Reinforcement Learning with Auxiliary Memory

Sterling Suggs
Brigham Young University
Reinforcement Learning with Auxiliary Memory

Sterling Suggs

A thesis submitted to the faculty of Brigham Young University in partial fulfillment of the requirements for the degree of Master of Science

David Wingate, Chair
Jacob Crandall
Daniel Zappala

Department of Computer Science
Brigham Young University

Copyright © 2021 Sterling Suggs
All Rights Reserved
Deep reinforcement learning algorithms typically require vast amounts of data to train to a useful level of performance. Each time new data is encountered, the network must inefficiently update all of its parameters. Auxiliary memory units can help deep neural networks train more efficiently by separating computation from storage, and providing a means to rapidly store and retrieve precise information. We present four deep reinforcement learning models augmented with external memory, and benchmark their performance on ten tasks from the Arcade Learning Environment. Our discussion and insights will be helpful for future RL researchers developing their own memory agents.

Keywords: reinforcement learning, auxiliary memory, neural computer, Atari, machine learning, Q-learning
ACKNOWLEDGMENTS

I would like to thank David for the guidance and motivation he provided, for the weekly meetings (when we remembered them), and for his confidence that I could do this hard thing.

I express appreciation to Tori for encouraging me and believing in me even when I didn’t, and to Rachel for the phone calls when I was stuck.

Finally, I gratefully acknowledge the support of the National Science Foundation via grant IIS 1652950. Any opinions, findings, conclusions, or recommendations expressed in this material are mine and do not necessarily reflect the views of the National Science Foundation.
# Table of Contents

List of Figures vi

List of Tables viii

1 Introduction 1

1.1 Thesis statement .................................................. 3
1.2 Thesis outline ..................................................... 3

2 Background 4

2.1 Reinforcement learning .............................................. 4
2.1.1 The Markov Decision Process .............................. 5
2.1.2 Policy and value functions .................................. 6
2.1.3 Optimality and learning ...................................... 7
2.1.4 Deep reinforcement learning and the Arcade Learning Environment 11
2.2 Auxiliary memory .................................................. 12
2.2.1 Defining features ............................................. 15
2.2.2 Comparison—LSTM .......................................... 17
2.2.3 Comparison—replay buffer ................................. 18

3 Related Work 20

3.1 The Differentiable Neural Computer .......................... 20
3.2 Navigation with memory ........................................... 21
3.3 Language with memory ........................................... 22
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4</td>
<td>Atari with memory</td>
<td>23</td>
</tr>
<tr>
<td>3.5</td>
<td>Learning to learn</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>Model Overview</td>
<td>28</td>
</tr>
<tr>
<td>4.1</td>
<td>Controller network</td>
<td>28</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Loss</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Memory unit</td>
<td>31</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Definitions and notation</td>
<td>31</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Indexing</td>
<td>31</td>
</tr>
<tr>
<td>4.2.3</td>
<td>The read operation</td>
<td>32</td>
</tr>
<tr>
<td>4.2.4</td>
<td>The write operation</td>
<td>33</td>
</tr>
<tr>
<td>4.3</td>
<td>Preprocessing</td>
<td>34</td>
</tr>
<tr>
<td>5</td>
<td>Models 1 and 2: Differentiable Neural Computer</td>
<td>36</td>
</tr>
<tr>
<td>5.1</td>
<td>Model description</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>Conceptual difficulties</td>
<td>41</td>
</tr>
<tr>
<td>6</td>
<td>Models 3 and 4: Product Key Memory</td>
<td>43</td>
</tr>
<tr>
<td>6.1</td>
<td>Model description</td>
<td>44</td>
</tr>
<tr>
<td>7</td>
<td>Experiments and Analysis</td>
<td>48</td>
</tr>
<tr>
<td>7.1</td>
<td>Plots</td>
<td>49</td>
</tr>
<tr>
<td>7.2</td>
<td>Discussion</td>
<td>62</td>
</tr>
<tr>
<td>7.3</td>
<td>Qualitative observations</td>
<td>65</td>
</tr>
<tr>
<td>8</td>
<td>Conclusions and Future Work</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>References</td>
<td>70</td>
</tr>
<tr>
<td>A</td>
<td>Performance Data</td>
<td>75</td>
</tr>
</tbody>
</table>
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Basic data flow of an augmented memory model.</td>
</tr>
<tr>
<td>4.1</td>
<td>A simplified outline of the hybrid memory Rainbow neural network. We have denoted convolution as ( \circledast ), a fully connected layer by ( \otimes ), element-wise addition by ( \oplus ), and the sigmoid function with ( \sigma ). The dotted line spanning the memory unit denotes the optional residual connection. Numbers of convolutions per layer and scale of embeddings may be inaccurate. See Hessel et al. [15] for additional details.</td>
</tr>
<tr>
<td>5.1</td>
<td>A depiction of the DNC read and write process. The write query is used to select locations for writing, which are updated with the write and erase vectors. The read vectors are produced by averaging over memory locations weighted by similarity to the read vectors.</td>
</tr>
<tr>
<td>6.1</td>
<td>An example of the Product Key Memory read process. (a) The memory matrix, showing how two sets of ( \sqrt{d_M} = 3 ) subkeys induce a memory with ( d_M = 9 ) rows. (b) A query is split in half, and each half is compared to one subkey set. The ( k = 2 ) best matches from each are selected. (c) The ( k ) product keys nearest the query are now guaranteed to be in the set induced by the selected subkeys, which is of size ( k^2 = 4 ). The full query is now compared to this set, yielding the ( k ) desired keys, from which can be obtained the desired values. Figure adapted from Lample et al. [20].</td>
</tr>
</tbody>
</table>
7.1 Median performance of each model across all ten games, normalized to human performance. ................................. 51
7.2 Space Invaders ......................................................... 52
7.3 Alien ................................................................. 53
7.4 Frostbite ............................................................. 54
7.5 Gravitar .............................................................. 55
7.6 Breakout ............................................................. 56
7.7 Enduro ............................................................... 57
7.8 Tutankham .......................................................... 58
7.9 Bowling ............................................................... 59
7.10 Venture .............................................................. 60
7.11 Seaquest .......................................................... 61
List of Tables

4.1 Q-network architectural hyper-parameters from Rainbow. .................. 30
4.2 Preprocessing parameters inherited from DQN via Rainbow. ............... 35

5.1 Hyper-parameters for the DNC memory layer. The DNC does not have
distinct keys and values, but each memory location is filled by a single cell.
Because each pass through the network involves the entire memory matrix,
computational constraints limit the number and size of memory cells. ........ 40

6.1 Hyper-parameters for the Product Key Memory layer. As recommended by
Lample et al. [20], this layer has a higher learning rate than the rest of the
network. This helps counteract the sparse training signal each parameter
receives due to being used in only a fraction of the updates. ................. 47

7.1 Training hyper-parameters retained from DQN. ............................... 49
7.2 Additional hyper-parameters retained from Rainbow. The * indicates this
parameter differs from the original paper. ........................................ 49

A.1 Periodic evaluation scores. Standard deviation in parentheses. .......... 77
Reinforcement learning (RL) agents are uncommon in the real world. Part of the reason is that they learn inefficiently; that is, it takes a lot of feedback to bring about a desired change in behavior. Humans excel at using little feedback or just a few examples to learn patterns or adapt to changing circumstances. Some of this flexibility would go a long way in pushing the capabilities of RL agents.

As an example, suppose we want to use RL to train robots to clean houses. When a robot puts something in the correct location, it receives a positive reward, and when it puts something in the wrong location, it receives a negative reward. With this reward signal, the robot updates the parameters of the model governing its behavior. However, given the learning rates of current training algorithms, the robot will have to tidy hundreds of millions of items in thousands of different living spaces before gaining competence. This process would be stunningly expensive even if the robots are trained in parallel and in partial simulation.

Besides the cost of training, there is an issue with adaptability. Suppose the world’s biggest tech companies do succeed in training and marketing this robot, and suppose you bring one home and put it to work cleaning your house. Because you are still in college, your baby chews on a rubber spatula for a teether and shakes a container of sprinkles for a rattle. (How you afforded the robot is left as an exercise for the reader.) Each time the robot puts the sprinkles and spatula into the kitchen instead of the toy box, you can send it a signal that it did the wrong thing. A human would only need one or two such experiences before modifying their behavior, but a robot that stores its entire world model in one convoluted
set of network weights will need to repeat this experiment thousands of times before it can sufficiently update the weights to change behavior. Most likely, this will take longer than the number of months your child is playing with the spatula.

In machine learning, *sample complexity* refers to the number of training samples needed for a model to fit a function to a given degree of accuracy. This quantity tends to be high for neural networks, which buy capacity to express complicated functions at the cost of long training times. To take some real-life examples, Rainbow DQN trains for a “full run” of 200 million frames [15], while AlphaGo Zero trained for nearly 5 million entire games [30]. OpenAI’s Dactyl robot hand trains for a hundred simulated years of experience, or 50 hours with 6144 CPUs and 8 GPUs [1]. Finally, OpenAI trained a team of RL agents to compete in the video game Dota “without fundamental advances” in architecture or algorithm. This was only possible with extraordinary amounts of computational power: using 256 GPUs and 128,000 CPUs, their agents played an equivalent of 180 years of Dota every day [24].

Humans achieve competence at these tasks with orders of magnitude less practice. A primary reason it takes so long for an RL agent to learn is because the agent relies on gradient propagation to update the model weights [6]. When an RL agent takes an action and receives a reward, either positive or negative, the weights are updated incrementally to increase or decrease the likelihood of taking that action again. As with supervised learning, it takes thousands of examples and thousands of tiny updates to add up to a meaningful change. This is necessary to keep the learning algorithm stable. But it means that in standard neural networks, where all knowledge is tangled up inside the weights and all weights contribute to every prediction, adjusting or adding new knowledge will necessarily be inefficient.

Another contributor to slowness in RL, particularly deep RL, is weak inductive bias [6]. Deep neural networks are low-bias models; that is, they make few assumptions about the function they are trying to fit. This means we can apply them to a huge range of problem domains, but they are costly to train, as expected due to the bias-variance trade-off. Both these issues, weak bias and slow weight updates, can be ameliorated through something called
auxiliary or external memory. With auxiliary memory, a neural network can rapidly encode and retrieve information without gradient propagation. In addition, auxiliary memory can allow a neural network to enjoy a hierarchy of learning, with a slow learning process powering a fast one. Through this meta-learning, or learning to learn, an agent can use past experience to narrow its hypothesis set, increasing (correct) bias and improving sample efficiency [6].

1.1 Thesis statement

Deep reinforcement learning could be much more efficient if the agent could rapidly store, access, and adjust specific information independent of the rest of the network. This can be accomplished through an auxiliary memory mechanism, equipped with explicit read and write functions. Prior research suggests that RL agents appropriately equipped with auxiliary memory will learn faster and adapt faster than otherwise similar models.

We present four auxiliary memory models and benchmark their performance across ten tasks from the popular Arcade Learning Environment. RL agents can already surpass human performance on most of the tasks; our goal is to shorten the time it takes to achieve that level of performance. We hope that the discussion and insights presented here will provide valuable reference and groundwork for those seeking to continue this research.

1.2 Thesis outline

Chapter 2 reviews essential background from reinforcement learning, and gives an overview of auxiliary memory. Chapter 3 summarizes some of the prior research in this area. The remaining chapters constitute our work. Chapter 4 gives an overview of features common to our models, while chapters 5 and 6 specify our models in detail. In chapter 7, we describe our experiments and discuss the results. Finally, in chapter 8 we make conclusions, point out shortcomings of our work, and suggest ideas for further research.
Chapter 2

Background

In this chapter, we give a refresher on the basic terms and concepts in reinforcement learning. Then we present an overview of auxiliary memory, and discuss the differences between it and similar ideas.

2.1 Reinforcement learning

Reinforcement learning is distinct from supervised and unsupervised learning. There are no labeled examples, nor does it simply seek to discover structure in data. Reinforcement learning attempts to solve a sequential decision making problem.

This problem is framed in context of an agent (one who acts) operating in an environment. The agent must be able to sense some aspects of the environment, and then take actions which influence the state of the environment, possibly in stochastic ways. This in turn affects what states and choices the agent may encounter in the future. Certain states may have rewards associated with them, which the agent receives upon arrival in those states. Our desire is for the agent to achieve a certain goal state; in contrast, the agent’s only goal is to maximize the amount of reward it gets from the environment. Therefore, the environment must be carefully designed to give rewards for the right states. As the agent chooses actions, visits states, and receives reward, it learns optimal behavior through trial and error.
2.1.1 The Markov Decision Process

The reinforcement learning problem can be formalized as a Markov Decision Process (MDP). An MDP is a tuple \((S, A, R, P)\), where \(S\) is a set of states, and \(A\) is a set of actions. \(R : S \times A \rightarrow \mathcal{R} \subseteq \mathbb{R}\) is a function, possibly stochastic, that describes the distribution of rewards that may be received by an agent taking action \(a \in A\) in state \(s \in S\). The function \(P : S \times S \times A \rightarrow [0, 1]\) is a probability mass function that describes the dynamics of the process and is often called the transition function. It is defined as

\[
P(s' | s, a) := Pr\{s_{t+1} = s' | s_t = s, a_t = a\}.
\] (2.1)

This function defines the distribution of succeeding states when the MDP is in state \(s_t\) and action \(a_t\) is enacted, with the subscript \(t\) denoting the current time step.

