# Probabilistic Model-Based Reinforcement Learning Using The Differentiable Neural Computer

*Adeel I. Mufti*


Master of Science

Artificial Intelligence

School of Informatics

University of Edinburgh

2018

# Abstract

This thesis investigates the use of the Differentiable Neural Computer (DNC) in model-based Reinforcement Learning (RL). The approach involves learning a predictive, probabilistic model in a Mixture Density Network (MDN) using a Recurrent Neural Network (RNN) – MDN-RNN – which is then used to train a controller to perform optimal actions to receive maximum rewards in an environment.

The MDN-RNN is trained to map a state $s_t$ and action $a_t$ at timestep $t$, to the parameters of a Gaussian Mixture Model which are used to sample the predicted next state $s_{t+1}$ at timestep $t+1$. Recent research has used Long Short Term Memory (LSTM) as the RNN layer for the MDN – MDN-LSTM – and achieved promising results on RL tasks. This research introduces a new MDN architecture that uses a DNC as the RNN layer, the *MDN-DNC*.

The experiments conducted for this work find that the MDN-DNC, trained for fewer epochs, outperforms its LSTM counterpart in the ViZDoom: Take Cover environment in the mean cumulative rewards received with a controller trained using simulations of the model generated through the MDN. The MDN-DNC is observed to learn a more robust model of the environment that is less prone to defects that hinder training of the controller. The MDN-DNC also outperforms when learning the Pommerman Team Competition environment, where quantitative tests are made to compare the states it predicts against the true states received from the environment.

# Acknowledgements

Many thanks to my supervisor Subramanian Ramamoorthy, for allowing me the opportunity to conduct this research under his guidance. To fellow student Svetlin Penkov, for the hours spent talking through ideas and troubleshooting MDNs. And to my wife Dorothee Bond, for her endless patience through the research process.

# Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Adeel I. Mufti*)

# Table of Contents

# Chapter 1

# Introduction

*"We can only see a short distance ahead, but we can see plenty there that needs to be done."*

–Alan Turing, *Computing machinery and intelligence*

[1]Artificial General Intelligence (AGI) can be defined as the intelligence of a machine that can perform any task that a human can. Creating AGI has been a goal for research scientists for decades, but has been a difficult feat to achieve. The majority of advancements in Artificial Intelligence (AI) have been in "Narrow AI" – intelligence applied to specialized tasks such as classification, regression, transcription, translation, synthesis, and more (Goodfellow et al., 2016), and in applications such as medical diagnosis, stock trading, advertising, and gaming, to name a few. However, these AI models are often narrowly tailored, and do not generalize (Goertzel and Pennachin, 2007).

Deep Learning is a general purpose framework for representation learning – taking inputs to learn a representation that is required to achieve an objective (Silver, 2016). Reinforcement Learning (RL) is a general purpose framework for learning what to do – selecting actions (making decisions), given a state, to maximize future rewards (Sutton and Barto, 1998). David Silver states that *Deep Reinforcement Learning* (deep RL), the combination of Deep Learning as a mechanism and Reinforcement Learning as an objective, is a step in the ladder towards AGI (Silver, 2016). Evolution Strategies (ES) are a related class of algorithms where given a state, actions are selected to maximize a reward. In both ES and RL, the goal is to learn an optimal policy.

[1]Parts of this introduction are based on the Informatics Project Proposal for the research carried out.

## 1.1 Motivation

There have been many advancements in deep RL in recent years. A notable deep RL approach uses Deep Q-Networks (DQNs), which are able to surpass human performance on a range of Atari 2600 games (Mnih et al., 2015). While the DQN is a model-free deep RL approach, Guided Policy Search is a model-based approach, which uses a model of the system dynamics to guide the search for an optimal policy (Levine and Koltun, 2013). Another model-based approach, World Models, learns a predictive, probabilistic model of the environment in Long Short Term Memory (LSTM), which is used to search for an optimal policy (Ha and Schmidhuber, 2018).

Recurrent Neural Networks (RNN–LSTM being an example) are adept at modeling data and tasks that require state that spans over many timesteps in an input sequence. A particular new enhancement to RNNs is presented in a framework termed the Differentiable Neural Computer (DNC). The DNC consists of a LSTM *controller* that is attached to an external memory matrix. This type of architecture has achieved state-of-the-art results over traditional RNNs in various tasks (Graves et al., 2016).

As LSTM has shown to be able to learn a model of an environment that can be used in deep RL, it follows that the DNC could be used in place of the LSTM to perhaps learn a better model. And that is the primary motivation of this work. At the time this research was planned and carried out, a search across Google, Google Scholar, and research publications archives, did not yield any existing work on using a DNC to learn a predictive model of an environment in the RL context.

