performance degrades drastically both for the TransformerXL and the Compressive Transformer. This is displayed in Figure 6.4.

Usually we consider distributional shift from the training data to the test data, but we can also observe a shift in the model when transferring from a training

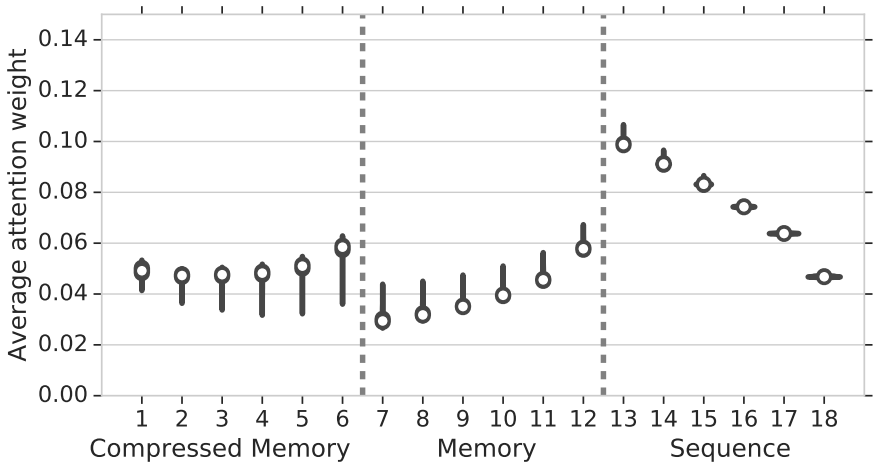


Figure 6.3: **Attention weight on Enwik8**. Average attention weight from the sequence over the compressed memory (oldest), memory, and sequence (newest) respectively. The sequence self-attention is causally masked, so more attention is placed on earlier elements in the sequence. There is an increase in attention at the transition from memory to compressed memory.

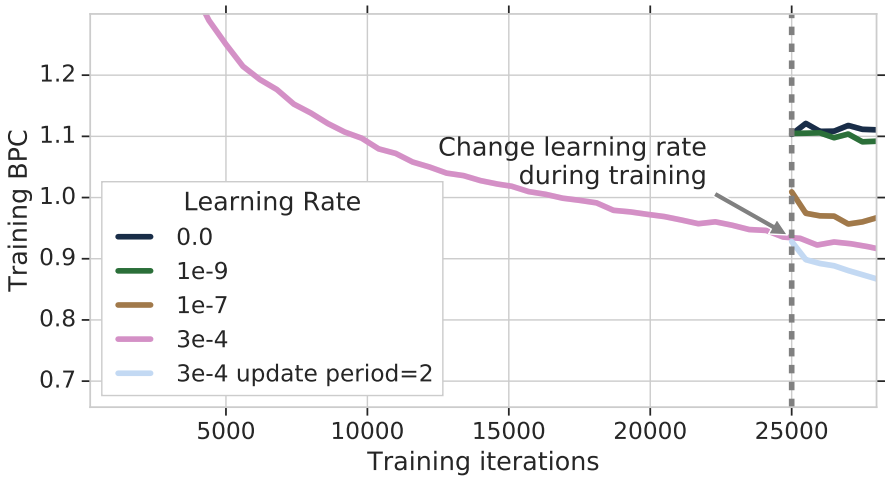


Figure 6.4: **Learning rate analysis**. Reducing the learning rate (e.g. to zero) during training (on Enwik8) harms training performance. Reducing the frequency of optimisation updates (effectively increasing the batch size) is preferable.

to evaluation mode (even when the model is evaluated on the training data). In this case, this is due to the online updating of parameters whilst processing long contiguous articles. We would like the model to generalise well to scenarios where it is not continuously optimised. Updating the parameters only at article boundaries (and then resetting the state) could be one solution for long-range memory models, but this would slow down learning significantly.

Instead, we propose reducing the frequency of optimisation updates during training. We find this allows for the best of both worlds — fast initial learning with frequent updates, and better generalisation near the end of training with less frequent updates (e.g. every 4 steps). Reducing the optimisation frequency increases the effective batch size, which has also been shown to be preferable to learning rate decay in image modelling (Smith et al., 2018). We observed a final performance improvement in our TransformerXL baseline on Enwik8, from 0.995 — which approximately replicates the published result — to 0.984 — which matches the most recent SotA architecture. We note, the additional space and compute cost of accumulating gradients is negligible across iterations, so there was no performance regression in using this scheme.

6.5.6 PG-19 Samples

We show a few different samples from the Compressive Transformer trained on PG-19. We use Nucleus Sampling with $p = 0.98$ (Holtzman et al., 2019). We choose extracts of books from the test set as prefixes. We see the model is able to continue in the style of the text, creating artificial dialogue or descriptive text, and remembering the names of characters over hundreds of words.

As the Compressive Transformer is trained without state resetting, it is actually slightly out of sample when provided with the (relatively) short contexts. This is because its memory and compressed memory may be still empty (whereas they are always full during training). However we see a trend of the samples usually improving towards the end.

Context from Baby Mine by Maragaret Mayo:

CHAPTER I

Even in college Alfred Hardy was a young man of fixed ideas and high ideals and proud of it.

His friend, Jimmy Jinks, had few ideas and no ideals, and was glad of it, and before half of their first college term had passed, Jimmy had ridded himself of all such worries as making up his own mind or directing his own morals. Alfred did all these things so much better, argued Jimmy, furthermore, Alfred LIKED to do them–Jimmy owed it to his friend to give him that pleasure.

Compressive Transformer:

He did not like doing them in the most elegant of all possible marble walks–where freshmen and sophomores stand and talk and laugh and chatter till evening falls and look cool and still and cool and beautifully white in spite of the hot sun, but had to go there for a month before going home. It happened one day that he and Jimmy were strolling together in the green shade of St. Martin's, and Jimmy naturally asked why he was so silent. He was very talkative as most boys are who have once got the habit of it on shore, so the need for answering was soon supplied.

"I am wondering," he said, "why my old friend and chum, Jimmy Jinks, is living a single life and so has no boys. Have you not noticed that many of his old friends had their own boys to follow in their footsteps and help, and often helped successfully too?"

"Oh, yes," said Alfred, "it was plain enough then that he liked to live by himself and got none of his present pleasures or entertainments because of the other fellow's giving it to him."

"Then it IS sad," said Jimmy, "that an old friend like the friend of Big Lights, Jimmy Reynolds, should be living the single life and having no boys."

"And it is pretty true, too," said Alfred, "yet I believe it is the best thing that could happen even now."

"And so do I," said Jimmy, "and I also believe that it is time someone did something to make the things worse for the other's partner."

"And so do I," said Alfred, "and so do I, and so do I."

"What can we do?" asked Jimmy.

"I don't know," said Alfred, "but if someone will come down to the valley and go through a regular mill with the old forces at work, doing the work of the two, and doing it better than any other Miller he ever saw or ever hoped to see, I think there will be a good deal of change in the things and things to be cared for for and for driven and counter-driven."

"I will come and do that mill," said Jimmy.

Context from The Diary of Samuel Pepys:

June 1st. Having taken our leaves of Sir W. Batten and my Lady, who are gone this morning to keep their Whitsuntide, Sir W. Pen and I and Mr. Gauden by water to Woolwich, and there went from ship to ship to give order for and take notice of their forwardness to go forth, and then to Deptford and did the like, having dined at Woolwich with Captain Poole at the tavern there. From Deptford we walked to Redriffe, calling at the half-way house, and there come into a room where there was infinite of new cakes placed that are made against Whitsuntide, and there we were very merry. By water home, and there did businesses of the office. Among others got my Lord's imprest of L1000 and Mr. Creed's of L10,000 against this voyage their bills signed. Having wrote letters into the country and read some things I went to bed.

Compressive Transformer:

So I rose and did some business at the office, and then to dinner, where great abundance of good things of the kitchen I only being partaker in the feast. After dinner I went to look at the yard and found it in a very bad condition, so I went home and there finding no delight in seeing things I should make shift to get in my way and so I will leave off.

So I took leave of my great friend, and the next day being Sunday, my wife and I and a great number of friends went by water to Deptford, and there took the oaths of allegiance and supremacy and promised such obedience to the Lord as should hereafter turn to the advantage of our State. Being come back we found the shops and warehouses in a pitiable condition, so my wife and I and all the rest of the friends went home and there did the like and by Saturday day I being alone in my office I found myself in a very bad situation, for my Lord, and Mr. Creed, and Pett were all standing ready to receive and receive the oaths of allegiance and right to make use of the ten days' licence I had allowed for this purpose by virtue of the law that was such a help to the servant. Having looked at the ships, I told my wife what I had seen and that I was greatly troubled at my situation and, I hope that will mend it.

So that is the end of a long and patient seeking after those things which in my opinion I deserve and may get if I can only by fair means get them.

Context from The Patrol of the Sun Dance Trail by Ralph Connor:

High up on the hillside in the midst of a rugged group of jack pines the Union Jack shook out its folds gallantly in the breeze that swept down the Kicking Horse Pass. That gallant flag marked the headquarters of Superintendent Strong, of the North West Mounted Police, whose special duty it was to preserve law and order along the construction line of the Canadian Pacific Railway Company, now pushed west some scores of miles.

Along the tote-road, which ran parallel to the steel, a man, dark of skin, slight but wiry, came running, his hard panting, his streaming face, his open mouth proclaiming his exhaustion. At a little trail that led to the left he paused, noted its course toward the flaunting flag, turned into it, then struggled up the rocky hillside till he came to the wooden shack, with a deep porch running round it, and surrounded by a rustic fence which enclosed a garden whose neatness illustrated a characteristic of the British soldier. The runner passed in through the gate and up the little gravel walk and began to ascend the steps.