An MDP is called Markov because it has the Markov property, which means that all past states are irrelevant given the current one. This highlights a distinction between a state and the agent’s observation of that state. In domains with visual input, we often refer to the pixel data as the state, but technically this is only an observation of the state. The state itself may include information absent from the pixel data. For example, suppose an agent can recharge its health by returning to a particular spot and remaining there for five seconds. After the five seconds are up, the optimal action is to leave that spot to seek more reward; until then, the optimal action is to remain in place. Because time is not encoded in the visual input, these states appear the same to an agent that only has access to pixel data. To such an agent, it can never be clear which action is correct. The convention of stacking four game frames tries to alleviate this issue for short-term effects, by wrapping the immediate history into the current state, but it is not sufficient to make all states fully observable.

The Decision part of an MDP is due to the agent’s role in the process. At each time step \(t\), the environment presents the agent with a state \(s_t \in S\). The agent chooses an action
at ∈ A, and the environment sends the agent a reward signal \( r_t = R(s_t, a_t) \in \mathcal{R} \) and a new state \( s_{t+1} \) sampled according to the transition function \( P(s_{t+1}|s_t, a_t) \).

For example, in chess, the state is the current configuration of pieces on the game board with an indication of the player whose turn it is. The actions are legal moves. After the agent or player selects an action, the state of the game board will update according to the transition function, which is deterministic in this case. Rewards could be defined in different ways, but one example is a reward of +1 when the game is won, −1 when the game is lost, and 0 at all other time steps.

### 2.1.2 Policy and value functions

The agent’s goal when interacting with an MDP is to select actions to maximize reward. The reward function \( R \) and transition function \( P \) (Equation 2.1) define how the environment responds to each action, including the resulting reward distribution. But in general, the agent does not have access to these functions and does not know ahead of time the effect an action will have. Many algorithms, known as model-based, attempt to learn a model of the transition function \( P \), but it is not strictly necessary to do so. It is possible to learn correct behavior merely by observing how the environment responds to certain actions.

An agent learns by updating its policy in response to the reward signal. A policy is simply a function \( \pi : \mathcal{S} \rightarrow \mathcal{A} \) that takes in a state and spits out an action; it tells the agent what to do in each state. Since the environment has the Markov property, previous states are irrelevant to the policy; therefore only the current state goes into this function as an argument.

It is significant that the agent’s goal is concerned with the accumulation of reward over a long period of time. The total expected return is the discounted sum of rewards gained over time,

\[
G_t := r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^T r_T.
\]
The discount factor $\gamma \in (0, 1]$ reflects although we are interested in the total sum, we do care more about imminent rewards than highly delayed ones.

The value of a state is the total discounted reward expected to be gained by starting in that state and following a particular policy $\pi$. This is defined as

$$v_\pi(s) := \mathbb{E}[G_t | s_t = s] = \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} | s_t = s \right].$$

We can also define the value of taking a certain action in a given state. The value of a state-action pair under policy $\pi$ is

$$q_\pi(s, a) := \mathbb{E}[G_t | s_t = s, a_t = a] = \mathbb{E} \left[ \sum_{k=0}^{\infty} \gamma^k r_{t+1+k} | s_t = s, a_t = a \right].$$

It is possible for an action to bring low immediate reward, but still have a high value. In chess, this could correspond to sacrificing a piece for strategic advantage. Likewise, it is possible for a state to have high immediate reward, but low value. A chess example is making a capture that jeopardizes your king. An RL agent must average over many experiences to estimate the true value of a state. A unique aspect of reinforcement learning is learning to plan ahead and to consider how one action will affect the states and rewards available later.

### 2.1.3 Optimality and learning

Any policy that yields maximum return is called an optimal policy and is denoted by $\pi^*$. Here, optimal means that there does not exist a policy which an agent can follow to get greater return. There is always an optimal policy. It has associated optimal value functions.
The optimal state value function is defined as

\[ v^*(s) := \max_\pi v_\pi(s), \forall s \in S. \] (2.2)

The optimal state-action value function is defined as

\[ q^*(s, a) := \max_\pi q_\pi(s, a), \forall s \in S, \forall a \in A. \] (2.3)

Finding the optimal policy \( \pi^* \) is our ultimate goal in reinforcement learning. But if the optimal state value function \( v^* \) is known, the optimal policy follows directly as

\[ \pi^*(s) = \arg \max_a \sum_{s_{t+1}} p(s_{t+1}|s_t = s, a_t = a) v^*(s_{t+1}). \]

This says that the best action be chosen according to what is most likely to lead to a high-value state on the next time step. Having the optimal state-action value function makes this even easier, removing the need to look ahead to the next time step. Simply choose the action maximizing \( q^* \) for the current state:

\[ \pi^*(s) = \arg \max_a q^*(s, a). \]

So, to find the optimal policy, we wish first to find the optimal state or state-action value function.

The optimal value function reflects the value of each state or state-action pair under the optimal policy; but the optimal value function can be written recursively without reference to any policy at all. This recursive relationship is called the Bellman optimality equation.
For the state value function, this is

\[ v^*(s) = \max_a \mathbb{E}[r_{t+1} + \gamma v^*(s_{t+1}) | s_t = s, a_t = a] \]

\[ = \max_a \sum_{s', r} P(s', r | s, a) [r + \gamma v^*(s')] \].

And for the state-action value function \( q^* \), this is

\[ q^*(s, a) = \mathbb{E}[r_{t+1} + \gamma \max_{a'} q^*(s_{t+1}, a') | s_t = s, a_t = a] \]

\[ = \sum_{s', r} P(s', r | s, a) [r + \gamma \max_{a'} q^*(s', a')] \].

Each Bellman equation defines a system of equations, one for each state, which can be solved exactly if the state dynamics function \( P \) (Equation 2.1) is known. In reality, \( P \) is almost never known, and in most cases of interest the number of states is too great for this to be practical. Instead we wish to approximate the value function \( v^* \) or \( q^* \). Estimating the value function is the core of most reinforcement learning algorithms.

There are many ways to do this. We will briefly describe one method, called Q-learning, which is the algorithm we train our memory agents with.

Q-learning specifies a simple rule that uses an agent’s experiences to update \( Q \), an estimate of the optimal state-action value function \( q^* \). First a representation of \( Q \) is created. For small domains, this is a table with an entry for every state-action pair. An agent interacts with the environment by taking actions, and updates the values in the table based on the reward it experiences, as

\[ Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \left[ r + \gamma \max_{a'} Q(s', a') \right], \quad (2.4) \]

where \( 0 < \alpha < 1 \) is a step size or learning rate. Intuitively, an agent finding itself in a given state \( s \) chooses an action \( a \); if the resulting reward \( r \) plus the valuation of the succeeding state
is higher than expected, the agent increases its estimate of $q^*(s, a)$. If lower than expected, the estimate is reduced. This is described in Algorithm 2.1.1.

Q-learning was proved by Watkins [39] to converge under certain conditions. First, the world must be Markov and second, it must be stationary (that is, not change over time). The learning rate must decrease at an appropriate rate. And finally, every state-action value needs to continue to be updated. The first two conditions are usually assumed, and the third is easy to ensure. The last condition is usually met by designing an appropriate exploration policy.

Exploration is when an agent chooses an action that it believes is sub-optimal, in hopes of discovering higher-value areas of the state space. One common method of doing this is called $\epsilon$-greedy exploration, where the agent chooses a random action with probability $\epsilon$, or the best available (“greedy”) action with probability $1 - \epsilon$. The parameter $\epsilon$ is usually relatively small, often around 0.01 to 0.1.

Exploration is vital to learning. If an agent only does what it currently thinks best (exploitation), it misses out on discovering other more rewarding behaviors. But exploration comes with a cost, namely, reward lost to sub-optimal actions. Further discussion is out of our scope, but the tradeoff between exploration and exploitation is a matter of ongoing research.

<table>
<thead>
<tr>
<th>Algorithm 2.1.1: Q-learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set step size $\alpha \in (0, 1]$</td>
</tr>
<tr>
<td>Initialize $Q(s, a)$ arbitrarily $\forall s \in S, a \in A$, but $Q(terminal, \cdot) = 0$</td>
</tr>
<tr>
<td><strong>for each episode</strong> do</td>
</tr>
<tr>
<td>$s \leftarrow S_{initial}$</td>
</tr>
<tr>
<td><strong>while</strong> $s$ is not terminal <strong>do</strong></td>
</tr>
<tr>
<td>Select $a \in A$ according to policy</td>
</tr>
<tr>
<td>Observe reward $r = R(s, a)$ and new state $s'$</td>
</tr>
<tr>
<td>$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a')]$</td>
</tr>
<tr>
<td>$s \leftarrow s'$</td>
</tr>
<tr>
<td><strong>end</strong></td>
</tr>
<tr>
<td><strong>end</strong></td>
</tr>
</tbody>
</table>
2.1.4 Deep reinforcement learning and the Arcade Learning Environment

Directly learning a value function would require storing a table with an entry for every state or state-action pair. After initialization, the learning algorithm would update the value table until convergence. If the state and actions space are large, this becomes prohibitive not only in the amount of memory required to store the table, but in the amount of agent experience needed to thoroughly test every action and every state.

Instead, many applications rely on function approximation. The approximating function is typically designed to enable the agent to generalize, or apply what it has learned about one state to a distinct but similar state. Besides reducing the size of the function representation, this can accelerate learning, since using knowledge about one state to inform a belief about a similar state induces a useful bias.

Commonly, the value function is represented with a deep neural network. Deep neural networks (DNNs) are universal function approximators, and now have a substantial body of research theoretically and empirically supporting their role as a primary tool for machine learning, including reinforcement learning.

Q-learning with a DNN was shown to be unstable by Tsitsiklis and Roy [34]. Several factors contribute to this instability, including a shifting data distribution as the agent learns, correlations between successive observations during training, and the “bootstrap” method of the network producing its own update target (the part of Equation 2.4 in brackets). In the groundbreaking paper Human-level Control through Deep Reinforcement Learning, Mnih et al. [22] addressed these instabilities with two key innovations. First, they store historical transition data in a replay buffer, which is sampled randomly for training. This breaks the temporal correlations and smooths over the shifting data distribution. Second, they produce the update targets with a separate network. They called their model a Deep Q-Network, or DQN. With this Q-learning variant, the DQN achieved human-level performance across 49 Atari video games in the Arcade Learning Environment.
The Arcade Learning Environment or ALE has since become one of the foremost test-beds for reinforcement learning agents. A collection of classic video games, the ALE offers a variety of diverse tasks that are designed to be engaging and challenging to humans. The action space, from 3 to 18 actions depending on the game, is small enough to be computationally feasible but large enough to challenge reinforcement learning agents. The reward signal is clear and the games require a moderate degree of planning and strategy. This makes them worthy subjects for models that seek to emulate human behavior and learning.

Following the success of Mnih et al. [22] on the ALE, many researchers independently produced other algorithmic variations to further improve deep Q-learning, including Double Q-learning [36], Dueling Networks [37], Distributional RL [3], and Noisy Nets [7]. A notable paper took several of the most significant variations and combined them into one model, called Rainbow [15]. Rainbow’s mean performance far surpassed each individual model it was based on. At the time this research began, Rainbow held the state of the art on the Atari test suite; therefore it serves as our baseline, as well as the starting point for our models.

2.2 Auxiliary memory

In 2014, researchers from Google DeepMind, inspired by the LSTM [16], enhanced an ordinary recurrent neural network (RNN) by coupling it with an addressable external memory unit [10]. They drew an analogue between their model and a Von Neumann computer architecture, with the RNN performing the role of a CPU and the memory matrix that of RAM. Since this model is differentiable, it can be trained end-to-end with gradient descent. This means that instead of executing a rigid set of hand-crafted, possibly sub-optimal procedures, the model can learn from data to use its memory to solve tasks efficiently.

This model, called the Neural Turing Machine (NTM), learned to infer algorithms including sorting, copying, and associative recall, demonstrating that the RNN was successfully using its memory unit to count, to compare, and to store and recall knowledge. More than learning a map between an unsolved task and the representation of its solution, the NTM
learned the algorithmic steps to obtain the solution, empowering it to succeed even on examples far more complex than those it was trained on. By contrast, the baseline LSTM was slower, converged to higher cost, and was not shown to have learned an algorithm.

Since then, the concept of auxiliary memory has been modified, extended, and applied in many ways. Implementations are as diverse as the problems they are applied to. In this section, we will summarize the common defining features.

An Auxiliary Memory Unit (henceforth AMU), also called external memory, is a matrix, where the rows are treated as distinct memory slots or locations. In the most general sense, it comes with a function to write information (i.e. replace or update the contents of a row), and read or extract information, possibly using a key or query for content-addressability. A neural network has an AMU as a layer. The network learns what to write, and it uses what it reads in prediction.

It is called a memory unit because of the similarity to computer memory, in which data is written to a specific location, to be recalled on demand, and will persist until the same location is explicitly erased or written to again. It is external to the rest of the network in a few senses, including that each forward pass may only use a portion of it, and its size is independent of any other factor in the network. Indeed, depending on the implementation, the memory could grow in size while adding minimal computational overhead. These aspects will be discussed further later on.