## 1.2 Hypothesis and Summary

The primary objective of this thesis was to learn a predictive, probabilistic model of an environment in a Differentiable Neural Computer (DNC), and use this model for Reinforcement Learning (RL) when searching for an optimal policy for a task in the environment. The hypothesis was that the DNC, in contrast to traditional LSTM, would provide a better learned predictive model, in environments with numerous complex states and dependencies spanning over long timesteps. It was suspected that such a robust model would be more useful when searching for a policy.

To test this hypothesis, model-based deep RL was performed with an environment model learned in a Mixture Density Network (MDN) based on a Recurrent Neural Network (RNN) – a MDN-RNN. LSTM was used as the RNN layer (MDN-LSTM)

to establish baselines, and then compared against a new architecture that used a DNC as the recurrent layer in the MDN – a *MDN-DNC*. Both variants of the MDN were used to learn the ViZDoom: Take Cover environment, which was then used to train a simple controller using Evolution Strategies, in an environment simulated by the MDN, to learn an optimal policy that maximized rewards. Additionally, both variants of the MDN were used to learn the Pommerman Team Competition environment, on which quantitative tests were made to compare the states predicted against the true states received from the environment. In either case, the MDN-DNC outperformed the MDN-LSTM in the experiments conducted.

## 1.3 Thesis Outline

Some important background information is provided in Chapter 2. Chapter 3 details the work undertaken, describing the conceptual design and actual implementation, along with problems and difficulties encountered through the process. Chapter 4 gives details of the particular experiments that were conducted during the research. The results of the experiments and an analysis are provided in Chapter 5. And finally, conclusions drawn and further work are covered in Chapter 6.

# Chapter 2

# Background

This chapter is aimed at providing the reader with background information on the core concepts and frameworks central to this thesis.

## 2.1 Model-Based Learning

According to Sutton and Barto (2017), in the Reinforcement Learning (RL) context, a model of an environment is "anything" that an AI agent can use to predict how an environment will respond to its actions. That is, "Given a state and an action, a model produces a prediction of the resultant next state and next reward". The model can be used to simulate an environment to produce simulated experience in which the agent learns. Or the model can be consulted in the live environment to get probabilistic predictions of what the next state could be, to facilitate learning. This is in contrast to model-free RL methods where no such predictive model is used, and learning is performed using only the live environment itself.

### 2.1.1 Guided Policy Search

Guided Policy Search (GPS) is a model-based RL method proposed by Levine and Koltun (2013). The GPS approach uses differential dynamic programming (DDP) to generate "guiding samples", that assist policy search by examining regions with high-reward. The guided samples are drawn from a distribution built around a DDP solution. Although the policy search component is model-free, DDP requires a model of the system dynamics. The DDP can be initialized from example human demonstrations, or an offline planning algorithm. Regularized importance sampled policy optimization

incorporates the guided samples into the policy search. Levine and Koltun (2013) evaluate GPS by learning neural network controllers for swimming, hopping, walking, and simulated 3D humanoid running. The authors show that GPS outperforms other RL methods on these tasks.

### 2.1.2 World Models

World Models is another notable framework for model-based RL, proposed by Ha and Schmidhuber (2018). The framework aims to train an agent that can perform well in virtual gaming environments. World Models consists of three main components: Vision (V), Model (M), and Controller (C) that interact together to form an agent.

The *Vision* consists of a Variational Autoencoder (VAE), which compresses state frames taken from the gameplay into a latent vector $z$. The *Model* consists of a Mixture Density Network (MDN), which involves outputting a mixture density model from a Recurrent Neural Network (RNN). This MDN-RNN takes latent vectors $z$ from Vision and predicts the next frame. And finally the *Controller* is a simple single layer linear neural network that maps the output from Model to actions to perform in the environment. The Controller is trained using Evolution Strategies, particularly the CMA-ES algorithm.

## 2.2 MDN-RNN

A Mixture Density Network (MDN), developed by Bishop (1994), combines a mixture density model with a neural network. The goal of the neural network is to output the parameters of a mixture density model which can be used to sample from to get an output, rather than have the neural network produce the output directly. The MDN can be trained to output the parameters of a Gaussian Mixture Model (GMM), which include a set of probabilities (coefficients for each mixture), set of means, and set of standard deviations.

The MDN approach is particularly useful when the desired output cannot simply be the average of likely correct outputs. In such data, mean square error (MSE) will not work. Take for example a toy problem where a single input $x$ could yield multiple outputs $y$ according to some probabilities. For example, at $x = 0.25$, $y$ could equal $\{0, 0.5, 1\}$, with each output occurring with probability of $1/3$. In such a case, using Mean Squared Error to train the network to produce a single output, would likely

always result in a value of 0.5.