Compressive Transformer:

As though by mutual consent, all the three members of the Superintendent Force were standing on the veranda watching his progress, a man came strolling slowly down the path, stopping often to gaze curiously about him, then stopping once more when he saw the four standing on the veranda. His face was dark, his eyes prominent, and the lines that furrowed his forehead showed marks of many woes, yet his step was alert and he began to mount the steps with steady strides.

The man's attention was diverted by the unexpected appearance of the two policemen; a third man came slipping up, grumbling under his breath, and stopped long enough to look doubtfully around him, then slipped away again. In the meantime, the four on the veranda continued to watch the approach of the new arrival. In spite of himself, the Superintendent Force became conscious of a curious feeling–uneasiness that was almost fear–in the presence of that strange man. He would not allow himself to admit the fact, yet he felt it in his bones. But to the watchers, the plain, everyday features of that stranger and his coming, seemed only just what the Seven White Shee owed him–their weight, their hurry, their blast.

Had a chariot been made for the good of the three horses, and had the driver been given orders that he should speed them that he might win, they would have been heartening things in the sight of the veteran and the victor. To you they would have been unintelligible to the root of your understanding. When you gaze up in the faces of those four gray horses, you can see clearly through the clouds of dust that rise from their hoofs, and discern plainly where the banker is and where the hobo. Then you will understand why you shall not press the bitter grapes and why you shall not spurn the generous doctrines. You will understand why you shall not praise the lash or the spur, for you will know where the true would be and where the false would be. Then you will understand why you, a man with reason and heart, need not tear your hair over-bitter and why you need not laugh over the blunders of an ignorant man.

About nine o'clock that morning, two buggies, drawn by powerful horses, crossed the Rubicon and turned the railroad from Sandhurst into the Hollow of the Mountains. And though the charioteers stood at their horses' heads, and their drivers cried at their loudest, there was not a man in the four teams who did not feel that his day was worth all the toil and all the peril that he had undergone. And if there were a man in them who did not know that–who did not feel that the road through the Hollow of the Mountains is made easy by the arrival of travelers and by the coming of government, there was one who did not at that moment care whether his day's work were worth all the toil and all the danger that he had had to endure or whether it were not worth more than all.

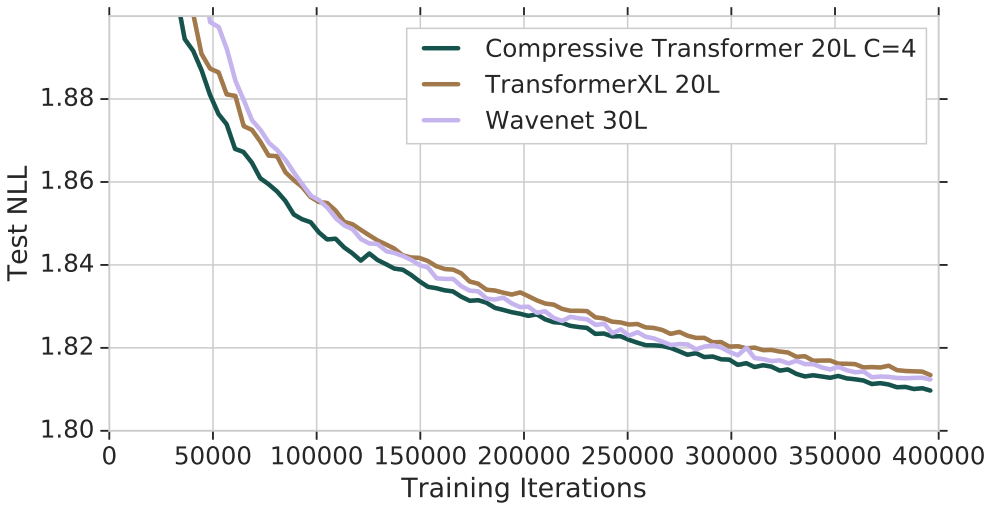


Figure 6.5: **Speech Modelling.** We see the Compressive Transformer is able to obtain competitive results against the state-of-the-art WaveNet in the modelling of raw speech sampled at 24kHz.

6.5.7 Speech

We train the Compressive Transformer on the waveform of speech to assess its performance on different modalities. Speech is interesting because it is sampled at an incredibly high frequency, but we know it contains a lot of information on the level of phonemes and entire phrases.

To encourage long-term reasoning, we refrain from conditioning the model on speaker identity or text features, but focus on unconditional speech modelling. We train the model on 24.6 hours of 24kHz North American speech data. We chunk the sequences into windows of size 3840, roughly 80ms of audio, and compare a 20-layer Compressive Transformer to a 20-layer TransformerXL and a 30-layer WaveNet model (Oord et al., 2016) — a state-of-the-art audio generative model used to serve production speech synthesis applications at Google (Oord et al., 2018). All networks have approximately 40M parameters, as WaveNet is more

parameter-efficient per layer. We train each network with 32 V100 GPUs, and a batch size of 1 per core (total batch size of 32) using synchronous training.

WaveNet processes an entire chunk in parallel, however the TransformerXL and Compressive Transformer are trained with a window size of 768 and a total memory size of $1,568$ (for the Compressive Transformer we use 768 memory + 768 compressed). We thus unroll the model over the sequence. Despite this sequential unroll, the attention-based models train at only half the speed of WaveNet. We see the test-set negative-log-likelihood in Figure 6.5, and observe that a Compressive Transformer with a compression rate of 4 is able to outperform the TransformerXL and maintain a slim advantage over WaveNet. However we only trained models for at most one week (with 32GPUs) and it would be advantageous to continue training until full convergence — before definitive conclusions are made.

6.5.8 Reinforcement Learning

Compression is a good fit for video input sequences because subsequent frames have high mutual information. Here we do not test out the Compressive Transformer on video, but progress straight to a reinforcement learning agent task that receives a video stream of visual observations — but must ultimately learn to use its memory to reason over a policy. We test the Compressive Transformer as a drop-in replacement to an LSTM in the IMPALA setup (Espeholt et al., 2018). Otherwise, we use the same training framework and agent architecture as described in the original work with a fixed learning rate of 1.5e-5 and entropy cost coefficient of 2e-3. We test the Compressive Transformer on a challenging memory task within the DMLab-30 (Beattie et al., 2016) domain, *rooms_select_nonmatching_object*. This

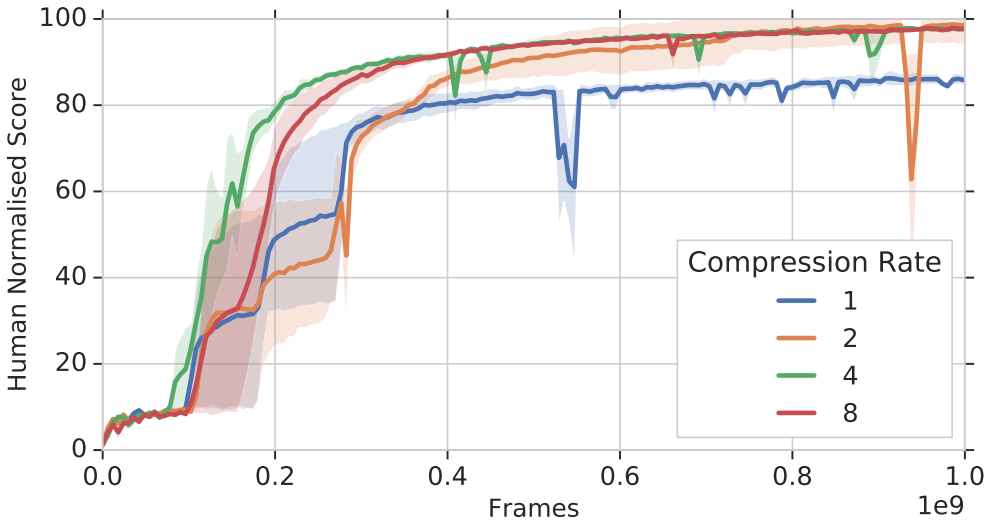


Figure 6.6: **Vision and RL**. We see the Compressive Transformer integrates visual information across time within an IMPALA RL agent, trained on an object matching task.

requires the agent to explore a room in a visually rich 3D environment and remember the object present. The agent can then advance to a second room where it must select the object *not present* in the original room. This necessitates that the agent both remember events far in the past, and also learn to efficiently reason about them.

We fix both the memory and compressed memory sizes to 64. In Figure 6.6, we present results for a range of compression rates, averaged over 3 seeds. We see that the best performing agents endowed with the Compressive Transformer are able to solve the task to human-level. We note that the model with compression rate 1 is unable to learn the task to the same proficiency. The speed of learning and stability seem to increase proportionally with higher rates of compression (up to a limit) – i.e. the effective memory window of the agent – and we find compression rate 4 to once again be the best performing. We see this as a promising sign

that the architecture is able to efficiently learn, and suitably use, compressed representations of its visual input and hope to test this more widely in future work.

6.6 Conclusion

In this chapter we explore the notion of compression as a means of extending the temporal receptive field of Transformer-based sequence models. We see a benefit to this approach in the domain of text, with the Compressive Transformer outperforming existing architectures at long-range language modelling. To continue innovation in this area, we also propose a new book-level LM benchmark, PG-19. This may be used to compare long-range language models, or to pre-train on other long-range reasoning language tasks, such as NarrativeQA (Kočiskỳ et al., 2018). We see the idea of compressive memories is applicable not only to the modality of text, but also audio, in the form of modelling the waveform of speech, and vision, within a reinforcement-learning agent trained on a maze-like memory task. In both cases, we compare to very strong baselines (Wavenet (Oord et al., 2016) and IMPALA (Espeholt et al., 2018)).