Graves et al. [11] described some of the advantages of separating data storage from computation. In modern computers, this allows data to be stored reliably over the long term. Treating memory locations as variables lets the computer generalize by performing already-learned operations on unfamiliar data. In contrast, neural networks suffer the handicap of having memory and computation resources blended together in a single set of weights. Despite all their successes and well-earned hype, network-based models remain forgetful and slow to generalize.
Intuitively, the benefit that an AMU offers to a reinforcement learning agent comes from having access to information not present in the input data. For instance, states in which the agent confronts an enemy may appear the same regardless of how many times the enemy has been shot. In fact, if the enemy has already been shot twice and so is about to die, the state is much different from the state where he has not yet been shot. The former should have higher value because it is nearer the reward that comes when the enemy dies; yet an agent with access only to pixel input data will view these states as the same. External memory may help the agent distinguish between states with unobservable differences, and so help it to learn a more accurate value function.

External memory should also help RL agents adapt faster to changing circumstances. The write and read functions offer immediate storage and retrieval; once these functions are learned, an agent can react to new events much faster than with gradient propagation alone.

Figure 2.1 shows a high-level depiction of a generic neural network augmented with an AMU. After receiving an input, the network both writes to and reads from the memory. It then outputs a prediction based on the input and the information read from memory. While simple in principle, an auxiliary memory unit has a number of ambiguous properties that require careful thought and design. We describe some defining characteristics in the next section, but we will save descriptions of specific functions for chapters 4 - 6.
2.2.1 Defining features

There are several characteristic features of an AMU. Such a memory unit can in theory expand or shrink according to the current needs of the controlling neural network, without otherwise affecting the network’s behavior. According to Graves et al. [11], this is the primary sense in which the memory is \textit{external} to the network.

Another important feature is that the contents of it are stable, meaning information written to memory will remain available until the network overwrites the same memory location. That is, it will not be changed by gradient propagation or anything else. Also, by addressing the memory element-wise through a key or query, the network can retrieve specific information, ignoring what is irrelevant at the moment. Both these features stand in contrast to a regular fully-connected layer, where the all weights potentially change at every training step, and every weight contributes to every prediction.

Significantly, the network can still write to memory after training is over. This means that although the network weights no longer change, the model can continue to learn and adapt during evaluation or deployment. This is another sense in which the memory is external to the network.

The memory unit itself can be classified along two axes, according to Gülçehre et al. [12]. The first is whether it indiscriminately stores all facts, or learns a potentially more expressive mechanism for choosing what to write. The second axis distinguishes memory with continuous, differentiable addressing from that with discrete and non-differentiable addressing.

We add a third classification axis, which we will describe as episodic versus general memory. Episodic memory can be viewed as a form of short-term memory. It is erased at the end of each episode, therefore it stores only that which is relevant to a single current episode. For instance, some Atari games depend on relatively rare events to progress the game play. In Montezuma’s Revenge, the player must pick up a key to be able to enter a
door. Remembering that the key has been located would enable an agent to focus on the next task, but this memory should be discarded after the termination of the current episode.

Fortunato et al. [9] describe episodic memory as long-term. This is in contrast to an even shorter-term variety of memory which they call working memory. Despite that difference, our meaning of episodic memory is essentially similar to theirs. We call episodic memory short-term to contrast it with general memory, which deals with information that is relevant across episodes. This type of memory would be useful, for instance, to build a model of the world, or to keep track of the effect of each action on the game (move left, jump, etc.). For example, in the Atari game Gravitar, gravity pulls the player’s ship toward the bottom of the screen. At a certain point in the game, however, gravity reverses and pushes the ship away. To adapt to this switch, an ordinary neural network will have to remap every state to a new action through gradient propagation, a slow and inefficient process. In fact, assuming the states are visually identical, this may be impossible for an ordinary deep RL agent to learn, since it must continue to behave in the ordinary way on the earlier portion of the game. An agent storing variables in memory, by contrast, could adapt quickly—possibly with no gradient propagation whatsoever—by updating those memory values with an already-learned write function.

Episodic memory is most beneficial in games like Montezuma’s Revenge, where the agent must quickly react to uncommon events like finding a key. Importantly, general memory will be valuable even in scenarios that do not on the surface appear to “require” memory, such as purely reactive games like Space Invaders where everything relevant is on the screen. Even if the memory contains only the same information learned by a regular network layer, a massive memory layer can be accessed very quickly through sparse addressing, since not all the information is needed at every step. This can be faster than stacking layers of proportionate size, which means we get more model capacity for lower cost.

We built two different memory models for our four benchmark memory agents. Both use continuous addressing. One, based on the Differentiable Neural Computer [11], learns
a flexible expressive write mechanism, and is an example of episodic memory that is reset every episode. The other, which uses product key memory [20], learns general memory, and subverts the classification scheme of Gülçehre et al. [12] by learning memory values through backpropagation. These two memory models will be discussed further in chapter 3.

2.2.2 Comparison—LSTM

It is important to compare the auxiliary memory unit to a closely related model, the LSTM [16]. LSTMs (Long Short-Term Memory) are inspired by a similar principle, the desire to write to and read from a vector that otherwise remains the same from time step to time step. In fact, an auxiliary memory unit can be considered a generalization or extension of the LSTM.

The most obvious difference is the LSTM has a state vector, while the AMU is a two-dimensional matrix. The LSTM state vector is analogous to a single row in the AMU; yet an AMU is not simply an LSTM with a two-dimensional state.

For one thing, the LSTM cell passes the entire state vector through a fully connected layer to produce the output. The AMU is content- or location-addressable, and so the network can selectively read only the most relevant information to produce the output.

Writing to the state or memory is similar in both models, but differences arise from the different memory shapes. At each time step $t$, the LSTM uses the input $x_t$ and the hidden state $h_{t-1}$ (previous cell output) to compute a forget gate $f$, an input gate $i$, and a candidate write vector $w$. Then to update the state vector $C$, we compute

$$C ← f \odot C + i \odot w,$$

where $\odot$ is the Hadamard product. With an AMU, an arbitrary neural network produces the necessary interface gates from $x_t$, using the previous cell output only if it happens to be a recurrent neural network. The details of the write function depend on implementation, but
the magnitude of writing and erasing are usually linked through one weighting vector, and
each memory location is written to a different degree. So in the Neural Turing Machine [10],
for example, a single write weighting $s$ is produced, with one element per memory location
(row). Then the network creates a write vector $w$, which will be added, and an erase vector $e$, which will be subtracted. The erase vector is related to the forget gate from the LSTM; essentially $e \equiv 1 - f$. Each memory vector $M_i$ is then updated as follows:

$$M_i \leftarrow M_i + s_i[w - e \odot M_i].$$

Note that although the same write and erase vectors are applied to every memory location,
it varies according to the magnitude of the weighting at that location. This keeps the model
differentiable all the way through.

The most crucial difference between the two approaches is that the state vector of the
LSTM is not external to the rest of the network, in the same way that the memory matrix is
with the AMU. With an AMU, behavior of the network is independent of memory size as
long as the memory is not full [11]. Moreover, it is possible to design an AMU so that it only
accesses a few memory locations at a time. This means that the AMU can in principle grow
or shrink according to the needs of the controlling network, without necessitating any other
change, as discussed earlier. The practical benefit of this is that we can make a huge memory
layer with only a minor increase in computational complexity [20].

2.2.3 Comparison—replay buffer

We must also clarify the difference between the AMU and the replay buffer, sometimes
called memory, commonly found in models fashioned after Deep Q-Network (DQN). While
the replay buffer may be considered a type of AMU, according to our general description,
there are a number of crucial differences. In DQN-type models, the replay buffer stores
training data, tuples consisting of historical state transitions with the corresponding action
and reward. This data is sampled randomly for training, which helps to reduce instability by breaking temporal relationships in the data and smoothing a shifting data distribution. Our version of the AMU, on the other hand, contains not training data but information to supplement the input to the network. That is the first major difference. The second is that the replay buffer is not content-addressable, as the AMU is. A third is that the network samples the replay buffer only during optimization, not during evaluation. In contrast, our AMU provides specific information that the network requests when estimating the value of a given state-action pair, both in training and in evaluation. This highlights perhaps the most subtle difference: the AMU is controlled directly by the network itself. But the replay buffer is a completely separate entity that merely stores and samples training data.
Chapter 3

Related Work

We turn our attention to other research on auxiliary memory models, beginning with the Differentiable Neural Computer by which most memory models have been inspired. Next, we review related research in maze navigation, followed by language tasks, notably including the product-key memory model. We then cover the prior work on our test domain, the Arcade Learning Environment. Finally, we touch on some related ideas in meta-learning.

3.1 The Differentiable Neural Computer

In section 2.2, we mentioned the genesis of auxiliary memory with the Neural Turing Machine [10], or NTM. The NTM demonstrated an exciting ability to learn algorithms, but it had a few drawbacks. The network used a version of sequential indexing; that is, it stored memory sequences in contiguous blocks. However, it had no way to keep different memory blocks from overlapping and overwriting each other; and, when the read head did jump to a new location, there was no way to track where it had been before. Also, there was no way to free memory locations independent of the erase vector used when writing.

Two years after the NTM, Graves et al. [11] addressed these issues by extending the concept of memory, creating an external memory unit that could be paired with an arbitrary neural network. They added free gates to deallocate memory, and a temporal link matrix to track write sequences. The network selected new write locations by soft attention over all memory rows. This model, termed a Differentiable Neural Computer (DNC), could create
and traverse its own data structures. With this capability, it succeeded in problems as diverse as question answering, graph navigation, and a simple reinforcement learning block puzzle.

The DNC and its experiments on the RL block environment are one of the primary inspirations for this work. The ability to encode and retrieve arbitrary information immediately should be a powerful aid in reinforcement learning. As the remainder of this chapter will explain, there has been a fair amount of research in that area since then, but mostly in navigation tasks. A few researchers have used a version of memory to learn Atari video games, but none have the degree of flexibility provided by the DNC, with dynamic memory allocation, the temporal link matrix, and multiple read modes. Our desire was to apply this powerful memory model to the Arcade Learning Environment, with the hypothesis that doing so would improve sample efficiency.

Two of our models contain a memory modeled after the DNC. Details will be given in chapter 5.

3.2 Navigation with memory

Since the DNC, research on reinforcement learning with auxiliary memory has progressed slowly. Much of the research has emphasized learning to explore 2- and 3-dimensional mazes. A prominent example is the Neural Map of Parisotto and Salakhutdinov [25] which applies a DNC-style model to agents learning to navigate mazes. Their agent’s memory is endowed with special features that suit it particularly well for tasks where navigation is a large component. These include an inductive bias on the write operation, a one-to-one correspondence between memory locations and physical locations in the 3D map, and a sparse write operator that avoids too frequently overwriting memory locations. The final trained agent completed search tasks in 2D and 3D environments far better than a simple LSTM-based agent, showing that their memory-equipped model remembered the given objective better while exploring brand new environments. One drawback to their approach is that the memory update process simply replaces the content of a single memory location with the new write candidate. This
is feasible for their situation, where each memory location corresponds with a physical map location, and the agent’s physical location is always known. For a more general purpose memory unit, a soft memory update like DNC [11] will be needed.

A similar example is the Memory Augmented Control Network of Khan et al. [19], which tackles path planning in partially observable environments by splitting planning into local and global levels. In this work, the network uses the memory matrix to assemble the local policies into a global policy. They argue that although LSTMs are theoretically capable of recalling and generating arbitrary sequences, this is often difficult to put into practice. Basing their work on the DNC, they train an agent to efficiently navigate new environments with sparse rewards.

Oh et al. [23] test more structured goal-oriented navigation tasks, in a Minecraft environment. They conclude that memory-based architectures generalize to unseen maps better than other traditional architectures. However, as Parisotto and Salakhutdinov [25] observe, the proposed memory units of Oh et al. [23] are restricted to remembering only the last $k$ observations, whereas it could be valuable for the network to choose what to remember.

Navigation tasks provide a good opportunity for memory agents to build a mental map, remember information about the state of the environment, and plan ahead. But they lack the degree of diversity of Atari video games. Atari games generally have a larger state space, require finer control, and involve more complex planning and strategy. The Arcade Learning Environment has become something of the CIFAR-100 of the RL world, and we wanted to see how an AMU could help an agent on that domain.

### 3.3 Language with memory

Augmented memory models shine not only in reinforcement learning, but in language processing tasks as well. Weston et al. [40] tested a memory network on a large-scale question answering task. Their network proved it could remember information from previous sentences, and chain together facts to make inferences about the intention of ambiguous words. In
Suzgun et al. [32], a memory augmented model based on the Neural Turing Machine learned to recognize hierarchical Dyck languages.

A third example, which bears particular relevance to this work, is the product key memory of Lample et al. [20]. This memory exploits the cross product of two sets of keys to efficiently query a much larger set of values. Interestingly, the product key memory (PKM) model lacks a write function. The network initializes the keys randomly at the beginning of training, and learns the memory values through backpropagation, like the rest of the network. The resulting instability of the memory values, and the inability to quickly write specific information, make this a much less flexible memory design than writable AMUs like the DNC. But it still provides the great advantage of increasing model capacity with only minor computational overhead. In addition, the layer is read sparsely, meaning only the most relevant weights contribute to any given prediction.