According to Sutton and Barto (2017), "If the model is stochastic, then there are several possible next states and next rewards, each with some probability of occurring". Hence, using a MDN to learn a model would be necessary if the environment is stochastic rather than deterministic – given a current state there may be several possible next states.

The *RNN* in MDN-*RNN* simply means that the neural network that outputs the MDN parameters is a Recurrent Neural Network. Using an RNN is a certain design choice, but is not necessary–any type of neural network could be used (for example a simple feedforward network). The purpose of using an RNN is that it maintains an internal state over a sequence of inputs, and can hence make more useful predictions in an environment given future outputs that may depend on distant past inputs.

## 2.3 Evolution Strategies

According to research scientists at OpenAI, Evolution Strategies (ES) are a scalable alternative to Reinforcement Learning (Salimans et al., 2017). Where RL is a guess and check on the actions, ES are a guess and check on the model parameters themselves. In ES, a *population* of *mutations* to seed parameters is created, and all mutated parameters are checked for fitness (in this case, the maximum cumulative reward received). The parameters are then adjusted towards the mean of the fittest mutations. Each iteration of evolution is called a *generation*.

Note that though Evolution Strategies can be considered to be an *alternative* to RL, it can be conjectured that they are perhaps a form of RL. In both, an AI agent is trained to maximize the rewards it receives on a certain task in an environment, with some form of trial and error. Thus, in this thesis ES are not distinguished from traditional RL.

### 2.3.1 CMA-ES

Covariance Matrix Adaptation Evolution Strategies (CMA-ES) are a particular type of ES where a covariance matrix, $C$, is adapted as generations of evolution progress (Hansen, 2016). Based on the fitness (cumulative rewards) from the previous generations, compared to the cumulative rewards of the current generation, CMA-ES dynamically adjusts $C$ which is used when sampling from $\mathcal{N}(m,C)$, a multivariate normal

distribution with mean $m$ (the current parameters) and symmetric positive-definite covariance matrix $C$. This allows CMA-ES to cast a wider net for the next population to try if the fitness is decreasing, or narrow down towards the fittest parameters as fitness increases, thus dynamically accelerating learning.

## 2.4 Differentiable Neural Computer

The Differentiable Neural Computer (DNC) is a form of a memory augmented Neural Network that has shown promise on solving complex tasks that are difficult for traditional neural networks. In the DNC architecture (Figure 2.1), a core neural network (called the *controller* – not to be confused with controllers used in Reinforcement Learning) is attached to an external memory matrix, and is given the capacity to learn to make use of this memory. When an RNN is used as the controller in the DNC architecture, it has the capacity to solve and generalize well on tasks with long temporal dependencies across a sequence (Graves et al., 2016).

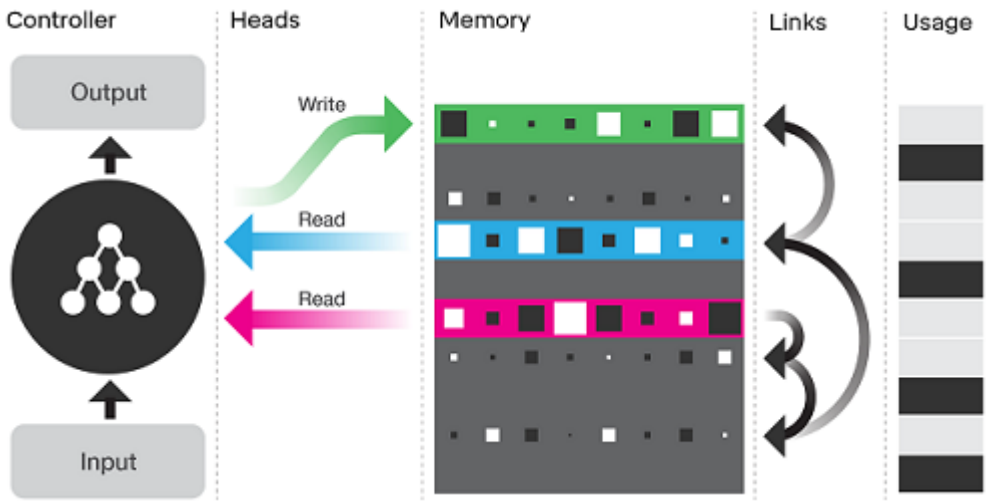



Figure 2.1: The Differentiable Neural Computer architecture (Graves et al., 2016).

At each timestep in a sequence, the DNC receives an external input, and memory data read from its internal read head mechanism from the previous timestep (with timestep 0 being a vector of 0's), to manipulate its internal state and produce the next output. The memory in the DNC is said to be external because the DNC can be trained on a smaller memory size, and then attached to a larger memory matrix.