The main limitation of this work is additional complexity, if the task one wishes to solve does not contain long-range reasoning then the Compressive Transformer is unlikely to provide additional benefit. However as a means of scaling memory and attention, we do think compression is a simpler approach to dynamic or sparse attention — which often requires custom kernels to make efficient. One can build effective compression modules from simple neural network components, such as convolutions. The compression components are immediately efficient to run on

GPUs and TPUs.

Memory systems for neural networks began as compressed state representations within RNNs. The recent wave of progress using attention-based models with deep and granular memories shows us that it is beneficial to refrain from immediately compressing the past. However we hypothesise that more powerful models will contain a mixture of granular recent memories and coarser compressed memories. Future directions could include the investigation of adaptive compression rates by layer, the use of long-range shallow memory layers together with deep short-range memory, and even the use of RNNs as compressors. Compressive memories should not be forgotten about just yet.

Chapter 7

Discussion

We have explored a variety of extensions to neural networks augmented with content-based attention mechanism. We have shown that it is possible to scale memory systems with attention that is very sparse ($\geq$ 99.9%). We find that this not only provides large speedups but, perhaps more surprisingly, can stabilise learning and produce more generalisable solutions. For example, we see the Sparse Access Memory model is considerably more data efficient than its dense analogue (DAM). We also see the Sparse Differentiable Neural Computer (SDNC) outperformed the Differentiable Neural Computer at the bAbI question answering suite, halving the error rate and solving an additional task. This is despite having an identical number of parameters. The SDNC remained state-of-the-art for the jointly-trained bAbI task for over two years since its publication — demonstrating that computational performance can be aligned with better memory systems.

We have also explored compacting slot-based memories into auto-associative memories, particularly for classification. Here the slot-based memory acts as a non-

parametric classifier, collecting observation embeddings alongside corresponding class labels. Such systems have been popular in combination with a parametric neural network, as the non-parametric memory is able to learn fast and identify previously unseen classes whereas the neural network can better capture statistical regularities. To scale the potential history of such a system, we consider storing a summation of observation embeddings in the output softmax parameters of a neural network. The softmax parameters now serve as an auto-associative memory of observations corresponding to a given class. We show this can recover a similar prediction to the original non-parametric memory, without using additional space or compute. We term this the Hebbian Softmax as the resulting memory system can be written as a learning rule that is a mixture of Hebbian learning and conventional gradient descent. We show this can speed up the identification of previously unseen classes and improve the modelling of rare classes. In the latter case, we show a significant improvement in language modelling and obtain state-of-the-art LSTM-based language model performance on a Wikipedia-text benchmark, WikiText-103.

We continue the theme of memory compaction by considering a memory architecture that is considerably more space-efficient than conventional memory systems (such as the DNC, Memory Networks etc.) but is also scalable to large capacities (unlike conventional RNNs). We take inspiration from memory systems in two disjoint fields; neuroscience and computer science. Namely we take inspiration from the Hippocampal's organisation of memories as distributed overlapping codes. We also take inspiration from a data structure called a Bloom Filter which stores data as a set of sparse distributed codes, and is primarily used as a space-efficient

mechanism for determining familiarity.

We coin a neural variant of the Bloom Filter, termed the Neural Bloom Filter, which we benchmark specifically for the task of familiarity against the Bloom Filter, DNC, LSTM, and Memory Network. We train the neural networks using meta-learning to perform familiarity in one-shot, and we find that the Neural Bloom Filter can outperform both existing neural networks at scale; by solving the task using considerably less memory, but also can outperform the production-standard Bloom Filter data-structure. This has learning implications — we can develop more compressive memory architectures via overlapping representations. It also has systems applications — neural networks can significantly outperform production-level data-structures at tasks they were designed, by taking advantage of statistical structure in the underlying data.

We continue on the theme of compacting and compressing memories by applying this idea to the Transformer, which has emerged as a very powerful general-purpose sequence learning algorithm by incorporating many layers of attention to reason over the past. The Transformer suffers from the same linear-time attention per time-step as the baseline 'Dense Access Memory' models of Chapter 3, and thus it can benefit from almost all of the technologies in this study — and make for a great case-study. We specifically focus on extending its range of attention by compressing old memories to coarser-grained memories, resulting in a linear increase in its temporal receptive field without incurring additional compute. Rewardingly we see this improves modelling performance across language modelling, audio modelling, and memory-based reinforcement learning tasks — in comparison to some of the strongest baseline models. Motivated from the lack of long-range language

modelling tasks, we also propose a new book-level benchmark PG-19.

As a theme we have focused on mechanisms of improving the scalability of conventional attention, focusing on supervised learning tasks using synthetic data for interpretability and then moving to real-world tasks with practical application such as language modelling. We have produced several state-of-the-art results on competitive benchmarks, such as the question answering task bAbI and the long-range language modelling benchmark WikiText-103. Such benchmarks encourage the use of sound experimental setup, and the development of strong baselines. However we have not optimised benchmarks as an endeavour in itself, they have been used where appropriate to help us develop better long-range memory systems for neural networks. The success of some of the ideas in this study can be analyzed in terms of their incorporation across the community, which we can use as a signpost towards what remaining components are needed for life-long reasoning. We first discuss the incorporation of content-based sparsity during the course of this study and discuss how hardware has influenced sparse attention mechanisms.

7.1 Sparsity and Hardware

The development of neural-network-friendly hardware has had an important influence on the development of long-range memory during this study. Whilst the author based many experiments on CPUs in Chapter 3, which have great support for sparsity but relatively poor parallel-processing support for multiplying large matrices, e.g. Intel(R) Xeon(R) CPU E5-1650 v2 with 172GFLOPs at single precision. The development of massively parallel but dense compute has changed the course of network architectures. Google's TPUv3 chip which was released nearer

the end of this study, and was used in Chapter 6 has almost zero sparsity support but massive parallel-processing: 420 TFLOPs per chip, and 100+ PFLOPs per pod. This has meant that even a linear-time dense attention operation has become up to 2,000x faster by following progressions in hardware. This has lead to motivations towards compressive memory systems, over sparse attention, which continue to rely on simple dense linear algebra primitives and can run efficiently on TPUs.

There have been some recent efforts to transfer the ideas of sparse access memory to modern hardware and Transformer architectures, namely the use of an approximate nearest neighbour KNN to speed up content-based attention, such as the Reformer (Kitaev et al., 2019) which incorporates locality-sensitive hashing and the Routing Transformer (Roy et al., 2020) which uses a k-means nearest-neighbour search. However given the additional implementation difficulty in making these algorithms work on accelerators has meant sparse content-based attention is not ubiquitous.

One alternative, which has gained some traction, is the use of fixed-pattern sparsity in the attention matrix. This has been proposed in various works: Sparse Transformers (Child et al., 2019), Big Bird (Zaheer et al., 2020), LongFormer (Beltagy et al., 2020), and LinFormers (Wang et al., 2020). The key idea is that if the sparsity pattern is fixed to a sparse selection of the past, such as *local attention* where attention is restricted to the past 128 (say) time-steps and *random attention* where attention may access a constant-number of fixed time-step locations of the past, then this can be efficiently implemented on GPUs. The key argument in these papers is that the cost of attention has been reduced from linear-time per time-step,

to constant-time; asymptotically this improves upon even the logarithmic-time Sparse Access Memory from Chapter 3.

However the author is nonetheless skeptical that fixed-pattern sparsity could form a valid solution to lifelong reasoning, and considers this a temporary state until efficient sparse content-based attention can be efficiently implemented with modern hardware. One reason for this, is that these fixed-pattern sparsity architectures may have a very large temporal receptive field in theory, but do not appear to make use of it in practice. This can be seen quantitatively from benchmark results. The Sparse Transformer gains only a 0.5% improvement on the character modelling task moving from a dense-memory baseline with an attention window of 1,500 to a sparse attention over 12,000 tokens. In contrast the Compressive Transformer only extends the receptive field to 4,000 tokens but obtained a 2.3% improvement from the same baseline.

The need for sparse content-based attention over (or alongside) fixed-pattern attention is also intuitive. We cannot only retrieve memories which occurred an odd number of days ago, for example. It is important we can retrieve the information that we search for, and this is what sparse content-based attention delivers. There are signs that sparsity will gain a more mainstream re-entry though later editions of neural network accelerators which incorporate sparse primitives, for example Nvidia's latest GPU release, the A100, incorporates a sparse tensor core instruction set (Krashinsky et al., 2020). This thesis supports the hypothesis that the marriage of large-scale parallel processing with fine-grained dynamic sparsity will lead to sparse content-based attention becoming ubiquitous.

7.2 Compression versus Retrieval

However there remains an unresolved question over whether compressing the past is truly necessary, versus storing the raw inputs from the past and efficiently attending to them on the fly. From a philosophical perspective this question bears resemblance to the famous Chinese Room Argument (Searle et al., 1980) where one posits whether an agent that translates English to Chinese via a large rule-book truly understands Chinese. Many would consider a system which relies on a giant 'non-parametric' book of rules to not contain true understanding. In this loose analogy, storing the past in its raw format is alike to the rule-book. The agent can scroll to individual points in the past and make a decision off the local information to that time-point, but it never spent its cognitive energy to understand the higher level details of the text.