Lample et al. [20] tested the PKM model on a massive language task (30 billion words) and found that their twelve-layer memory model outperformed a twenty-four layer transformer architecture, yet was twice as fast. This clearly shows memory’s potential to increase learning efficiency. The PKM serves as the memory layer in two of our models. More details of this memory model will be given in Chapter 6.

3.4 Atari with memory

Since DQN, the Arcade Learning Environment has remained a favorite for RL research. In section 2.1.4 we summarized only a few of the papers that have contributed to RL’s ever-expanding success on the Atari test set. Every new paper pushes performance just a little further.

Despite all these improvements, sample efficiency remains poor when compared to human learning. Final performance does indeed surpass that of humans on many of the games (though this is disputed by Toromanoff et al. [33]), but it is problematic that it should take so long to reach even human-level performance. It is true that humans have years of
prior knowledge, experience, and beliefs which they bring to bear on any new problem faced. For instance, using prior game experience and existing object segmentation skills, a human can quickly identify what pixels on the screen constitute his avatar, platforms he or she can stand on, and objects or other agents that may be harmful. Furthermore, once the object under the player’s control is identified, a human player probably has a good idea of what effect each control will have on that object (such as moving to the left or right, jumping, or firing). An RL agent has to learn all of this from scratch, for every individual game it is trained to play.

However, even if a human’s beliefs about, say, the controls should prove wrong, it only takes a few trials to figure out what each button does and to remember it. This is the type of resilience and adaptability we would like to see in RL agents. The ability to learn from a few examples is a well-studied field in its own right; an overview of some relevant work is given in the next section.

Previous RL agents were typically appraised by comparing final trained performance to a human baseline. We are more concerned with learning rate than with final performance, although it seems reasonable to hope that an agent that can learn more efficiently will be better at exploiting the environment to achieve higher reward. But while a fresh computer agent necessarily begins training on a new task with no prior knowledge, it is impossible to separate a human from his prior knowledge and so it may be unfair to compare an RL agent’s learning speed to a human’s. Therefore we are interested in comparing learning efficiency only to previous computer models, not to humans.

We are aware of only a limited amount of work on RL agents with AMUs learning to play Atari games. Blundell et al. [5] tested a model that simply stored in a table the highest reward ever obtained in each state $s$ by taking action $a$. By storing the highest reward rather than average reward, they hoped to exploit the fact that actions in most Atari games yield deterministic reward; that is, taking action $a$ in state $s$ will have the same effect every time. This model outperformed DQN and Prioritised DQN on four out of five Atari games tested.
Several of the same authors went on to develop this model further. In Pritzel et al. [26], both keys and values of the memory are updated through gradient descent, as well as through direct modification by the location-based write operation. This slightly confusing development was not thoroughly justified, but the authors claim great improvements in data efficiency over DQN and several other models.

Both the previous models came out before Rainbow, but the foremost reason to continue working in this direction is that those memory models were highly restrictive in what can be learned. They reduce the problem nearly to tabular Q-learning; and while a table can admittedly be termed a sort of memory, we do not expect a table-based memory to furnish the same benefits as the flexible writable memory described earlier. Our work constructs an agent with a general memory, modeled after the DNC with general read and write functions.

3.5 Learning to learn

Learning to learn, or meta-learning, is the concept that an appropriately designed model may become more proficient at solving new tasks as it trains. This implies that the amount of data required to learn each new task decreases over time. The idea is closely tied to that of transfer learning, since some skills learned on earlier tasks are implicitly transferred when the agent switches to a new task. In particular, the skills transferred are the skills surrounding the ability to learn itself; this is what merits the meta- prefix. In other words, an agent training on a task is not only learning how to solve that task, but it is learning skills that will enable it to learn the next task even faster. It is learning how to learn. For example, Hochreiter et al. [17] trained an LSTM model to learn new quadratic functions with very few data samples.

Auxiliary memory connects directly to this field of study. An agent equipped with an AMU with well-trained write and read operations can store and retrieve information quickly in the short term [26], even without gradient propagation. This enables it to adapt and react
faster to new stimuli. Of course, a brand new untrained agent does not yet have a functioning memory unit. A feature of most meta-learning efforts is a hierarchy of patterns to learn. In our case, the agent depends on backpropagation over long time scales to learn how to interact with its memory unit; as reading and writing skills improve, it begins to rely the memory for rapid adaptation when faced with new data, accelerating learning on specific tasks. Due to this double time-scale aspect of meta-learning, it may actually be longer before an agent reaches a given level of performance. But if it effectively learned how to learn, it will have stronger generalization and adaptation skills.

Santoro et al. [28] give an excellent presentation of these ideas, and argue convincingly that memory-augmented agents are uniquely suited to learning few-shot tasks. They argue that a scalable solution to the meta-learning problem must require memory storage that is persistent and selectively addressable, and the memory must be external to the rest of the network, meaning the number of parameters is independent of memory size. These are attributes of memory networks such as the Neural Turing Machine and the Differentiable Neural Computer, in contrast to LSTMs, which do not meet these requirements.

The Memory Augmented Neural Network of Santoro et al. [28] successfully classified novel Omniglot samples after only a few examples. This kind of flexibility would greatly benefit Atari-playing RL agents, for example, in quickly identifying new sources of reward or new hazards.

In summary, we do not want the weights of the network to learn to play the game. We want the network to learn how to extract relevant information from the state, but some predictive model constructed in the memory is used to select actions. That is, we don’t want to learn a model of the game, we want to learn a model for learning games. This will help to achieve the end goal of greater data efficiency.

In this work we do not train any of our models across multiple Atari games. However, given the complex and often progressive nature of many individual Atari games, the principles of meta-learning will still benefit an agent with an effective AMU in learning a single game.
For example, having learned that objects that pursue it generally bring negative reward, an agent encountering a visually new type of enemy may require only a few interactions before updating its policy to avoid that enemy.
Chapter 4

Model Overview

Our main hypothesis was that an RL agent with an AMU would use its rapid store and recall functions to adapt and react faster, thereby increasing sample efficiency. In designing such an agent, we read dozens of relevant papers promising performance benefits from their new training technique or architectural variation or exploration policy. In our work, we did not attempt to synthesize every optimal improvement from every paper. Our baseline is Rainbow DQN; so for a fair comparison, we only wanted to add a memory layer to Rainbow while changing as little else as possible.

Each of our models extends Rainbow DQN with a memory unit. We designed and tested two different AMUs, with two variations of each, for a total of four models. We refer to the network architecture surrounding the AMU as the controller, after the pattern of Graves et al. [11] and others. In this chapter, we describe the controller network and aspects of memory common to all four models. Details of our AMUs will follow in the next two chapters.

Our code is based on the Rainbow implementation by Arulkumaran et al. [2].

4.1 Controller network

Rainbow itself is the combination of improvements from many different papers, yet most of the architecture is the same as the original DQN. The input to the network is four stacked game frames, with other preprocessing steps applied as described in section 4.3. The processed input passes through three layers of convolutions, with a rectified linear unit after each. This
Figure 4.1: A simplified outline of the hybrid memory Rainbow neural network. We have denoted convolution as $\ast$, a fully connected layer by $\otimes$, element-wise addition by $\oplus$, and the sigmoid function with $\sigma$. The dotted line spanning the memory unit denotes the optional residual connection. Numbers of convolutions per layer and scale of embeddings may be inaccurate. See Hessel et al. [15] for additional details.

is followed by a fully connected layer. In the original DQN, a final fully connected layer produces the outputs, one per action. The outputs are interpreted as action probabilities, reflecting the estimated Q-value of taking each action in the current input state.

One of the improvements assimilated by Rainbow was a dueling network architecture [38]. Instead of the final fully connected layer, the network is split by two separate fully connected layers into two heads, one stream computing a value estimate and one the advantage. These two streams are then merged by a special function. See Wang et al. [38] for more details. This is further adapted by Rainbow to learn return distributions as in Bellemare et al. [4]. For more information, see Hessel et al. [15].

The other modifications Rainbow included are noisy nets [8], double Q-learning [13], prioritized replay [29], and multi-step Q-learning [31]. These will not be described here; we refer the reader to each original paper or to Hessel et al. [15] for a summary. Our models retain every improvement aggregated by Rainbow.

Our addition to the architecture is the auxiliary memory unit or AMU. Figure 4.1 shows a simplified schematic of our hybrid memory architecture. We want the network to read from and write to memory after extracting the feature embedding, but before computing
a Q-value estimate, so we positioned our AMU between the final convolution layer and the advantage-value split.

Table 4.1 summarizes the hyper-parameters governing the Rainbow architecture.

<table>
<thead>
<tr>
<th>Hyper-parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channels</td>
<td>32, 64, 64</td>
</tr>
<tr>
<td>Filter size</td>
<td>$8 \times 8, 4 \times 4, 3 \times 3$</td>
</tr>
<tr>
<td>Stride</td>
<td>4, 2, 1</td>
</tr>
<tr>
<td>Embedding (hidden) size</td>
<td>512</td>
</tr>
<tr>
<td>Distributional atoms</td>
<td>51</td>
</tr>
<tr>
<td>Value stream hidden size</td>
<td>512</td>
</tr>
<tr>
<td>Value stream output size</td>
<td># atoms</td>
</tr>
<tr>
<td>Advantage stream hidden size</td>
<td>512</td>
</tr>
<tr>
<td>Advantage stream output size</td>
<td># atoms $\times$ # actions</td>
</tr>
<tr>
<td>Output size</td>
<td>Number of actions</td>
</tr>
</tbody>
</table>

Table 4.1: Q-network architectural hyper-parameters from Rainbow.

4.1.1 Loss

Rainbow’s loss function is a multi-step variant of the distributional loss [4]. First we define a return distribution $d_t := (z, p_\theta(s_t, a_t))$, where $z$ is a discretized support vector containing $n$ “atoms”. The parameters $\theta$ are updated so that $d_t$ will approximate a target distribution $d_t^{(n)}$ based on actual returns,

\[
d_t^{(n)} := (G_t^{(n)} + \gamma_t z, p_\bar{\theta}(s_{t+n}, a_{t+n}^*))\]

where $G_t^{(n)}$ is the discounted sum of $n$ rewards, and $a_{t+n}^*$ is the greedy action in state $s_{t+1}$. The parameters $\bar{\theta}$ are the parameters of the frozen target network that produces the update targets, as in DQN.

The loss is then defined as the KL divergence between these two distributions,

\[
D_{\text{KL}}(\phi_z d_t^{(n)} || d_t),
\]
where $\phi_z$ is the projection onto $z$. For full details, refer to Hessel et al. [15].

4.2 Memory unit

The focus of our research was the Auxiliary Memory Unit at the center of the model. Here we describe in more detail the key features of the AMU. Some specifics are saved for subsequent chapters.

4.2.1 Definitions and notation

First we define a few terms. This notation will apply through the next few chapters. We will refer to the actual memory matrix as $M$. It has $d_M$ rows. In some designs, the entire row $M_i$ represents a value; in other cases, it is divided into a key and a value as $M_i = [k_i, v_i]$. We refer to the dimension of the value by $d_v$ and the key by $d_k$. The set of all keys will be represented by the $d_M \times d_k$ matrix $K$, and the set of all values by the $d_M \times d_v$ matrix $V$.

Note that if the memory does not employ keys, then $V$ is exactly equivalent to $M$.

A neural network interacts with an AMU through write (input) and read (output) operations. The output of the read operation is called a read vector; similarly, the input to the write operation is called a write vector. Both operations depend on a query vector, with which the network determines which memory locations are to be read from or written to.

We will discuss these functions further in the following sections.

4.2.2 Indexing

Write and read functions both depend on indexing into the memory. We want to find the memory location(s) most similar to some query.

Some memory implementations simply index discretely by location. But, in general, it is unlikely that a specific query will be found in the memory, and it is desirable to keep the operation differentiable; therefore, most implementations employ a content-addressing mechanism using attention over all locations. We follow this precedent in our models. This
attention mechanism returns a weighting over memory locations, where each weight is between 0 and 1 and the weights sum to 1. This weighting is viewed as a probability distribution over memory locations, with a high value indicating a high degree of similarity to the query.

Given a set of keys $\mathcal{K}$ and a query $q \in \mathbb{R}^{d_q}$, where $d_q = d_k$, we wish to produce a normalized distribution $s$ of weights describing the similarity of the query with each key.

First $q$ is compared to each item in $\mathcal{K}$ using a positive definite kernel $\kappa$ of some sort. Common choices for the kernel are cosine similarity, inverse distance, or inner product. For simplicity, we will define the kernel to permit broadcasting over a matrix, that is,

$$\kappa(\mathcal{K}, q) := [\kappa(K_i, q)]_{i=1}^{d_M}.$$ 

Applying $\kappa$ to $\mathcal{K}$ and $q$ gives us a vector of scores, $\hat{s} := \kappa(\mathcal{K}, q)$, which are passed through a softmax function to produce a probability distribution over memory locations:

$$s := \sigma(\hat{s}) = \left[ \frac{e^{\hat{s}_i}}{\sum_j e^{\hat{s}_j}} \right]_{i=1}^{d_M} \quad (4.1)$$

In some memory models, the memory consists only of values and not key-value pairs. In this case, the query $q$ is of size $d_v$, and the kernel is applied directly to the entire memory row, giving $\hat{s} := \kappa(\mathcal{V}, q)$. These scores are softmaxed as before to yield $s$.

When specifically referring to the query and weight vectors for the read operation, we will write $q^r$ and $s^r$. Similarly, the query and weight vectors for the write operation will be denoted $q^w$ and $s^w$.

### 4.2.3 The read operation

The goal of the read operation is to retrieve a memory, prompted by a query.
With the attention scores $s^r$ from the previous section, we can compute a weighted average over the memory values. This produces the read output vector

$$ r := \mathcal{V}^T s^r \in \mathbb{R}^{d_v}.$$

While every memory location contributes to the read output, this vector most closely resembles the values whose keys were nearest the query due to the weights $s^r$.

One can vary the number of read “heads” by repeating the operation with multiple queries $\{q_i^r\}_{i=1}^{n}$. This returns $n$ read vectors,

$$ r_i := \mathcal{V}^T s_i^r,$$

where $s_i^r := \sigma(\kappa(K, q_i^r))$. These can then be summed, averaged, or used independently as desired.

It is also possible to execute a global read on the entire memory, by passing $\mathcal{V}$ through a linear transformation or MLP. This is impractical for large memory layers due to computational costs, and we do not include this read mode in our models.

4.2.4 The write operation

The write operation is more complex, and there is an array of different approaches. These can be broadly grouped by what is written and how. Because these groups are so broad, we will not define specific functions yet.

In the simplest case, there is no write operation and the contents of the memory are learned through backpropagation, the same as the rest of the network. This does not relegate the memory to the status of an ordinary fully connected layer, because it is still accessed only through the read function. This allows sparse access to a potentially huge layer, giving a benefit of speed and minimizing interference from irrelevant information that may be contained in the memory layer. It does however miss out on the near-immediate memory
updates that a write function can provide. The memory unit of Lample et al. [20] is of this type.

In the second category, typically described as cyclic, the write function simply appends to the memory at every timestep. If the memory is full, it overwrites either the oldest or the least recently accessed item. Within this group of functions, the network can either learn what to write, or be explicitly programmed. In the latter case, the write vector is often the same as the query from the read operation, or some map between a state representation and a value estimate [5, 26]. One example of this type of memory is that of Oh et al. [23], which simply stored the most recent \( M \) observation encodings. While the simplicity of this approach is appealing, [25, 41] noted it restricts the space of things the network can learn.

In the third and final category, the network learns what to write, and it learns how to write it instead of simply overwriting the oldest or least-used item. This is the most complex approach and offers the most power and flexibility. As with the read operation, the network first computes a vector of weightings over memory locations. These may be based on current allocation as well as memory content. Then network produces an erase vector which is used to replace memory content with the candidate write vector, according to the weightings. This is the approach of the Differentiable Neural Computer [11].

For our four models, we built two different AMUs, one from the first category and one from the third. Their write functions will be described in the next two chapters.

4.3 Preprocessing

We employ the same preprocessing steps as Rainbow and DQN to reduce the amount of computation required. First, the input game frames are reduced from three channels (RGB) to one, and downsampled from \( 210 \times 160 \) to \( 84 \times 84 \). The four latest frames are stacked to provide input to the agent. Mnih et al. [22] argue that repeating an agent’s action for \( k \) frames reduces computational load by a factor of \( k \), since it is much cheaper to run the emulator than for the agent to select actions. This means an agent can play roughly \( k \) times
more games within a given amount of time. We follow DQN and Rainbow in repeating the last action \( k = 4 \) times.

Finally, we terminate episodes after 108 thousand steps. This is because in some games it is possible to engage in behavior which will indefinitely postpone the natural termination of the episode. For instance, while playing Breakout, some of our agents learned to never launch the last ball, perhaps preferring to remain in a state with positive potential future reward rather than seek to obtain that reward but risk ending the episode. While such behavior is interesting, we do not have the computational resources needed to train agents arbitrarily long. Toromanoff et al. [33] suggest terminating after a certain amount of time passes with no reward, which would avoid cutting off agents that are doing well. However, we keep to the conventions followed by our baseline, Rainbow.

These preprocessing steps are summarized in table 4.2.

<table>
<thead>
<tr>
<th>Hyper-parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grayscale</td>
<td>True</td>
</tr>
<tr>
<td>Observation downsample size</td>
<td>84 × 84</td>
</tr>
<tr>
<td>Stacked frames</td>
<td>4</td>
</tr>
<tr>
<td>Action repetitions</td>
<td>4</td>
</tr>
<tr>
<td>Reward clipping</td>
<td>[-1, 1]</td>
</tr>
<tr>
<td>Terminal on loss of life</td>
<td>True</td>
</tr>
<tr>
<td>Max frames per episode</td>
<td>108K</td>
</tr>
</tbody>
</table>

Table 4.2: Preprocessing parameters inherited from DQN via Rainbow.
Chapter 5

Models 1 and 2: Differentiable Neural Computer

Seeking maximum read and write flexibility, we fashioned our first two models after the Differentiable Neural Computer (DNC). The DNC itself was tested on a simple reinforcement learning problem. Inspired by this, we wanted to equip Rainbow DQN with a DNC-style memory and solve a more complicated task.

5.1 Model description

The original DNC was tested on a basic RL problem that involved moving a maximum of six two-dimensional blocks into a goal configuration. It was a non-visual task, meaning the environment state and goals were described with a prescribed ad-hoc vector representation rather than pixel data.

Learning to play Atari games from pixel data is a harder task, and we made a number of adjustments to the DNC architecture, as we incorporated it into a larger agent and trained with Q-learning. First, the controlling network, an RNN in the original paper, is replaced with Rainbow. The output of Rainbow’s convolutions is passed via a linear layer as the input to the DNC memory, and then the output of the DNC memory is returned to Rainbow to compute the final prediction.

The other modifications we made were necessary because Rainbow is trained offline, while DNC is trained online. An online algorithm takes data a piece at a time as it is made available, and after that datum is used to update the model parameters, it is discarded. An online RL agent trains in real time as it interacts with the environment. For its RL
block task, the Differentiable Neural Computer was trained online using a policy-gradient algorithm. On the other hand, an offline algorithm has another source of data on which to train, separate from real-time experience. In the case of deep Q-learning, this is the replay buffer. It may be argued that deep Q-learning is not strictly offline since the buffer is continually being washed out with new experience, but it is sufficiently offline to introduce a few more complications to our idea of memory.

The first of these complications pertains to the read vectors. In the original DNC, the vectors read at time step $t$ are appended to the input at time step $t + 1$. With offline Q-learning, this does not make sense, because training data is sampled randomly and so the context of one time step has no relevance to the next. In our version, the read vectors are handed back to the controlling network, where they supplement the input as the network computes a prediction for the current time step.

The randomly-sampled training data also interferes with the process of the temporal link matrix in the DNC. This matrix tracked the order that memory locations were written to, and played a big role in how the DNC learned traversable data structures. We hoped that learned data structures would benefit agents playing Atari, but there are two reasons it was not practical to keep the temporal link matrix. The first is that we need a larger memory to learn more complex tasks like Atari video games, but the cost of storing and computing the link matrix is $O(d_M^2)$, which grows prohibitively with the number of memory locations. The second is again due to our offline training. When training on randomly sampled data, tracking the order of memory writes is meaningless since there is no connection between what is being written. Because of these two issues, we gave up our hope of using the temporal link matrix and cut it out completely. Therefore, we read only with the content-matching mode, not the forward and backward modes.

Finally, the difference in batch size complicates the write operation. An online algorithm like DNC has an effective batch size of one, but Rainbow and our derivative models are trained with mini-batches of size 32. The write operation does not have an obvious
extension for batch updates: normally a batch of inputs produces a batch of outputs, but in this case, the output of the write operation is a single updated memory matrix, so the batch elements would have to be combined together in some way. One answer is to perform the write updates sequentially. The problem with this is that the contents of the memory will change with each write, and then the interface parameters for succeeding writes will no longer be relevant to the current state of the memory. This results in the output being dependent on the order of the writes, which is undesirable in a batch operation. Another solution, which we employ for our models, is to perform each write operation on its own copy of the memory, and then average the results back into one memory. Further research could establish a more efficient and more sensible way to handle this issue.

We now describe the read and write functions as implemented in our version of DNC. Both functions index with a differentiable attention mechanism over all memory locations, as described in section 4.2.2. A unique feature of the DNC is that each memory location is fully occupied by the value, rather than being split into a key-value pair. That is, $M = V$, and indexing is performed directly over values.

For the read operation, we have the attention weights $s^r_i := \sigma(\kappa(V, q^r_i))$. As before, $q^r_i$ is the query for the $i$th read head, and $\sigma$ is the logistic sigmoid function (Equation 4.1), normalizing the scores into a probability distribution over locations. In this model, the similarity measure $\kappa$ is cosine similarity:

$$\kappa(V, q) := \left[ \frac{V^T q}{\| V_i \| \| q \|} \right]_{i=1}^{d_M}. $$

To produce a read output vector, the memory values are simply averaged according to the attention weights, giving $r_i := V^T s^r_i$. Thus every memory location contributes to each read operation, but the locations that are most similar to the query contribute the most. As mentioned before, the original DNC had two additional read modes employing the temporal
Figure 5.1: A depiction of the DNC read and write process. The write query is used to select locations for writing, which are updated with the write and erase vectors. The read vectors are produced by averaging over memory locations weighted by similarity to the read vectors.

link matrix to read backward and forward in time; since we cut out the link matrix, we only read with the mode described here.

For the write operation, the weight vector $s^w$ depends not only on content-matching using the kernel, but on an allocation system which tracks the usage of each memory location. For the content-matching weights, we have, based on a write query $q^w$,

$$s^{w'} := \sigma(\kappa(V, q^w)) \in \mathbb{R}^{d_M}.$$

A thorough discussion of the usage and allocation vectors is beyond the scope of this overview. Details can be found in [11]. The idea is to create a sense of memory locations being full or empty, and consequently available or unavailable for subsequent write operations. The usage vector $u \in \mathbb{R}^{d_M}$ is defined so that the more a memory location is written to, the more its usage increases, up to a usage of 1. Usage can only be decreased, down to 0, through free gates emitted by the controller network. Then the allocation vector $a \in \mathbb{R}^{d_M}$ is computed using $u$, and determines which locations will be available for writing. The write weighting $s^w$ is an interpolation between allocation and the content weights $s^{w'}$,

$$s^w := g [ca + (1 - c)s^{w'}].$$
The parameters $g$ and $c$ are produced by the controller network. The write gate $g \in [0, 1]$ controls the overall strength of the write operation, regardless of the other write parameters. The parameter $c$ simply determines the interpolation. The network also produces a candidate write vector $w \in \mathbb{R}^{d_c}$ and an erase vector $e \in [0, 1]^{d_c}$.

Equation 5.1 shows the update step for the memory. $J_{d_M,d_v}$ denotes the $d_M \times d_v$ matrix of ones, and $\odot$ is the Hadamard product. Intuitively, each memory row $i$ is reduced by an amount proportional to itself, the erase vector, and the weight for that row; the write vector is then added to each row, again in proportion to the weight.

$$M \leftarrow M \odot (J_{d_M,d_v} - s^w e^T) + s^w w^T \quad (5.1)$$

Our two DNC models differ by a single residual connection across the memory layer. Given an input tensor $x$, let $y = f(x)$ denote the output of the portion of the network prior to the memory layer, and let $h(\cdot)$ be the memory layer and $g(\cdot)$ the remainder of the network. Then Model 1, also called DNC-noRes, is defined to be simply $g(h(y))$. Model 2 incorporates the residual connection and is called DNC-res. It is defined as $g(h(y) + y)$, where the addition is element-wise. In Figure 4.1, this is depicted by the dotted line crossing the memory.

Our DNC code is a modification of the implementation by Ixaxaar et al. [18].

<table>
<thead>
<tr>
<th>Hyper-parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory size</td>
<td>4096</td>
</tr>
<tr>
<td>Cell size</td>
<td>64</td>
</tr>
<tr>
<td>Read heads</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 5.1: Hyper-parameters for the DNC memory layer. The DNC does not have distinct keys and values, but each memory location is filled by a single cell. Because each pass through the network involves the entire memory matrix, computational constraints limit the number and size of memory cells.
5.2 Conceptual difficulties

As we built and tested this model, we became aware of several intriguing issues. One was the complications with the temporal link matrix, as already discussed in section 5.1. Another was the computational cost of querying and updating this type of memory unit. Since all entries in the memory contribute to every memory operation, the memory cannot be that large before experiencing massive slowdowns. The memory unit of the DNC was quite small, containing only 100 vectors of size 32. We built ours substantially larger to accommodate the additional complexity of learning from pixel data, but this became very expensive to train. Without a good way to reason about what the optimal size of memory is for a given task, we chose a size as large as possible while still being feasible to train.

Other issues were related to a conflict between the purpose of the memory and the offline nature of the deep Q-learning algorithm that trains the network. First, as discussed in the last section, it is unclear how to execute a batch write to memory, as multiple writes cannot occur at once, and the end result will depend strongly on order if they are done sequentially. Of yet greater significance is the question of whether a write update should occur during training at all. In section 2.2.1 we discussed how an episodic memory like the DNC’s needs to be erased at the end of each episode, since the things written during an episode will have no relevance once a new episode starts. Similarly, when training on a batch of randomly sampled historical data, the inputs to the memory will have no relevance to the episode the agent is currently experiencing, so it might not make sense to perform a write update. Yet the agent must write during training, or else the lack of gradient flow will result in the agent not learning how to write at all. These issues all stem from the memory relying on temporal connections between samples of experience to store meaningful things, but other aspects of the algorithm seeking to break temporal correlations.