We want our agents to understand and reason over the past, and compression is one such approach to understanding (Legg and Hutter, 2007). If our agent reads the book *Life of Pi* and is asked, "Is the narrator reliable?" we would like the agent to reflect on the key events and form an opinion. We would like an agent that understands the text. A retrieval-augmented model which jumps between pages containing key-words from the query such as "reliable" and "narrator" will likely form a confused response. However the question of compressing the past versus retrieving over raw inputs can be framed empirically in terms of NLP benchmarks, in this setting retrieval is very promising.

Retrieval systems such as the Retrieval-Augmented Generation (RAG) (Lewis et al., 2020) are combining Transformer language models with approximate nearest neighbour search and sparse attention — techniques proposed in Chapter 3 — to

already attend over millions of tokens of text. One could argue such systems are already attending over a lifetime's worth of sensory input, or they certainly have the potential to in the next few years with the advance of faster hardware. This approach is currently state-of-the-art in open-domain question answering. Similar approaches of storing the entire training set and attending to it on-the-fly are also showing promise in language modelling, such as KNN-LM (Khandelwal et al., 2019) and Semi-Parametric Language Models (Yogatama et al., 2021); these models obtain performance that reaches or exceeds the Compressive Transformer. Retrieval models can locally reason over their input as they re-encode their retrieved text and — in the case of RAG — condition on this information throughout each layer of the network.

Whilst QA benchmarks heavily favour retrieval systems, it may still be preferable to query a large-scale compressive memory when more abstract reasoning is required. Let's consider the scenario of a literature exam about a particular text where factual questions concerning the plotline are posed. Two students take separate variants of the exam, one is closed-book and the other is open-book. For the closed-book exam the student must first start by reading the entire book, closing it and then approaching the questions. The open-book exam, the student skips reading the book but begins with the questions and scans the book to find relevant answers. Which student will perform best? Likely the open-book student; if they have an efficient way to scan to the correct pages in the book then they can inspect the relevant text without any loss of information or forgetting. Which student will complete the exam fastest, and thus use less cognitive compute? This will also be the open-book student, as they do not need to initially read the book.

Thus from the question answering thought experiment, the open-book student uses less compute and obtains better performance. However it intuitively appears the closed-book student should have a better comprehension of the text as they have read the entire book, and if the exam were switched to be progressively more high-level, such as writing essays of analysis of the plot-line then the closed-book student would struggle.

This hypothetical scenario represents the state of NLP comprehension benchmarks where much progress is currently being made on open-domain question answering but deeper comprehension tasks, the author would argue, are still in the stage of infancy. On open-domain QA where RAG can be thought of as the open-book student and a large language model such as GPT-3 (a Transformer with 175B parameters and 96 layers) can be thought of as the closed-book student, open-book clearly wins. When compared on open-domain QA such as Natural Questions, a few-shot primed GPT-3 obtains 29.9% whereas RAG obtains 44%, a 40% improvement. It is arguable that GPT-3 understands the web content that it has been trained over to a greater extent however, as it can be applied few-shot to many other problems with competitive performance.

Drawing out tasks which truly probe at higher-level comprehension is likely crucial to seeing a benefit to compressed memory. Book summarisation, for example, has been proposed as a task via the NarrativeQA benchmark (Kočiskỳ et al., 2018). However the reality is that we do not have models which can learn to summarise books via input-output examples, and our strongest pre-trained language models such as GPT-3 (Brown et al., 2020) cannot attend to book-length contexts yet. The task is essentially still too hard to measure real progress, and state-of-the-

art models have so far relied on copying spans of text. Whilst open-domain QA is currently a task that we can see large performance gains from large memory retrieval systems over raw data, compressive memory may begin to shine once long-context summarisation becomes more tractable.

In this thesis we have explored temporally compressing memories to a constant factor, e.g. 3×. We next discuss how we may be able to exploit a much greater temporal redundancy via weight-based episodic memories.

7.3 Memory as Weights

Where may the future lie in terms of groundbreaking new memory architectures with very different properties to those discussed in this study? We have almost always considered memory architectures that store hidden activations at some layer of a deep neural network. These activations may be stored in entirety as 'slots' in a large memory and accessed sparsely, as in Chapter 3 or they may be compacted, or compressed, to a smaller set of slots using a learnable parametric function, as in Chapter 5 & 6.

One interesting direction which moves away from memories as a simple collection of past activations, is the storage of memories within the weights of a neural network — stored in a highly compressed manner via an optimisation-based write. Of course, neural networks already do store long-term memories in their parameter weights, and this is most apparent for language models such as BERT that have an auto-associative objective function and learn many facts and associations from large corpora of text. However here we really mean an *episodic memory* from

Tulving's classification as introduced in the Background Section 2.1.1. We almost touch upon such an architecture in Chapter 4 where we use the softmax linear weights as an auto-associative memory for rarely observed classes. However the write mechanism is essentially fixed as a moving average of past activations for each class, we do not consider memories that are stored into the weights of a neural network via an optimisation process such as gradient descent.

Despite the fact that biological episodic memory systems are known to be weight-based (in the synaptic strengths of the CA3), weight-based memory systems appear to be an under-explored research direction in the deep learning memory literature. There have been a few notable works in this area. The notion of *fast weights* was proposed by Hinton and Plaut (1987) and revisited by Ba et al. (2016a); Miconi et al. (2018). Here, parametric linear maps in neural networks are replaced with dual slow and fast weights

$$h(t+1) = f\left(h(t)W_h + x(t)W_x + M(t)h_s(t+1)\right)$$

where W_x, W_h indicate regular learnable slow-weights optimised via gradient descent and $M(t)$ represents a fast-weight matrix defined as the moving average of activation outer products

$$\mathbb{R}^{d\times d} \ni M(t) = \lambda M(t) + (1-\lambda)h(t)h(t)^T \ .$$

The term $h_s(t+1)$ can be thought of as the output of s iterations of fast and

slow-weight 'reads':

$$h_s(t+1) = f\left(h(t)W_h + x(t)W_x + M(t)h_{s-1}(t+1)\right) \dots h_0(t+1) = h(t)$$

which allows it to converge its attention to a prior hidden activation. This iterative read can be thought of as type of optimisation process where $h_s(t+1)$ converges to some attractor value, and the 'fast weight' memory is almost identical to the Hopfield Network's sum of outer products $\sum_t h_t h_t^T$. However this architecture has not shown to improve the model's performance on long-range modelling tasks, it has been more so targeted towards fast adaptation. The capacity of the $d \times d$ memory is potentially limiting, as is the fixed write scheme.

A weight-based memory which uses an optimisation process to write memories was explored by Bartunov et al. (2020) and appears very promising as a highly-compressed one-shot memory system. Here, the memories are stored into the weights of an energy model $E_\theta(x) \in \mathbb{R}$ e.g. a deep convolutional neural network, which outputs a single scalar value: the energy. The energy model is meaningless upon initialisation E_{θ_0}, however over time it is learned to be a useful optimisation landscape for fast and compressed memory storage. For a training batch t the training setup is informally as follows:

$$\theta_t^* \leftarrow \texttt{minimize params } E_{\theta_t}(\mathbf{x_t}) \qquad \text{Write} \qquad (7.1)$$

$$y_t^* \leftarrow \texttt{minimize inputs } E_{\theta_t^*}(\tilde{\mathbf{x}}_\mathbf{t}) \qquad \text{Read} \qquad (7.2)$$

$$\theta_{t+1} \leftarrow \theta_t - \alpha \frac{\partial L}{\partial \theta_t} \qquad \text{Meta-learn } \theta \qquad (7.3)$$

where the write is defined to be an n-step gradient descent of the energy function

with a clamped input $\mathbf{x_t}$, optimising the network's parameters θ from an initial value θ_t to a converged value θ_t^*. These converged weights are the memory store of $\mathbf{x_t}$. The read is also an n-step gradient descent of the energy but this time with the weights clamped θ_t^* and the query inputs $\tilde{x}_t$ optimised to provide an eventual output y_t^*. The whole read and write optimisations can be backpropagated through to provide a gradient-based update to θ_t with respect to the task loss L. This whole training setup can be thought of as an instance of model-agnostic meta-learning (MAML) (Finn et al., 2017). The θ_t parameters slowly converge to shape the energy landscape to serve a useful purpose (i.e. create an energy landscape that defines a useful write) as well as find a good initial set of parameters that can be optimised with a few steps.

Whilst the experiments in Bartunov et al. (2020) are largely focused on simple auto-associative tasks on images with occlusion, it would be an interesting future direction to apply this to sequence modelling. For example one could take the approach in Chapter 6 and replace the compressive memory with this energy-based memory. One would imagine such a memory to have very different properties, more retrieval noise but a higher rate of compression. Furthermore because the write operation is defined as a generic optimisation which is meta-learned to be task specific, the network can choose what aspects of the inputs need to be stored and which classes of inputs are more important to store.

Connections between recent memory models and Hopfield Networks have been established to show that the multi-head attention operation used within Memory Networks, the NTM, DNC and Transformer can be viewed as an update formula to a continuous state-space Hopfield Network (Ramsauer et al., 2020). Thus there

may also be avenues to renew weight-based episodic memory in neural networks via differentiable Hopfield Networks that store memories in the weights and actually use many of the known components to perform updates.

Exploring this very different class of memory architectures is a natural next step to investigating compressive memory architectures which could supplement a more granular short-term slot-based memory. Ideally this would be investigated in challenging natural-data tasks, where there are challenges of noisy gradients and changing representations.

7.4 Complexity of Memory-Based Reasoning

This study has endeavoured to build better long-range memory architectures, and has achieved its goal as judged by a number of memory benchmarks and temporal modelling tasks. The longer-term goal is for these to be used in an agent or algorithm which will reason over time in a rich manner. We hope such a learning system can learn many different function approximators over time to support many types of queries, "when did I last see x?" "how many times have I spoken to y?", "what is the object which sounds like z?" versus a fixed set of simple operators "is this scene familiar or not".

The author has focused on the scaling of memory systems in this study, and has not specified a research agenda for how complex memory interactions should best arise. Partly this is due to a change in perspective during this study, both from the author and, to some extent, from the wider research community.

One approach, which more closely aligns with the research into Sparse Access

Memory, follows the idea that we should train on a set of memory tasks which probe specific functions of memory we know to be important: auto-association, repetition of sequences, arithmetic and algorithmic operations over time, and synthetic question answering tasks probing different types of reasoning. Such a system which trains well on this suite can then be incorporated into a reinforcement learning agent which performs a more complex set of memory tasks inspired by psychology experiments (such as the water maze task in Chapter 6). Then finally an agent which integrates well with its memory on these tasks can be applied to more open-ended continual reinforcement learning tasks, with the hope that memory will become a tool which it can use for many sub-tasks and we begin to see the emergence of an agent with a prototypical intelligence and the ability for rich reasoning over its stream of experience.

This research agenda is inspired by the course of development of the LSTM which was originally designed and tuned from synthetic memory tasks, and has then become naturally incorporated into many natural data supervised-learning tasks, and into reinforcement learning agents such as A3C (Mnih et al., 2016). It also fits the format of works including Memory Networks (Weston et al., 2015) which proposed a suite of synthetic question answering tasks, bAbI (Weston et al., 2014) along with the NTM and DNC (Graves et al., 2014, 2016), which proposed much more complex algorithmic memory tasks and lead to memory-based RL agents such as MERLIN (Wayne et al., 2018).

Whilst the author remains confident that these tasks remain an interesting way to benchmark neural networks of their strengths and weaknesses, it is no longer believed that the approach of training a neural network on synthetic memory tasks

tabula rasa is effective at discerning neural network architectures. Namely, training on a suite of orthogonal algorithmic tasks results in neural networks that learn to use their memory for very narrow and fixed functionalities. It also depends on several tricks for the neural network to learn; one which was imperative for scaling was the use of a *curriculum*. A large amount of performance can be obtained by hand-crafting curricula for these tasks, and whilst successively more advanced curricula can result in successively better benchmark performance — it doesn't translate to neural networks with richer memory reasoning.

Instead the author has chosen to settle from Chapter 4-6 on natural-data tasks which do not contain a set of explicit memory operations, but instead implicitly require memory in many ways. Language modelling has been the best example of this, and a recurring task over several chapters. To model natural language, the network can benefit from a natural curriculum during learning. The network can immediately observe progress by considering the previous word alongside the current (thus modelling bi-grams); after some training it can eventually learning to query distant named entities, conjugate verbs appropriately, and remain on topic over paragraphs of text.

It was not clear at the beginning of this study that language modelling would be a good benchmark for memory, nor a good task to learn long-range reasoning. The most popular language modelling benchmarks operating at the beginning of this study used short (e.g. sentence-level) stretches of text: Penn Treebank from Marcus et al. (1993) and 1B benchmark from Chelba et al. (2013). Thus there was no room for long-range memory to improve top-line performance. Furthermore even with longer range benchmarks such as WikiText-103 (Merity et al., 2016),

there was evidence that state-of-the-art neural language models would rarely focus beyond the past sentence of immediate context (Paperno et al., 2016; Daniluk et al., 2017). One reservation was that the log-loss may be so dominated by local context that one would not see any noticeable difference in the top-line performance (such as perplexity) by improving or extending memory. However we have seen demonstrably throughout this study that there is a significant amount of long-range signal in language modelling, enough to discern memory architectures. We hypothesise this will continue as book-level language modelling becomes more widely adopted, and this thesis takes a large step in the direction of this future research by releasing an openly available dataset and benchmark PG-19.

Language modelling as a rich pre-training task, for memory and language understanding, has been popularised by a contemporary set of works using Transformer-based language models (either auto-regressive or in-filling). The most notable of these are BERT (Devlin et al., 2019), GPT and GPT-2 (Radford et al., 2018, 2019); by pre-training a large language modelling state-of-the-art performance has been obtained either zero-shot or few-shot, i.e. with a small amount of fine-tuning. More recently, by scaling up the dataset and model size GPT-3 (Brown et al., 2020) has demonstrated the ability for very general and complex memory-based reasoning. Most of these are in the form of demos and thus should be treated with caution, but the model appears to be able to perform many tasks to a reasonable performance level using a small number of demonstrations, such as generating web widgets in Javascript from text description, processing invoices, or transferring the style of text. We thus see the model using its memory as a sophisticated few-shot learning system.

If one took such a powerfully pre-trained language model such as GPT-3, and fine-tuned it (or primed it with examples in memory) on the original set of algorithmic memory tasks, then we hypothesize that the system will do much better than a system trained tabula rasa, despite the architecture being the same.

One important path to complex memory systems appears to be via the further scaling of models *and* datasets — a 'bitter lesson' that has recurred in many areas of machine learning and reinforcement learning (Sutton, 2019). Indeed, the ingredient of task selection is as important to the development of better memory systems as architectural innovation. We have found rich unsupervised tasks such as language modelling can be used to train and discern better memory architectures without explicitly defining a benchmark of memory operations. It seems plausible this trend will continue beyond text to architectures which reason over other modalities, such as long streams of video. It will be interesting to see what forms of memory-based learning and reasoning we can induce from such systems as the bandwidth of data becomes larger and more aligned with the multimodal stream of data that humans perceive.

Whilst we have not yet achieved human-level lifelong reasoning within our deep temporal models, we have shown that learning is possible with highly sparse attention mechanisms scanning over hundreds of thousands of timesteps, and the huge cost of memory can be significantly decreased via compression.

Bibliography

Keith Adams. The life of a typeahead query, 2010. URL `https://www.facebook.com/Engineering/videos/432864835468/`. [Online, accessed 01-August-2018].

Rami Al-Rfou, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones. Character-level language modeling with deeper self-attention. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3159–3166, 2019.

Jay Alammar. The illustrated transformer, 2018. URL `http://jalammar.github.io/illustrated-transformer/`.

Scott Alexander. A very unlikely chess game, 2020. URL `https://slatestarcodex.com/2020/01/06/a-very-unlikely-chess-game/`.

Karl Ameriks and Desmond M Clarke. *Aristotle: Nicomachean Ethics*. Cambridge University Press, 2000.

Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*, pages 459–468. IEEE, 2006.

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David

Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pages 3981–3989, 2016.

Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C Lawrence Zitnick, and Devi Parikh. Vqa: Visual question answering. In *Proceedings of the IEEE international conference on computer vision*, pages 2425–2433, 2015.

Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, November 1998. ISSN 0004-5411. doi: 10.1145/293347.293348. URL `http://doi.acm.org/10.1145/293347.293348`.

Jimmy Ba, Geoffrey E Hinton, Volodymyr Mnih, Joel Z Leibo, and Catalin Ionescu. Using fast weights to attend to the recent past. In *Advances in Neural Information Processing Systems*, pages 4331–4339, 2016a.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016b.

Alexei Baevski and Michael Auli. Adaptive input representations for neural language modeling. 2019.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. Convolutional sequence modeling revisited, 2018a. URL `https://openreview.net/forum?id=rk8wKk-R-`.

Shaojie Bai, J Zico Kolter, and Vladlen Koltun. Trellis networks for sequence modeling. *arXiv preprint arXiv:1810.06682*, 2018b.

Ziv Bar-Yossef, Thathachar S Jayram, Ravi Kumar, and D Sivakumar. An information statistics approach to data stream and communication complexity. *Journal of Computer and System Sciences*, 68(4):702–732, 2004.

Sergey Bartunov, Jack Rae, Simon Osindero, and Timothy Lillicrap. Meta-learning deep energy-based memory models. In *International Conference on Learning Representations*, 2020.

Mayank Bawa, Tyson Condie, and Prasanna Ganesan. Lsh forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*, pages 651–660. ACM, 2005.

Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. Deepmind lab. *CoRR*, abs/1612.03801, 2016. URL `http://arxiv.org/abs/1612.03801`.

Anthony Bell and Terrence J Sejnowski. Edges are the'independent components' of natural scenes. *Advances in neural information processing systems*, 9:831–837, 1996.

Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003. ISSN 1532-4435.

Tim VP Bliss and Graham L Collingridge. A synaptic model of memory: long-term potentiation in the hippocampus. *Nature*, 361(6407):31–39, 1993.

Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

Rafal Bogacz and Malcolm W Brown. Comparison of computational models of familiarity discrimination in the perirhinal cortex. *Hippocampus*, 13(4):494–524, 2003.

Allan Borodin, Rafail Ostrovsky, and Yuval Rabani. Lower bounds for high dimensional nearest neighbor search and related problems. In *Proceedings of the thirty-first annual ACM symposium on Theory of computing*, pages 312–321, 1999.

James Bradbury, Stephen Merity, Caiming Xiong, and Richard Socher. Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576*, 2016.

Timothy F Brady, Talia Konkle, George A Alvarez, and Aude Oliva. Visual long-term memory has a massive storage capacity for object details. *Proceedings of the National Academy of Sciences*, 105(38):14325–14329, 2008.

John S Bridle. Training stochastic model recognition algorithms as networks can

lead to maximum mutual information estimation of parameters. In *Advances in neural information processing systems*, pages 211–217, 1990.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Jehoshua Bruck, Jie Gao, and Anxiao Jiang. Weighted bloom filter. In *Information Theory, 2006 IEEE International Symposium on*, pages 2304–2308. IEEE, 2006.

Jonathon Cai, Richard Shin, and Dawn Song. Making neural programming architectures generalize via recursion. *International Conference on Learning Representations*, 2016.

Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 59–65. ACM, 1978.

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 30–39. Society for Industrial and Applied Mathematics, 2004.

Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp

Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling. *arXiv preprint arXiv:1312.3005*, 2013.

Danqi Chen, Jason Bolton, and Christopher D Manning. A thorough examination of the cnn/daily mail reading comprehension task. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2358–2367, 2016.

Welin Chen, David Grangier, and Michael Auli. Strategies for training large vocabulary neural language models. *arXiv preprint arXiv:1512.04906*, 2015.

Yang Chen, Abhishek Kumar, and Jun Jim Xu. A new design of bloom filter for packet inspection speedup. In *Global Telecommunications Conference, 2007. GLOBECOM'07. IEEE*, pages 1–5. IEEE, 2007.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*, 2016.

Ronan Collobert and Jason Weston. A unified architecture for natural language

processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

Suzanne Corkin. Acquisition of motor skill after bilateral medial temporal-lobe excision. *Neuropsychologia*, 6(3):255–265, 1968.

Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*, pages 219–232. IEEE, 2017.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, 2019.

Zihang Dai, Guokun Lai, Yiming Yang, and Quoc V Le. Funnel-transformer: Filtering out sequential redundancy for efficient language processing. *arXiv preprint arXiv:2006.03236*, 2020.

Michal Daniluk, Tim Rocktaschel, Johannes Welbl, and Sebastian Riedel. Frustratingly short attention spans in neural language modeling. *Proceedings of International Conference on Learning Representations (ICLR)*, November 2017.

Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. *arXiv preprint arXiv:1612.08083*, 2016.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

Kamran Diba. Neural circuits of the hippocampus, 2018. URL `https://sites.lsa.umich.edu/diba-lab/neural-circuits-of-the-hippocampus/`.

Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership. In *International Colloquium on Automata, Languages, and Programming*, pages 385–396. Springer, 2008.

Peter C Dillinger and Panagiotis Manolios. Bloom filters in probabilistic verification. In *International Conference on Formal Methods in Computer-Aided Design*, pages 367–381. Springer, 2004.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Jeffrey L Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

David Eppstein and Michael T Goodrich. Streaming algorithms for straggler detection, 2007.

Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Volodymyr Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning*, pages 1406–1415, 2018.

Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88. ACM, 2014.

Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1126–1135, 2017.

Monica Gagliano, Vladyslav V Vyazovskiy, Alexander A Borbély, Mavra Grimon-

prez, and Martial Depczynski. Learning by association in plants. *Scientific reports*, 6:38427, 2016.

Shahabeddin Geravand and Mahmood Ahmadi. Bloom filter applications in network security: A state-of-the-art survey. *Computer Networks*, 57(18):4047–4064, 2013.

Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. 1999.

Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *Vldb*, volume 99, pages 518–529, 1999.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

Joshua Goodman. Classes for fast maximum entropy training. In *Acoustics, Speech, and Signal Processing, 2001. Proceedings.(ICASSP'01). 2001 IEEE International Conference on*, volume 1, pages 561–564. IEEE, 2001.

Edouard Grave, Armand Joulin, Moustapha Cissé, David Grangier, and Hervé Jégou. Efficient softmax approximation for gpus. *arXiv preprint arXiv:1609.04309*, 2016a.

Edouard Grave, Armand Joulin, and Nicolas Usunier. Improving neural language models with a continuous cache. *arXiv preprint arXiv:1612.04426*, 2016b.

Edouard Grave, Moustapha M Cisse, and Armand Joulin. Unbounded cache model

for online language modeling with open vocabulary. In *Advances in Neural Information Processing Systems*, pages 6044–6054, 2017.

Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

Alex Graves and Jürgen Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. In *Advances in neural information processing systems*, pages 545–552, 2009.

Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE, 2013.

Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Advances in Neural Information Processing Systems*, pages 1819–1827, 2015.

Jiatao Gu, Yong Wang, Kyunghyun Cho, and Victor OK Li. Search engine guided non-parametric neural machine translation. *arXiv preprint arXiv:1705.07267*, 2017.

Gaël Guennebaud, Benoit Jacob, Philip Avery, Abraham Bachrach, Sebastien Barthelemy, et al. Eigen v3, 2010.

Caglar Gulcehre, Sungjin Ahn, Ramesh Nallapati, Bowen Zhou, and Yoshua Bengio. Pointing the unknown words. *arXiv preprint arXiv:1603.08148*, 2016.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. Retrieval augmented language model pre-training. In *International Conference on Machine Learning*, 2020.

David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Mike Hearn and Matt Corallo. BIPS: Connection Bloom filtering, 2012. URL `https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki`. [Online; accessed 01-August-2018].

Donald O Hebb. The organization of behavior: A neurophysiological approach, 1949.

Nicolas Heess, Jonathan J Hunt, Timothy P Lillicrap, and David Silver. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455*, 2015.

Karl Moritz Hermann, Tomas Kocisky, Edward Grefenstette, Lasse Espeholt, Will Kay, Mustafa Suleyman, and Phil Blunsom. Teaching machines to read and

comprehend. In *Advances in Neural Information Processing Systems*, pages 1684–1692, 2015.

Felix Hill, Antoine Bordes, Sumit Chopra, and Jason Weston. The goldilocks principle: Reading children's books with explicit memory representations. *arXiv preprint arXiv:1511.02301*, 2015.

Geoffrey E Hinton and David C Plaut. Using fast weights to deblur old memories. In *Proceedings of the ninth annual conference of the Cognitive Science Society*, pages 177–186, 1987.

Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Sepp Hochreiter, A Steven Younger, and Peter R Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94. Springer, 2001.

Ari Holtzman, Jan Buys, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8): 2554–2558, 1982.

Marcus Hutter. The human knowledge compression contest. *URL http://prize. hutter1. net*, 6, 2012.

Sébastien Jean, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. On using very large target vocabulary for neural machine translation. *arXiv preprint arXiv:1412.2007*, 2014.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *arXiv preprint arXiv:1705.03551*, 2017.

Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 1–12. IEEE, 2017.

Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

Rudolf Kadlec, Martin Schmid, Ondřej Bajgar, and Jan Kleindienst. Text understanding with the attention sum reader network. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 908–918, 2016.

Lukasz Kaiser, Ofir Nachum, Aurko Roy, and Samy Bengio. Learning to remember rare events. In *International Conference on Learning Representations*, 2017.

Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*, 2014.

Nal Kalchbrenner, Lasse Espeholt, Karen Simonyan, Aaron van den Oord, Alex Graves, and Koray Kavukcuoglu. Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*, 2016.

Kazuya Kawakami, Chris Dyer, and Phil Blunsom. Learning to create and reuse words in open-vocabulary neural language modeling. *arXiv preprint arXiv:1704.06986*, 2017.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. In *International Conference on Learning Representations*, 2019.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *International Conference on Learning Representations*, 2019.

Reinhard Kneser and Hermann Ney. Improved backing-off for m-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 181–184. IEEE, 1995.

Tomáš Kočiskỳ, Jonathan Schwarz, Phil Blunsom, Chris Dyer, Karl Moritz Hermann, Gábor Melis, and Edward Grefenstette. The narrativeqa reading com-

prehension challenge. *Transactions of the Association for Computational Linguistics*, 6:317–328, 2018.

Ronny Krashinsky, Olivier Giroux, Stephen Jones, Nick Stam, and Sridhar Ramaswamy. Nvidia ampere architecture in-depth, 2020. URL `https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/`.

Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

Piotr Kraus and Witold Dzwinel. Nearest neighbor search by using partial kd-tree method. *Theor. Appl. Genet*, 20:149–165, 2008.

Ben Krause, Liang Lu, Iain Murray, and Steve Renals. Multiplicative lstm for sequence modelling. *arXiv preprint arXiv:1609.07959*, 2016.

Ben Krause, Emmanuel Kahembwe, Iain Murray, and Steve Renals. Dynamic evaluation of transformer language models. *CoRR*, abs/1904.08378, 2019. URL `http://arxiv.org/abs/1904.08378`.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, et al. Natural questions: a benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:453–466, 2019.

Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266): 1332–1338, 2015.

Balaji Lakshminarayanan, Daniel M Roy, and Yee Whye Teh. Mondrian forests: Efficient online random forests. In *Advances in Neural Information Processing Systems*, pages 3140–3148, 2014.

Guillaume Lample, Alexandre Sablayrolles, Marc'Aurelio Ranzato, Ludovic Denoyer, and Hervé Jégou. Large memory layers with product keys. *arXiv preprint arXiv:1907.05242*, 2019.

Jason Lee, Kyunghyun Cho, and Thomas Hofmann. Fully character-level neural machine translation without explicit segmentation. *Transactions of the Association for Computational Linguistics*, 5:365–378, 2017.

Shane Legg and Marcus Hutter. Universal intelligence: A definition of machine intelligence. *Minds and machines*, 17(4):391–444, 2007.

Patrick Lewis, Ethan Perez, Aleksandara Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kuttler, Mike Lewis, Wen tau Yih, Tim Rocktaschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. 2020.

G Lowe. Sift-the scale invariant feature transform. *Int. J*, 2:91–110, 2004.

Mitchell P Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Computational linguistics*, 19(2):313–330, 1993.

André FT Martins and Ramón Fernandez Astudillo. From softmax to sparsemax: A sparse model of attention and multi-label classification. *arXiv preprint arXiv:1602.02068*, 2016.

Ward Douglas Maurer and Theodore Gyle Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975.

James L McClelland, Bruce L McNaughton, and Randall C O'reilly. Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. *Psychological review*, 102(3):419, 1995.