Finally, in deep Q-learning, a secondary target network estimates the target update values. This target network is merely a frozen copy of the primary network. Although the target net’s parameters are frozen, the memory values are not; therefore, when a state is
passed through the target net, it will write to its copy of the memory. Over time, the target
network’s memory will diverge from the primary network’s. Is that a problem? What if the
target net just does not write to memory? Perhaps the two nets ought to share access to a
single memory, and the target net does not write? We have not found a satisfactory answer
to these questions.
Chapter 6

Models 3 and 4: Product Key Memory

In this chapter, we sacrifice some memory flexibility to address some of the concerns with the prior models, discussed at the end of the last chapter. Those concerns mainly dealt with computational efficiency, and difficulties arising from writing to memory during offline training.

For these models, we replace the DNC-style memory with a product key memory (PKM) [20]. This memory has no write function at all, but the values inside are learned through backpropagation like a regular neural network layer. What keeps it different from a regular network layer is that it is accessed sparsely through its read operation, and so only the most relevant information contributes to the prediction at each time step.

There are other interesting differences between the PKM and DNC memories. In the DNC, the write function updates the memory parameters instantly when faced with new data; the idea behind the design of that model is that the memory helps the network adapt quickly to events within an episode. In this scenario, it would be harmful to keep things in memory from one episode to the next, since they would no longer be relevant. Therefore, the memory is erased between episodes. This is true during both training and evaluation.

With the product key memory, the concept of memory is a little different. Because the values are learned through gradient propagation rather than written explicitly, the memory is not erased between episodes. This guides the memory to learn things that remain pertinent at all times across episodes. Moreover, the memory does not update at all during evaluation, only during training. In a sense training is like studying for a test, and evaluation is taking
In the test: studying should fill the memory with crucial information, and during the test all of that knowledge is used but no new information is gained. Thus while training, we fill and tune the contents of the memory, but during evaluation, memory is read-only.

The upshot of this is we lose some major flexibility from being able to write specific things as needed, but we also avoid difficulties associated with writing during randomly sampled training batches. The DNC-style memory focuses more on specific short-term knowledge, while this model learns abstract knowledge that will be helpful across a broader range of scenarios. With some notable exceptions, most Atari games do not require much short-term memory, since all necessary information is typically visible on the screen. As argued in section 2.2, these games may be purely reactive, yet can still benefit from the abstract large-scale memory that the PKM provides.

6.1 Model description

As with the previous models, we built on the framework of Rainbow DQN. Please see section 4.1 to review Rainbow as needed.

We will now describe the product key memory.

Recall that in the standard key-based read operation described in section 4.2.3, the query $q^r$ is compared to every key $k_i \in \mathcal{K}$, resulting in a total of $|\mathcal{K}| = d_M$ kernel operations, where $d_M$ is the number of elements in memory. The read output vector is then a weighted sum of all memory locations. This limits the size of the memory as it becomes inefficient to compute a large number of these operations. One strategy is to only include the top $k$ memory locations whose keys are most similar to the query—but how can these be determined except by passing all of the keys through the kernel? The product key memory introduces a clever strategy to solve this issue with far fewer kernel evaluations. This lets us greatly increase the memory size, and model capacity, without a great increase in cost.
Figure 6.1: An example of the Product Key Memory read process. (a) The memory matrix, showing how two sets of $\sqrt{d_M} = 3$ subkeys induce a memory with $d_M = 9$ rows. (b) A query is split in half, and each half is compared to one subkey set. The $k = 2$ best matches from each are selected. (c) The $k$ product keys nearest the query are now guaranteed to be in the set induced by the selected subkeys, which is of size $k^2 = 4$. The full query is now compared to this set, yielding the $k$ desired keys, from which can be obtained the desired values. Figure adapted from Lample et al. [20].

We define two ordered sets of subkeys,

$$C := \{c_i\}_{i=1}^{\sqrt{d_M}}, \quad \text{and}$$

$$C' := \{c'_i\}_{i=1}^{\sqrt{d_M}}.$$

The subkeys $c_i$ and $c'_i$ are initialized randomly and do not change. Each $c_i$ and $c'_i$ is an element of $\mathbb{R}^{d_k/2}$, with $d_k = d_q$ as before. These two sets induce the much larger set of product keys,

$$\mathcal{K} := C \times C' = \{[c, c']|c \in C, c' \in C'\}.$$

Note that $\mathcal{K}$ has $d_M$ elements, and that this requires $d_M$ to be a perfect square.

Remember, the goal is to efficiently query $\mathcal{K}$. The idea is to reduce the pool of candidate keys by first querying the smaller sets of subkeys $C$ and $C'$. 


45
First, the query \( q^r \) is written \( q^r = [q_1, q_2] \in \mathbb{R}^{d_4/2} \times \mathbb{R}^{d_4/2} \). The subqueries \( q_1 \) and \( q_2 \) are scored each against one set of subkeys as follows:

\[
\begin{align*}
  s_1 &:= \sigma(\kappa(C, q_1)) \in \mathbb{R}^{\sqrt{d_M}}, \\
  s_2 &:= \sigma(\kappa(C', q_2)) \in \mathbb{R}^{\sqrt{d_M}}.
\end{align*}
\]

Here, the kernel \( \kappa \) is the inner product, so \( \kappa(C, q_1) := [c^T q_1]_{c \in C} \).

Next, let \( T_k \) be the top-\( k \) operator on vectors \( x \in \mathbb{R}^n \), so that \( T_k(x) \) returns the set of indices corresponding to the \( k \) largest elements of \( x \). That is, \( T_k(x) := I, |I| = k \), such that \( \min(\{x_i\}_{i \in I}) \geq \max(\{x_i\}_{i \notin I}) \). We select the product keys from \( C \) and \( C' \) corresponding to the largest values of their respective score vectors:

\[
\begin{align*}
  C_k &:= \{c_i | i \in T_k(s_1)\}, \\
  C'_k &:= \{c'_i | i \in T_k(s_2)\}.
\end{align*}
\]

We are guaranteed that the \( k \) keys nearest to the original query \( q^r \) will be among the product set

\[
\hat{K} := C_k \times C'_k = \{(c, c') | c \in C_k, c' \in C'_k\} \subset K,
\]

which is of size \( k^2 \ll d_M \). Now the kernel can efficiently be applied to this set, giving the scores \( \hat{s} := \sigma(\kappa(\hat{K}, q^r)) \). With these scores, we select the final \( k \) keys to be used in the read operation, \( K := \{\hat{k}_i | i \in T_k(\hat{s})\} \). Finally, the read output vector can be computed

\[
r := V^T s,
\]

where \( V \subset \mathcal{V} \) is the set of memory values corresponding to the selected keys \( K \), and \( s \) comprises their associated scores from \( \hat{s} \). A depiction of this process is shown in Figure 6.1.
The complexity of a query over flat keys is $O(|\mathcal{K}| \times d_q)$, while the complexity of using product keys is only $O((\sqrt{|\mathcal{K}|} + k^2) \times d_q)$. For details, see Lample et al. [20]. Remember that $|\mathcal{K}| = d_M$, the number of elements in the memory. For a given computation budget, using product keys allows the memory to be much larger.

The choice of $k$ is somewhat arbitrary, but computational gains are biggest if $k^2$ is multiple orders of magnitude smaller than $d_M$. For our experiments, we set $k = 16$.

There is no theoretical reason that the PKM does not have a write function. As implemented by the authors, all parameters are learned through backpropagation. For our purposes, this helps to sidestep many of the difficulties encountered when training the DNC-style models. In future work it would be interesting to add a write function to the PKM models.

As with the DNC-style models, we test two model variants with product-key memory, one with and one without a residual connection across the memory layer. The details are the same as in section 5.1. These two models are likewise called PKM-noRes and PKM-res.

In addition to the hyper-parameters inherited from Rainbow and DQN described in tables 4.2 and 4.1, we have the following hyper-parameters specific to the PKM memory models.

<table>
<thead>
<tr>
<th>Hyper-parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>K nearest neighbors</td>
<td>16</td>
</tr>
<tr>
<td>Key dimension</td>
<td>64</td>
</tr>
<tr>
<td>Number of keys</td>
<td>256</td>
</tr>
<tr>
<td>Memory size</td>
<td>65,536</td>
</tr>
<tr>
<td>Cell size</td>
<td>512</td>
</tr>
<tr>
<td>Read heads</td>
<td>2</td>
</tr>
<tr>
<td>Learning rate</td>
<td>0.001</td>
</tr>
</tbody>
</table>

Table 6.1: Hyper-parameters for the Product Key Memory layer. As recommended by Lample et al. [20], this layer has a higher learning rate than the rest of the network. This helps counteract the sparse training signal each parameter receives due to being used in only a fraction of the updates.

We constructed this memory unit with code linked by Lample et al. [20] in their paper.
Chapter 7

Experiments and Analysis

We tested and benchmarked our four models on ten games from the Atari Learning Environment. Since Rainbow serves as our baseline, we attempted to choose some games on which Rainbow performed strongly, some very poorly, and some at a mediocre level, as compared to other models. This should help illuminate in what ways if any the auxiliary memory is helping.

We selected the following games where Rainbow performed well: Space Invaders, Alien, Frostbite, and Gravitar. On Gravitar, a uniquely complicated game, Rainbow did far better than its baselines, yet did not surpass the human performance baseline. We were curious whether the AMU would help the agent to better understand and keep track of the various planets and universes, camera perspectives and hierarchical series of goals.

For tasks where Rainbow performed poorly compared to its baselines, we include the games Bowling, Venture, and Seaquest. Rainbow did successfully learn to play Seaquest, but it was surpassed by other models. This suggests there is room to improve, hence it is of special interest as well.

We also include three games where Rainbow’s performance was near the average of the other models it was compared to. These are Breakout, Enduro, and Tutankham. This gives us a total of ten test games.

Hessel et al. [15] trained Rainbow for 200 million game frames. Due primarily to computational costs, we terminate training at 30 million frames, even though many agents were still improving at the end of that period. This does not compromise our results, since
we are more interested in learning rate than final performance, and 30 million is enough to observe the learning behaviors of the different agents. Therefore instead of reporting only the final performance of each agent on every game, we report the average performance of each model at six points during training, with no implication that the performance at 30 million frames is “final” in any sense. This data is included in Appendix A.

We trained each model ten times on each game. Every hundred thousand training steps, training was paused for evaluation. The agent’s performance was measured as the average score on ten evaluation episodes. Training hyper-parameters inherited from DQN are shown in table 7.1. Table 7.2 shows additional training parameters introduced by Rainbow.

<table>
<thead>
<tr>
<th>Hyper-parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discount factor</td>
<td>0.99</td>
</tr>
<tr>
<td>Replay buffer size</td>
<td>1M transitions</td>
</tr>
<tr>
<td>Replay period</td>
<td>every 4 agent steps</td>
</tr>
<tr>
<td>Minibatch size</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 7.1: Training hyper-parameters retained from DQN.

<table>
<thead>
<tr>
<th>Hyper-parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam Learning rate</td>
<td>0.0000625</td>
</tr>
<tr>
<td>Adam $\epsilon$</td>
<td>0.00015</td>
</tr>
<tr>
<td>Priority exponent $\omega$</td>
<td>0.5</td>
</tr>
<tr>
<td>Priority importance sampling $\beta$</td>
<td>$0.4 \rightarrow 1.0$</td>
</tr>
<tr>
<td>Distributional min/max values</td>
<td>[-10, 10]</td>
</tr>
<tr>
<td>Noisy nets $\sigma_0$</td>
<td>0.1</td>
</tr>
<tr>
<td>Multi-step returns $n$</td>
<td>3</td>
</tr>
<tr>
<td>Target net period</td>
<td>8K*</td>
</tr>
<tr>
<td>Frames before learn start</td>
<td>20K*</td>
</tr>
</tbody>
</table>

Table 7.2: Additional hyper-parameters retained from Rainbow. The * indicates this parameter differs from the original paper.

7.1 Plots

Figure 7.1 shows the median performance across all ten games, normalized to the human baselines reported by Mnih et al. [22], with 0% being the average score of a random agent.
Figures 7.2 through 7.11 depict performance on each individual game. In these plots, the left $y$-axis shows the score during training averaged over ten consecutive episodes (no parameter updates occur during these evaluation periods). The right $y$-axis shows this score as a percentage of the human baselines. These human baseline scales are for reference only; we did not train for as many frames as Rainbow, and since this is a benchmarking task, we care more about the agents’ performance relative to each other than to any external yardstick. Moreover, since we are concerned with data efficiency, we are primarily interested in the slope of the learning curves.

For each game, the right plot includes shaded regions indicating one standard deviation from the mean performance; the left plot shows only the mean for legibility.

In all plots, curves are smoothed with a moving average of ten.
Figure 7.1: Median performance of each model across all ten games, normalized to human performance.
Figure 7.2: Space Invaders
Figure 7.3: Alien
Figure 7.4: Frostbite
Figure 7.5: Gravitar
Figure 7.6: Breakout
Figure 7.7: Enduro
Figure 7.8: Tutankham
Figure 7.9: Bowling
Figure 7.10: Venture
Figure 7.11: Seaquest
7.2 Discussion

A quick look at these results reveals that our memory models have not improved over the baseline. The key insights of the following discussion are summarized here.