ROBERTJ McEliece, Edwardc Posner, EUGENER Rodemich, and SANTOSHS Venkatesh. The capacity of the hopfield associative memory. *IEEE transactions on Information Theory*, 33(4):461–482, 1987.

Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*, 2017.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Thomas Miconi, Jeff Clune, and Kenneth O Stanley. Differentiable plasticity: training plastic neural networks with backpropagation. *arXiv preprint arXiv:1804.02464*, 2018.

Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of*

the 34th International Conference on Machine Learning, volume 70, pages 2430–2439. PMLR, 2017.

Michael Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)*, 10(5):604–612, 2002.

Michael Mitzenmacher. A model for learned bloom filters and related structures. *arXiv preprint arXiv:1802.00884*, 2018a.

Michael Mitzenmacher. Optimizing learned bloom filters by sandwiching. *arXiv preprint arXiv:1803.01474*, 2018b.

Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

Rajeev Motwani, Assaf Naor, and Rina Panigrahy. Lower bounds on locality sensitive hashing. *SIAM Journal on Discrete Mathematics*, 21(4):930–935, 2007.

Marius Muja and David G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 36, 2014.

Shanmugavelayutham Muthukrishnan et al. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005.

Kazu Nakazawa, Linus D Sun, Michael C Quirk, Laure Rondi-Reig, Matthew A

Wilson, and Susumu Tonegawa. Hippocampal ca3 nmda receptors are crucial for memory acquisition of one-time experience. *Neuron*, 38(2):305–315, 2003.

Sharan Narang, Erich Elsen, Gregory Diamos, and Shubho Sengupta. Exploring sparsity in recurrent neural networks. *International Conference on Learning Representations*, 2017.

David Nister and Henrik Stewenius. Scalable recognition with a vocabulary tree. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 2161–2168. Ieee, 2006.

John O'Keefe and Jonathan Dostrovsky. The hippocampus as a spatial map: preliminary evidence from unit activity in the freely-moving rat. *Brain research*, 1971.

Christopher Olah. Understanding lstm networks. 2015.

Aaron Oord, Yazhe Li, Igor Babuschkin, Karen Simonyan, Oriol Vinyals, Koray Kavukcuoglu, George Driessche, Edward Lockhart, Luis Cobo, Florian Stimberg, et al. Parallel wavenet: Fast high-fidelity speech synthesis. In *International Conference on Machine Learning*, pages 3915–3923, 2018.

Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*, 2016.

Randall C O'Reilly. Biologically plausible error-driven learning using local activation differences: The generalized recirculation algorithm. *Neural computation*, 8(5):895–938, 1996b.

Randall C O'Reilly. *The Leabra model of neural interactions and learning in the neocortex*. PhD thesis, PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1996a.

D Paperno, G Kruszewski, A Lazaridou, QN Pham, R Bernardi, S Pezzelle, M Baroni, G Boleda, R Fernández, K Erk, et al. The lambada dataset: Word prediction requiring a broad discourse context. Association for Computational Linguistics, 2016.

Robert Parker, David Graff, Junbo Kong, Ke Chen, and Kazuaki Maeda. English gigaword fifth edition ldc2011t07. dvd. *Philadelphia: Linguistic Data Consortium*, 2011.

Loïc Paulevé, Hervé Jégou, and Laurent Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. *Pattern recognition letters*, 31(11):1348–1358, 2010.

Ivan Petrovitch Pavlov and W Gantt. Lectures on conditioned reflexes: Twenty-five years of objective study of the higher nervous activity (behaviour) of animals. 1928.

Yan Qiao, Tao Li, and Shigang Chen. One memory access bloom filters and their generalization. In *INFOCOM, 2011 Proceedings IEEE*, pages 1745–1753. IEEE, 2011.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training, 2018.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya

Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 2019.

Jack Rae, Jonathan J Hunt, Ivo Danihelka, Timothy Harley, Andrew W Senior, Gregory Wayne, Alex Graves, and Tim Lillicrap. Scaling memory-augmented neural networks with sparse reads and writes. In *Advances in Neural Information Processing Systems*, pages 3621–3629, 2016.

Jack W. Rae and Timothy P Lillicrap. A new model and dataset for long-range memory, 2020. URL `https://deepmind.com/blog/article/A_new_model_and_dataset_for_long-range_memory`.

Jack W Rae, Chris Dyer, Peter Dayan, and Timothy P Lillicrap. Fast parametric learning with activation memorization. *arXiv preprint arXiv:1803.10049*, 2018.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21:1–67, 2020.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1264. URL `https://www.aclweb.org/anthology/D16-1264`.

Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Lukas Gruber, Markus Holzleitner, Milena Pavlović, Geir Kjetil

Sandve, Victor Greiff, et al. Hopfield networks is all you need. *arXiv preprint arXiv:2008.02217*, 2020.

Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning. *International Conference on Learning Representations*, 2016.

Blake A Richards and Paul W Frankland. The persistence and transience of memory. *Neuron*, 94(6):1071–1084, 2017.

Edmund T Rolls. A computational theory of episodic memory formation in the hippocampus. *Behavioural brain research*, 215(2):180–196, 2010.

Frank Rosenblatt. Principles of neurodynamics. perceptrons and the theory of brain mechanisms. 1961.

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *arXiv preprint arXiv:2003.05997*, 2020.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, 1986.

Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *arXiv preprint arXiv:1402.1128*, 2014.

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and T Lillicrap. Meta-learning with memory-augmented neural networks. In *International conference on machine learning*, 2016a.

Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Tim-

othy Lillicrap. Meta-learning with memory-augmented neural networks. In *Proceedings of the 33rd International Conference on Machine Learning*, pages 1842–1850, 2016b.

Adam Santoro, Ryan Faulkner, David Raposo, Jack Rae, Mike Chrzanowski, Theophane Weber, Daan Wierstra, Oriol Vinyals, Razvan Pascanu, and Timothy Lillicrap. Relational recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 7299–7310, 2018.

John R Searle et al. Minds, brains, and programs. *The Turing Test: Verbal Behaviour as the Hallmark of Intelligence*, pages 201–224, 1980.

Claude E Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers)*, pages 464–468, 2018.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Guizhu Shen, Qingping Tan, Haoyu Zhang, Ping Zeng, and Jianjun Xu. Deep learning with gated recurrent unit networks for financial sequence predictions. *Procedia computer science*, 131:895–903, 2018.

Gordon M Shepherd. *The synaptic organization of the brain*. Oxford university press, 2004.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2019.

Paul Smith, 2019. URL https://www.geeksforgeeks.org/implementing-our-own-hash-table-with-separate-chaining-in-java/.

Sam Smith, Pieter jan Kindermans, Chris Ying, and Quoc V. Le. Don't decay the learning rate, increase the batch size. 2018. URL https://openreview.net/pdf?id=B1Yy1BxCZ.

Pablo Sprechmann, Siddhant Jayakumar, W. Jack Rae, Alexander Pritzel, Adria Badia Puigdomenech, Benigno Uria, Oriol Vinyals, Demis Hassabis, Razvan Pascanu, and Charles Blundell. Memory-based parameter adaptation. *International Conference on Learning Representations*, 2018.

Philip Sterne. Efficient and robust associative memory from a generalized bloom filter. *Biological cybernetics*, 106(4-5):271–281, 2012.

Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 2*, pages 2440–2448. MIT Press, 2015.

Sainbayar Sukhbaatar, Édouard Grave, Piotr Bojanowski, and Armand Joulin. Adaptive attention span in transformers. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 331–335, 2019.

Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks

for language modeling. In *Thirteenth Annual Conference of the International Speech Communication Association*, 2012.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc., 2014. URL `http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf`.

Richard Sutton. The bitter lesson. *Incomplete Ideas (blog), March*, 13:12, 2019.

Makarand Tapaswi, Yukun Zhu, Rainer Stiefelhagen, Antonio Torralba, Raquel Urtasun, and Sanja Fidler. Movieqa: Understanding stories in movies through question-answering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4631–4640, 2016.

Wilson L Taylor. "cloze procedure": A new tool for measuring readability. *Journalism quarterly*, 30(4):415–433, 1953.

Sebastian Thrun. Lifelong learning algorithms. In *Learning to learn*, pages 181–209. Springer, 1998.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *Neural networks for machine learning*, 4(2):26–31, 2012.

Endel Tulving. Elements of episodic memory. 1985.

Endel Tulving et al. Episodic and semantic memory. *Organization of memory*, 1: 381–403, 1972.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

Oriol Vinyals, Charles Blundell, Timothy Lillicrap, Koray Kavukcuoglu, and Daan Wierstra. Matching networks for one shot learning. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pages 3637–3645, 2016.

Sinong Wang, Belinda Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Xiujun Wang, Yusheng Ji, Zhe Dang, Xiao Zheng, and Baohua Zhao. Improved weighted bloom filter and space lower bound analysis of algorithms for approximated membership querying. In *International Conference on Database Systems for Advanced Applications*, pages 346–362. Springer, 2015.

Larry Wasserman. *All of nonparametric statistics*. Springer Science & Business Media, 2006.

Greg Wayne, Chia-Chun Hung, David Amos, Mehdi Mirza, Arun Ahuja, Agnieszka Grabska-Barwinska, Jack Rae, Piotr Mirowski, Joel Z Leibo, Adam Santoro, et al. Unsupervised predictive memory in a goal-directed agent. *arXiv preprint arXiv:1803.10760*, 2018.

Jason Weston. Icml 2016 tutorial on memory networks for language understanding, 2016. URL `http://www.thespermwhale.com/jaseweston/icml2016/`.

Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *arXiv preprint arXiv:1410.3916*, 2014.