- The DNC-style models struggle because of conflicts between the intent of the replay buffer and the auxiliary memory. It is not clear how the AMU should behave, particularly the write function, when training on batches of randomly sampled historical transitions. In addition, the computational cost of this memory layer prohibits it from being very large.

- In seeking to avoid the difficulties plaguing the DNC models, our PKM agents gained an efficient large memory layer, but sacrificed the flexibility of a write function. This tradeoff improved performance, but still does not provide a consistent advantage over our baseline.

- The models with a residual connection outperformed their non-residual counterparts. The residual connection connects the early layers to a strong gradient signal so they can more quickly learn to produce useful observation encodings.

- Once these issues are resolved, further research will determine to what extent memory models can improve reinforcement learning.

In section 5.2 we mentioned several issues that make it difficult to train a DNC-style memory with the deep Q-learning algorithm used by Rainbow and other Q-network models. For one thing, the goal of the memory and the purpose of the replay buffer are at odds. One of the memory’s main purposes is to provide context to an observation by storing relevant recent information. A primary goal of the replay buffer, on the other hand, is to strip observations of all context by presenting them in random order. This was necessary to stabilize the algorithm, but it makes sense that the memory will not be as useful when training on randomly sampled observations. It could in fact be harmful as information is written with no regard to temporal
connection, and then queried for events that may or may not have anything to do with what was lately written.

When a human reviews their own past experience, such as evaluating how they performed in some given situation, it does make sense both to “read” earlier memories and to “write” or draw new conclusions. Perhaps this is because a human reviews not isolated state transitions but entire sequences at a time. A possible solution, then, is for the replay buffer to store *sequences* of transitions of a given length, rather than individual transitions, and then to sample randomly those among those sequences. This may effectively compromise between the opposing goals of the replay buffer and the AMU, by being sufficiently random to smooth the shifting data distribution, but provide enough context so that reading and querying the memory are meaningful.

We removed the temporal link matrix, one of the DNC’s most powerful features, for the similar reason that it does not make sense to remember the order memories were written when there is no temporal correlation between the inputs. The DNC was originally trained online only, meaning on one sample at a time as the agent experiences it, not training on batches of historical data. The temporal link matrix makes much more sense with online training. We hypothesize that the temporal link matrix would be a powerful tool for learning in the Atari environment, but it would require a new training algorithm, probably an online one, or else very careful thought and planning to make it work with offline data, perhaps involving sequences of transitions as mentioned before.

More thought should also be given to the function of the memory unit in the target network, employed by the deep Q-learning algorithm to stabilize updates. The target network is simply an old copy of the primary network, and so it has an old version of the memory. The main question is whether the target net ought to write to memory when it is producing an update target. Our intuition says it should not, so we turned it off for our experiments. But we did not research this deeply and there might be a better answer.
Again, it is not clear whether the network should write to memory when training on a batch of randomly sampled data. And if it does write, what is the correct way to write a batch of memory updates? Each update vector is queried against the existing memory to determine where it should be written, so the order they are written makes a huge difference. We solved this question by writing each new update to a separate copy of the memory, then averaging the memory copies back down into one. Other models, including [23], have simply designed a hard-coded write function, and so it is not necessary to write during a training step. Every approach has advantages and disadvantages, and more research may illuminate what is best for any given scenario.

Due to all these complications, it is not surprising to us that the DNC models performed the worst. The deep Q-learning algorithm does not work well with the idea of writing to a persistent memory. In fact, the DNC models actually performed worse than a random agent on Bowling and Gravitar. DNC with no residual connection also performed worse than random on Space Invaders and Tutankham. A random agent will continually bump into small amounts of reward in these games, but it is possible for a poor RL agent never to get out of the rut of doing the wrong thing.

The PKM models sidestepped all of these memory issues by not having a write function at all, and learning the memory values through backpropagation. This has its own issues. The two huge advantages we intended to gain through external memory were rapid storage of information and precise retrieval. By not writing at all we completely wipe out the first advantage. We can still accurately query specific information independent of the rest of the layer, but the memory is not stable, that is, we cannot count on finding the same information in the same place from timestep to timestep (at least during training).

The astute reader may point out that our memory models, in particular the PKM models, have vastly more parameters than the baseline Rainbow. We attempted to train a version of Rainbow with as many parameters as our PKM model, to ensure there was a fair comparison. We were unable to complete the experiment because the model was far too
computationally expensive to train. This drives home the essential advantage of product-key memory: efficient access to a huge layer.

A similar issue is that the DNC-style memory units do not scale up well. Our DNC models had a much smaller memory than our PKM models due to computational constraints, and this could be a source of bias in our results. Had we been aware of it early enough, we would have used the Sparse DNC of Rae et al. [27] for this research.

With this mix of advantages and drawbacks, our best PKM model performs on par with the baseline, Rainbow. The DNC-based models performed far worse. With both models, the version with the residual connection performed better than the version without. We believe the primary reason for this is the same reason that residual connections were shown to benefit deep models in the first place, namely improved flow of gradients back to the earlier layers of the network [14]. Our models are not so deep in number of layers, but the memory layer is large, and in the case of PKM, only a small portion of contributes to each prediction. This means that the gradient reaching the convolutional layers has high variance, with the result that those layers take longer to learn to produce useful encodings, which in turn impacts the learning of the read and write functions which depend on those encodings. The residual connection reduces that variance, which is especially crucial early in training when the convolutions are still learning how to send a non-random signal on to the memory.

An alternative explanation to the improved performance of the residual models, is that the memory units, as we’ve implemented them, only harm performance and the residual connection lets the network effectively ignore the AMU completely. We did not investigate this possibility.

7.3 Qualitative observations

What are these agents actually learning? We observed several agents evaluate on each game, and were intrigued by what we saw. In many games, including Enduro, Alien, Frostbite, Seaquest, and Space Invaders, the agents did perform strongly. However, in several of the
harder games, the agents did not appear to be performing well, despite achieving fairly high scores relative to the human baselines reported by Mnih et al. [22].

For instance, in Gravitar our best agents achieved 70-80% human performance, yet observation revealed that they had not learned anything resembling human behavior. The agent careens wildly about the screen and crashes into the bunkers, a behavior which does earn points but costs a life. Evidently this game is difficult enough that the professional game tester in Mnih et al. [22] fared only a little better trying to play the game properly.

In Venture, there are treasures in four rooms connected by hallways. Monsters patrol the hallways trying to kill the player. Many of our agents learned to run to the edge of the screen and wait for a monster to come and kill them. The best agents (reaching 100% human performance) also learned to enter some of the rooms to gather treasures, but still exhibited this unusual hiding behavior. An RL agent cannot begin learning until it receives some reward; it is extremely unlikely for a fresh randomly initialized agent to dodge multiple monsters, find the doorway to a room, and stumble into a treasure under an $\epsilon$-greedy exploration policy. We were not surprised this game was hard to learn, but we were surprised that this behavior is described as 90–100% human performance.

A similar thing happened in Tutankham. In this game, the player is supposed to navigate a maze, collecting treasures and shooting enemies along the way. Our best agent purportedly achieved 150% human performance, but upon observation we discovered that the agent simply remained in one place shooting enemies as they spawned, until ammunition ran out. Then the agent would venture to collect two or three treasures before finding a place where enemies either will not or cannot come, to wait until the episode times out. Even if it is possible to collect a fair number of points this way, we would hesitate to describe that as super-human performance. We observed a high-scoring Rainbow agent, as well as one of each memory model, and all three exhibited similar behavior.

We see strange behavior even in some games where agents perform well. In Breakout, agents that surpass 1000% human performance fall into patterns where they cease to progress.
Near the end of the first level, when there are only a few bricks left, an agent will either choose never to launch the last ball, or else bounce the ball repeatedly off the wall but never clear another brick. This is odd behavior for an agent that has already demonstrated the skills needed to finish the level. We hypothesize that the agent has learned that the fewer bricks remain, the harder it becomes to get reward. Perhaps at some point, clearing an additional brick brings no expectation of future reward. A better exploration policy could mediate this as well. Without clearing the last bricks, an agent can never realize that doing so would bring a new screen full of bricks and opportunity for more reward.

It was somewhat disappointing to observe these idiotic strategies, learned by agents with scores nearing or surpassing those of a professional human tester. We do not question the integrity of that tester or their reported scores in Mnih et al. [22], but we do question whether we are measuring the right thing. It seems disingenuous to call an agent superhuman which learns to never launch the last ball, and to declare success when dismal behavior is concealed behind relatively high point scores. Toromanoff et al. [33] discuss this issue further and suggest a new baseline metric based on human world records, against which the median score of their Rainbow variant is only 3.1%.
Chapter 8

Conclusions and Future Work

In this thesis we’ve created four auxiliary memory models and benchmarked their performance across ten tasks. Due to limitations with each model, none of them consistently improved performance over the baseline. Our experiments are insufficient to conclude that auxiliary memory is not a good idea; there is far more research suggesting that it is. Rather, we conclude that auxiliary memory has the potential to benefit reinforcement learning, but it will require more work to design a memory agent that will avoid the pitfalls we encountered.

Anyone seeking to create a better reinforcement learning memory agent should consider the following points. First, the memory unit needs to have the full expression of a write function. The PKM models we tested, which learn memory values by gradient propagation, cannot adjust their memory fast enough to reap the full benefit of auxiliary memory. However, the product-key method of memory indexing is fast and allows the memory layer to be huge. On the other hand, the DNC models, despite having a write function, suffered from complications arising from the interaction of the memory and the offline training algorithm. A clear next step, then, is to carefully design a model and training algorithm that will work together in harmony, enabling the full power of a memory unit with write and read functions.

To do this, it will probably be necessary to disconnect from Rainbow. Our models attempted to add a minimal memory layer to Rainbow while changing as little else as possible, and we observe no benefit. A better memory agent will be constructed from scratch, with all features, architectures, and training algorithms specifically designed to work well with a memory unit. In particular, the training algorithm will either need to be purely online, or
else designed carefully to preserve the intention and meaning of memory when training on temporally disconnected samples.

There are other interesting questions to consider as well. An obvious one is “What is the memory doing anyway?” As memory networks become more widespread, it will be important to have tools for examining and understanding the contents of an AMU. High-dimensional visualization methods like t-SNE [35] or UMAP [21] will be helpful but insufficient.

It would also be interesting to build a model with multiple stacked memory layers. How would two small stacked memory units compare to one large one? What about a hierarchy with several small units feeding into one larger one? Would each memory layer remember different sorts of things? How deep could we go?

It seems reasonable to believe that a successful memory agent would have an advantage in transfer learning. Having not only learned to perform a task, but having learned how to learn to perform tasks, an agent could conceivably adapt to a new task or environment with minimal or no weight updates. The network already knows how to use the read and write functions, and all task-specific information can be stored quickly in the memory.

Augmented memory networks remain a relatively unexplored subfield of deep learning. We are excited to see what future research will discover.
References


[8] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alex Graves, Vlad Mnih, Rémi Munos, Demis Hassabis, Olivier Pietquin, Charles


[17] Sepp Hochreiter, A. Steven Younger, and Peter R. Conwell. Learning to learn using
gradient descent. In IN LECTURE NOTES ON COMP. SCI. 2130, PROC. INTL.

2020-09-23.

[19] Arbaaz Khan, Clark Zhang, Nikolay Atanasov, Konstantinos Karydis, Vijay Kumar,

[20] Guillaume Lample, Alexandre Sablayrolles, Marc’Aurelio Ranzato, Ludovic Denoyer,

[21] Leland McInnes, John Healy, and James Melville. Umap: Uniform manifold approxima-

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness,
Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg
Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King,
Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level
ISSN 00280836. URL http://dx.doi.org/10.1038/nature14236.

[23] Junhyuk Oh, Valliappa Chockalingam, Satinder P. Singh, and Honglak Lee. Control of
memory, active perception, and action in minecraft. CoRR, abs/1605.09128, 2016. URL

2018-11-22.

1702.08360.

[26] Alexander Pritzel, Benigno Uria, Sriram Srinivasan, Adrià Puigdomènech Badia, Oriol
Vinyals, Demis Hassabis, Daan Wierstra, and Charles Blundell. Neural episodic control.


Appendix A

Performance Data

Table A.1 shows the performance of each model on each game at six different moments during training. We show the mean and standard deviation over ten training runs. For every game, the mean is highlighted at each time step for the model that is currently performing best. Note that this table contains raw data, while the plots in Figures 7.2–7.11 average the data over ten consecutive time steps. This may result in apparent discrepancies in some cases.
<table>
<thead>
<tr>
<th>Game</th>
<th>Model</th>
<th>0.5e7</th>
<th>1e7</th>
<th>1.5e7</th>
<th>2e7</th>
<th>2.5e7</th>
<th>3e7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rainbow</td>
<td>1065.0 (192.5)</td>
<td>1807.5 (350.4)</td>
<td>2313.8 (235.2)</td>
<td>3473.5 (1959.6)</td>
<td><strong>6757.4</strong> (4627.4)</td>
<td><strong>6850.9</strong> (4440.8)</td>
<td></td>
</tr>
<tr>
<td>PKM-res</td>
<td><strong>1172.3</strong> (278.9)</td>
<td>2398.9 (310.0)</td>
<td><strong>3002.2</strong> (627.2)</td>
<td><strong>4594.9</strong> (2943.0)</td>
<td>4857.9 (2029.5)</td>
<td>6447.1 (3549.0)</td>
<td></td>
</tr>
<tr>
<td>Space Invaders</td>
<td>PKM-noRes</td>
<td>768.0 (175.4)</td>
<td>1419.8 (358.4)</td>
<td>1530.7 (372.4)</td>
<td>1709.6 (264.1)</td>
<td>2050.9 (482.3)</td>
<td>2005.7 (266.8)</td>
</tr>
<tr>
<td>DNC-res</td>
<td>457.9 (124.5)</td>
<td>447.6 (194.4)</td>
<td>465.8 (152.0)</td>
<td>429.9 (133.0)</td>
<td>367.9 (97.9)</td>
<td>432.4 (240.1)</td>
<td></td>
</tr>
<tr>
<td>DNC-noRes</td>
<td>238.5 (85.4)</td>
<td>118.9 (67.5)</td>
<td>65.8 (77.5)</td>
<td>94.2 (106.8)</td>
<td>70.7 (84.9)</td>
<td>95.8 (115.3)</td>
<td></td>
</tr>
<tr>
<td>Rainbow</td>
<td>3771.1 (2475.0)</td>
<td><strong>7257.4</strong> (192.5)</td>
<td>7986.4 (3379.0)</td>
<td>10027.1 (3040.5)</td>
<td>11791.5 (6044.6)</td>
<td>12513.8 (5874.0)</td>
<td></td>
</tr>
<tr>
<td>PKM-res</td>
<td><strong>3796.2</strong> (1933.9)</td>
<td>6582.2 (3957.1)</td>
<td><strong>9405.9</strong> (8355.3)</td>
<td><strong>15130.0</strong> (15875.4)</td>
<td><strong>21369.2</strong> (32637.9)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seaquest</td>
<td>PKM-noRes</td>
<td>1404.4 (598.3)</td>
<td>2576.7 (1008.0)</td>
<td>3601.0 (1067.7)</td>
<td>4829.2 (1295.2)</td>
<td>5890.6 (933.5)</td>
<td>6632.8 (1802.3)</td>
</tr>
<tr>
<td>DNC-res</td>
<td>813.2 (329.6)</td>
<td>933.7 (565.0)</td>
<td>1120.0 (591.1)</td>
<td>1491.7 (1272.4)</td>
<td>880.1 (490.1)</td>
<td>961.5 (1142.7)</td>
<td></td>
</tr>
<tr>
<td>DNC-noRes</td>
<td>195.4 (121.4)</td>
<td>257.0 (162.7)</td>
<td>460.8 (242.2)</td>
<td>248.0 (212.5)</td>
<td>293.4 (169.8)</td>
<td>359.8 (200.2)</td>
<td></td>
</tr>
<tr>
<td>Rainbow</td>
<td>2049.1 (618.0)</td>
<td>2630.2 (826.9)</td>
<td>3920.1 (950.4)</td>
<td>4525.0 (1267.3)</td>
<td>4050.7 (991.0)</td>
<td>5119.4 (1622.1)</td>
<td>4722.6 (783.8)</td>
</tr>
<tr>
<td>PKM-res</td>
<td><strong>2076.8</strong> (409.1)</td>
<td><strong>3032.9</strong> (907.6)</td>
<td><strong>4525.0</strong> (8355.3)</td>
<td><strong>15130.0</strong> (15875.4)</td>
<td><strong>21369.2</strong> (32637.9)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alien</td>
<td>PKM-noRes</td>
<td>1735.5 (479.5)</td>
<td>2134.3 (755.7)</td>
<td>2529.2 (612.7)</td>
<td>3034.9 (667.2)</td>
<td>2853.9 (713.9)</td>
<td>3310.7 (669.4)</td>
</tr>
<tr>
<td>DNC-res</td>
<td>1215.6 (148.6)</td>
<td>1254.3 (408.9)</td>
<td>1313.7 (247.9)</td>
<td>1068.6 (340.7)</td>
<td>840.5 (264.8)</td>
<td>837.3 (198.3)</td>
<td></td>
</tr>
<tr>
<td>DNC-noRes</td>
<td>422.7 (196.7)</td>
<td>361.6 (167.9)</td>
<td>226.5 (119.7)</td>
<td>258.1 (187.4)</td>
<td>316.3 (132.2)</td>
<td>369.9 (260.0)</td>
<td></td>
</tr>
<tr>
<td>Rainbow</td>
<td>2154.1 (81.5)</td>
<td><strong>2200.9</strong> (51.7)</td>
<td><strong>2214.9</strong> (67.9)</td>
<td><strong>2247.5</strong> (42.2)</td>
<td>2269.8 (28.6)</td>
<td><strong>2243.1</strong> (44.4)</td>
<td></td>
</tr>
<tr>
<td>PKM-res</td>
<td>1927.2 (150.5)</td>
<td>2175.2 (120.0)</td>
<td>2175.2 (333.6)</td>
<td>2225.3 (69.6)</td>
<td><strong>2821.5</strong> (61.1)</td>
<td>2232.3 (95.6)</td>
<td></td>
</tr>
<tr>
<td>Enduro</td>
<td>PKM-noRes</td>
<td>1436.1 (218.0)</td>
<td>1683.7 (303.2)</td>
<td>1836.3 (183.9)</td>
<td>1778.6 (306.9)</td>
<td>1921.6 (198.6)</td>
<td>1991.0 (263.6)</td>
</tr>
<tr>
<td>DNC-res</td>
<td>1296.1 (233.9)</td>
<td>1109.1 (539.2)</td>
<td>840.9 (528.1)</td>
<td>687.7 (403.9)</td>
<td>458.9 (282.9)</td>
<td>393.0 (267.0)</td>
<td></td>
</tr>
<tr>
<td>DNC-noRes</td>
<td>7.0 (6.3)</td>
<td>19.0 (26.1)</td>
<td>41.2 (54.9)</td>
<td>61.9 (9.8)</td>
<td>8.2 (12.8)</td>
<td>6.8 (11.8)</td>
<td></td>
</tr>
<tr>
<td>Rainbow</td>
<td>202.7 (43.0)</td>
<td><strong>329.6</strong> (34.8)</td>
<td><strong>358.8</strong> (31.8)</td>
<td><strong>344.0</strong> (33.6)</td>
<td><strong>335.9</strong> (58.6)</td>
<td><strong>351.2</strong> (34.7)</td>
<td></td>
</tr>
<tr>
<td>PKM-res</td>
<td>167.4 (77.3)</td>
<td>276.8 (101.8)</td>
<td>315.1 (69.7)</td>
<td>276.3 (73.2)</td>
<td>317.4 (37.7)</td>
<td>289.5 (45.6)</td>
<td></td>
</tr>
<tr>
<td>Breakout</td>
<td>PKM-noRes</td>
<td>97.1 (62.1)</td>
<td>37.6 (30.9)</td>
<td>26.6 (30.6)</td>
<td>24.8 (29.2)</td>
<td>26.0 (37.7)</td>
<td>20.1 (23.9)</td>
</tr>
<tr>
<td>DNC-res</td>
<td>2.8 (2.4)</td>
<td>2.4 (1.4)</td>
<td>3.5 (6.2)</td>
<td>2.0 (3.0)</td>
<td>1.6 (2.2)</td>
<td>1.5 (1.4)</td>
<td></td>
</tr>
<tr>
<td>DNC-noRes</td>
<td>0.9 (0.7)</td>
<td>0.7 (0.4)</td>
<td>1.9 (1.4)</td>
<td>1.5 (1.1)</td>
<td>1.4 (1.2)</td>
<td>1.6 (0.9)</td>
<td></td>
</tr>
<tr>
<td>Game</td>
<td>Model</td>
<td>Training frames</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>-----------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.5e7</td>
<td>1e7</td>
<td>1.5e7</td>
<td>2e7</td>
<td>2.5e7</td>
<td>3e7</td>
</tr>
<tr>
<td>Rainbow</td>
<td>PKM-res</td>
<td>66.9</td>
<td>(7.5)</td>
<td>46.4</td>
<td>(14.9)</td>
<td>45.6</td>
<td>(16.8)</td>
</tr>
<tr>
<td>Bowling</td>
<td>PKM-noRes</td>
<td>30.0</td>
<td>(0.0)</td>
<td>30.0</td>
<td>(0.0)</td>
<td>27.0</td>
<td>(9.0)</td>
</tr>
<tr>
<td></td>
<td>DNC-res</td>
<td>14.0</td>
<td>(11.7)</td>
<td>19.2</td>
<td>(15.7)</td>
<td>23.2</td>
<td>(21.9)</td>
</tr>
<tr>
<td></td>
<td>DNC-noRes</td>
<td>22.0</td>
<td>(25.2)</td>
<td>8.8</td>
<td>(12.1)</td>
<td>7.5</td>
<td>(9.1)</td>
</tr>
<tr>
<td>Rainbow</td>
<td>PKM-noRes</td>
<td>153.8</td>
<td>(18.7)</td>
<td>161.3</td>
<td>(19.9)</td>
<td>160.8</td>
<td>(18.9)</td>
</tr>
<tr>
<td>Tutorham</td>
<td>PKM-noRes</td>
<td>125.5</td>
<td>(21.6)</td>
<td>115.2</td>
<td>(14.3)</td>
<td>102.5</td>
<td>(18.1)</td>
</tr>
<tr>
<td></td>
<td>DNC-res</td>
<td>46.7</td>
<td>(56.2)</td>
<td>60.3</td>
<td>(59.9)</td>
<td>74.5</td>
<td>(51.7)</td>
</tr>
<tr>
<td></td>
<td>DNC-noRes</td>
<td>1.0</td>
<td>(1.7)</td>
<td>4.9</td>
<td>(8.7)</td>
<td>2.6</td>
<td>(7.2)</td>
</tr>
<tr>
<td>Venture</td>
<td>PKM-noRes</td>
<td>684.0</td>
<td>(493.6)</td>
<td>642.0</td>
<td>(466.6)</td>
<td>409.0</td>
<td>(276.4)</td>
</tr>
<tr>
<td></td>
<td>PKM-res</td>
<td>142.8</td>
<td>(16.4)</td>
<td>154.6</td>
<td>(14.8)</td>
<td>154.7</td>
<td>(29.8)</td>
</tr>
<tr>
<td></td>
<td>DNC-res</td>
<td>24.0</td>
<td>(30.7)</td>
<td>629.0</td>
<td>(524.0)</td>
<td>592.0</td>
<td>(495.0)</td>
</tr>
<tr>
<td></td>
<td>DNC-noRes</td>
<td>0.0</td>
<td>(0.0)</td>
<td>0.0</td>
<td>(0.0)</td>
<td>0.0</td>
<td>(0.0)</td>
</tr>
<tr>
<td>Frostbite</td>
<td>PKM-NoRes</td>
<td>7575.0</td>
<td>(1161.5)</td>
<td>9998.5</td>
<td>(593.3)</td>
<td>10133.8</td>
<td>(518.5)</td>
</tr>
<tr>
<td></td>
<td>PKM-res</td>
<td>7703.6</td>
<td>(1684.5)</td>
<td>9434.6</td>
<td>(1171.8)</td>
<td>10794.0</td>
<td>(1745.9)</td>
</tr>
<tr>
<td></td>
<td>DNC-res</td>
<td>5974.3</td>
<td>(1262.7)</td>
<td>8506.2</td>
<td>(1230.5)</td>
<td>8185.2</td>
<td>(2043.6)</td>
</tr>
<tr>
<td></td>
<td>DNC-noRes</td>
<td>24.0</td>
<td>(30.7)</td>
<td>629.0</td>
<td>(524.0)</td>
<td>592.0</td>
<td>(495.0)</td>
</tr>
<tr>
<td></td>
<td>PKM-noRes</td>
<td>0.0</td>
<td>(0.0)</td>
<td>0.0</td>
<td>(0.0)</td>
<td>0.0</td>
<td>(0.0)</td>
</tr>
<tr>
<td>Gravitar</td>
<td>PKM-NoRes</td>
<td>356.4</td>
<td>(266.8)</td>
<td>400.7</td>
<td>(207.8)</td>
<td>482.1</td>
<td>(327.8)</td>
</tr>
<tr>
<td></td>
<td>PKM-res</td>
<td>649.5</td>
<td>(195.2)</td>
<td>883.0</td>
<td>(139.4)</td>
<td>1073.0</td>
<td>(241.3)</td>
</tr>
<tr>
<td></td>
<td>DNC-res</td>
<td>551.0</td>
<td>(85.3)</td>
<td>678.5</td>
<td>(148.5)</td>
<td>851.5</td>
<td>(125.0)</td>
</tr>
<tr>
<td></td>
<td>DNC-noRes</td>
<td>29.0</td>
<td>(66.6)</td>
<td>207.0</td>
<td>(79.3)</td>
<td>201.0</td>
<td>(97.9)</td>
</tr>
<tr>
<td></td>
<td>PKM-noRes</td>
<td>39.5</td>
<td>(65.1)</td>
<td>42.0</td>
<td>(56.7)</td>
<td>3.0</td>
<td>(6.4)</td>
</tr>
</tbody>
</table>

Table A.1: Periodic evaluation scores. Standard deviation in parentheses.