Jason Weston, Antoine Bordes, Sumit Chopra, Alexander M Rush, Bart van Merriënboer, Armand Joulin, and Tomas Mikolov. Towards ai-complete question answering: A set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*, 2015.

Felix Wu, Angela Fan, Alexei Baevski, Yann N Dauphin, and Michael Auli. Pay less attention with lightweight and dynamic convolutions. *arXiv preprint arXiv:1901.10430*, 2019.

Yan Wu, Greg Wayne, Alex Graves, and Timothy Lillicrap. The kanerva machine: A generative distributed memory. In *International Conference on Learning Representations*, 2018a.

Yan Wu, Gregory Wayne, Karol Gregor, and Timothy Lillicrap. Learning attractor dynamics for generative memory. In *Advances in Neural Information Processing Systems*, pages 9401–9410, 2018b.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Yi Yang, Wen-tau Yih, and Christopher Meek. Wikiqa: A challenge dataset for open-domain question answering. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 2013–2018, 2015.

Zhilin Yang, Zihang Dai, Ruslan Salakhutdinov, and William W Cohen. Break-

ing the softmax bottleneck: a high-rank rnn language model. *arXiv preprint arXiv:1711.03953*, 2017.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, 2018.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

Andrew Chi-Chih Yao. Some complexity questions related to distributive computing (preliminary report). In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 209–213. ACM, 1979.

Dani Yogatama, Cyprien de Masson d'Autume, and Lingpeng Kong. Adaptive semiparametric language models. *arXiv preprint arXiv:2102.02557*, 2021.

Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *arXiv preprint arXiv:2007.14062*, 2020.

Wojciech Zaremba and Ilya Sutskever. Reinforcement learning neural turing machines. *arXiv preprint arXiv:1505.00521*, 2015.

Rui Zhao, Ruqiang Yan, Zhenghua Chen, Kezhi Mao, Peng Wang, and Robert X Gao. Deep learning and its applications to machine health monitoring. *Mechanical Systems and Signal Processing*, 115:213–237, 2019.

Fengwei Zhou, Bin Wu, and Zhenguo Li. Deep meta-learning: Learning to learn in the concept space. *arXiv preprint arXiv:1802.03596*, 2018a.

Luowei Zhou, Yingbo Zhou, Jason J Corso, Richard Socher, and Caiming Xiong. End-to-end dense video captioning with masked transformer. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8739–8748, 2018b.

Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.

Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 4189–4198. JMLR. org, 2017.

George K Zipf. The psychology of language. *NY Houghton-Mifflin*, 1935.

Appendix A

Scaling Memory with Sparsity

A.1 Training details

Here we provide additional details on the training regime used for our experiments used in Figure 3.4. To avoid bias in our results, we chose the learning rate that worked best for DAM (and not SAM). We tried learning rates $\{10^{-6}, 5 \times 10^{-5}, 10^{-5}, 5 \times 10^{-4}, 10^{-4}\}$ and found that DAM trained best with 10^{-5}. We also tried values of K $\{4, 8, 16\}$ and found no significant difference in performance across the values. We used 100 hidden units for the LSTM (including the controller LSTMs), a minibatch of 8, 8 asynchronous workers to speed up training, and `RMSProp` (Tieleman and Hinton, 2012) to optimise the controller. We used 4 memory access heads and configured SAM to read from only $K = 4$ locations per head.

A.2 Benchmarking details

Each model contained an LSTM controller with 100 hidden units, an external memory containing N slots of memory, with word size 32 and 4 access heads. For speed benchmarks, a minibatch size of 8 was used to ensure fair comparison - as many dense operations (e.g. matrix multiplication) can be batched efficiently. For memory benchmarks, the minibatch size was set to 1.

We used Torch7 (Collobert et al., 2011) to implement SAM, DAM, NTM, DNC and SDNC. Eigen v3 (Guennebaud et al., 2010) was used for the fast sparse tensor operations, using the provided CSC and CSR formats. All benchmarks were run on a Linux desktop running Ubuntu 14.04.1 with 32GiB of RAM and an Intel Xeon E5-1650 3.20GHz processor with power scaling disabled.

Appendix B

Compressive Memory for Identifying Rare Classes

B.1 Language Model details

For WikiText-103 we swept over LSTM hidden sizes $\{1024, 2048, 4096\}$, no. LSTM layers $\{1, 2\}$, embedding dropout $\{0, 0.1, 0.2, 0.3\}$, use of layer norm (Ba et al., 2016b) $\{True, False\}$, and whether to share the input/output embedding parameters $\{True, False\}$ totalling 96 parameters.

A single-layer LSTM with 2048 hidden units with tied embedding parameters and an input dropout rate of 0.3 was selected, and we used this same model configuration for the other language corpora. We trained the models on 8 P100 Nvidia GPUs by splitting the batch size into 8 sub-batches, sending them to each GPU and summing the resulting gradients. The total batch size used was 512 and a sequence length of 100 was chosen. Gradients were clipped to a maximum norm

value of 0.1. We did not pass the state of the LSTM between sequences during training, however the state is passed during evaluation.

B.2 Dynamic Evaluation Parameters

For the Neural Cache, we swept over the hyper-parameters:

- Softmax inverse temperature: $\theta_{cache} \in \{0.1, 0.2, 0.3\}$
- Cache output interpolation: $\lambda_{cache} \in \{0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35\}$
- Cache size $n_{cache} \in \{1000, 5000, 8000, 9000, 10000\}$

and chose $\theta_{cache} = 0.3$, $\lambda_{cache} = 0.1$, $n_{cache} = 10000$ by sweeping over the validation set.

For the mixture of Neural Cache and MbPA we swept over the same cache parameters, alongside:

- MbPA output interpolation: $\lambda_{mbpa} \in \{0.02, 0.04, 0.06, 0.08, 0.10\}$,
- Number of neighbours retrieved from memory: $K \in \{512, 1024\}$,
- Number of MbPA steps: $n_{mbpa} \in \{1, 2\}$

and selected $\lambda_{mbpa} = 0.04, \lambda_{cache} = 0.1, \theta_{cache} = 0.3, K = 1024, n_{mbpa} = 1, n_{cache} = 10000$. We also selected the MbPA learning rate $\alpha_{lr} = 0.3$, and the L2-regularization $\beta_{mbpa} = 0.5$ on the MbPA-modified parameters. The memory size for MbPA was chosen to be equal to the cache size.

Appendix C

Compressive Memory for Data Structures

C.1 Model Size

For the MNIST experiments we used a 3-layer convolutional neural network with 64 filters followed by a two-layer feed-forward network with 64&128 hidden-layers respectively. The number of trainable parameters in the Neural Bloom Filter (including the encoder) is $243,437$ which amounts to 7.8Mb at 32-bit precision. We did not optimise the encoder architecture to be lean, as we consider it part of the library in a sense. For example, we do not count the size of the hashing library that an implemented Bloom Filter relies on, which may have a chain of dependencies, or the package size of TensorFlow used for our experiments. Nevertheless we can reason that when the Neural Bloom Filter is 4kb smaller than the classical, such as for the non-uniform instance-based familiarity in Figure 5.2b, we would expect to

see a net gain if we have a collection of at least $1,950$ data-structures. We imagine this could be optimised quite significantly, by using 16-bit precision and perhaps using more convolution layers or smaller feed-forward linear operations.

For the database experiments we used an LSTM character encoder with 256 hidden units followed by another 256 feed-forward layer. The number of trainable parameters in the Neural Bloom Filter $419,339$ which amounts to 13Mb. One could imagine optimising this by switching to a GRU or investigating temporal convolutions as encoders.

C.2 Hyper-Parameters

We swept over the following hyper-parameters, over the range of memory sizes displayed for each task. We computed the best model parameters by selecting those which resulted in a model consuming the least space. This depends on model performance as well as state size. The Memory Networks memory size was fixed to equal the input size (as the model does not arbitrate what inputs to avoid writing).

Memory Size (DNC, NBF)	{2, 4, 8, 16, 32, 64}
Word Size (MemNets, DNC, NBF)	{2, 4, 6, 8, 10}
Hidden Size (LSTM)	{2, 4, 8, 16, 32, 64}
Sphering Decay η (NBF)	{0.9, 0.95, 0.99}
Learning Rate (all)	{1e-4, 5e-5}

Table C.1: Hyper-parameters considered

C.3 Experiment Details

For the class-based familiarity task, and uniform sampling task, the model was trained on the training set and evaluated on the test set. For the class-based task sampling, a class is sampled at random and S is formed from a random subset of images from that class. The queries q are chosen uniformly from either S or from images of a different class.

For the non-uniform instance-based familiarity task we sampled images from an exponential distribution. Specifically we used a fix permutation of the training images, and from that ordering chose $p(i_{th}$ image$) \propto 0.999^i$ for the images to store. The query images were selected uniformly. We used a fixed permutation (or shuffle) of the images to ensure most probability mass was not placed on images of a certain class. I.e. by the natural ordering of the dataset we would have otherwise almost always sampled 0 images. This would be confounding task non-uniformity for other latent structure to the sets. Because the network needed to relate the image to its frequency of occurrence for task, the models were evaluated on the training set. This is reasonable as we are not wishing for the model to visually generalise to unseen elements in the setting of this exact-familiarity task. We specifically want the network weights to compress a map of image to probability of storage.

For the database task a universe of $2.5M$ unique tokens were extracted from Giga-Word v5. We shuffled the tokens and placed 2.3M in a training set and 250K in a test set. These sets were then sorted alphabetically. A random subset, representing an SSTable, was sampled by choosing a random start index and selecting the next n elements, which form our set S. Queries are sampled uniformly at random

from the universe set. Models are trained on the training set and evaluated on the test set.