### 2.4.1 DNC versus LSTM

The DNC is marketed as a neural *Turing machine*. A Turing machine is a hypothetical machine that can simulate any computer algorithm given the right capacity and instructions (Turing, 1937). In the classical example, a Turing machine consists of a long tape, and a read and write *head* for the tape which can manipulate symbols on the tape according to certain instructions (such as a table of rules). The DNC is constructed similarly, where a neural network controller capable of learning to perform instructions to solve a task, is coupled with read and write *heads* attached to memory (the *tape*).

While the DNC generally consists of LSTM as its controller, it is much more powerful than LSTM on its own. The authors show that the DNC is capable of learning to perform algorithms, and hence the comparison to a Turing machine can be made – though in the DNC's case, the algorithm is *induced* (learned) on its own, when the DNC is trained with input/output examples, rather than using an explicit table of instructions provided externally as with a Turing machine.

In their experiments, Graves et al. (2016) show that the DNC far outperforms traditional LSTM on a series of tasks. These tasks include simple ones such as copying, sorting, and associative recall, and more complex tasks such as traversing graphs. The DNC is also able of generalizing well on variations of input (particularly longer sequence lengths) that it has not seen during training, in comparison to LSTM.

## 2.5 Gaming Environments

There were three gaming environments that experiments for this research were conducted on. These are detailed below.

### 2.5.1 CarRacing-v0

*CarRacing-v0*[1] is part of the OpenAI Gym environment. It presents a continuous control task in a top-down racing environment. The state consists of a 96x96x3 pixel frame, so the task is to learn from pixels. The primary goal is to quickly drive through a randomly generated track, without going off track. The environment returns a reward of $-0.1$ for every frame, and $+1,000/N$ for every track tile visited, where $N$ is the total number of tiles in the track. The episode finishes when either (a) all tiles are visited, or

[1] https://gym.openai.com/envs/CarRacing-v0/

(b) the agent drives too far off the track into the grass and falls of the side, or (c) 1,000 timesteps pass – whichever happens first. The actions include:

1. Steer: continuous space between -1 to 1, with values closer to 0 meaning no steer, closer to -1 meaning steer left, and closer to 1 meaning steer right.

2. Accelerate: continuous space between 0 to 1, with 0 meaning no acceleration, and 1 meaning maximum acceleration.

3. Break: continuous space between 0 to 1, with 0 meaning no breaking and 1 meaning maximum breaking.

The game is considered solved when an agent is able to achieve a mean reward of 900 over 100 rollouts (episodes).

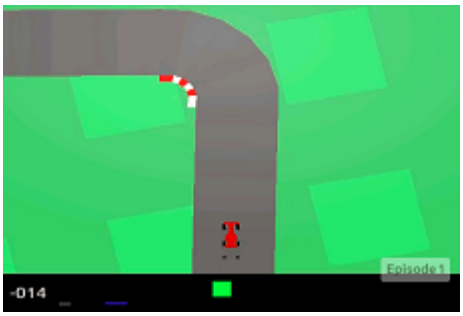



Figure 2.2: Live gameplay from CarRacing-v0. The goal is to navigate through the track as quickly as possible.

### 2.5.2 ViZDoom: Take Cover

*ViZDoom*[2] is a framework intended for research in machine visual learning, and deep reinforcement learning. It is based on the popular first-person shooter video game Doom. *Take Cover* is a particular scenario in ViZDoom, which presents an environment with the player in a closed room facing monsters on the opposite sides. The monsters shoot fireballs, which the player has to dodge by traveling left or right. The primary goal is to live as long as possible by avoiding fireballs. The states again are raw pixels from the gameplay, so the task is to learn to survive the longest using just the pixels. The reward is 1 point every 0.028 seconds the player survives, and the game

[2]http://vizdoom.cs.put.edu.pl/

is considered solved when the agent is able to live for 750 frames, which is approximately 20 seconds[3]. The game consists of the following discrete one-hot encoded actions:

1. Go left: $[1, 0]$.
2. Go right: $[0, 1]$.
3. No action: $[0, 0]$ or $[1, 1]$.

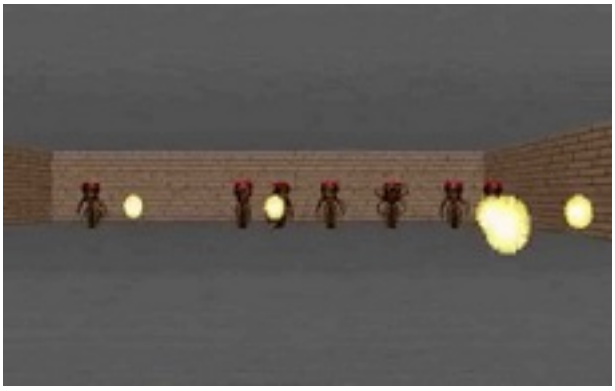


Figure 2.3: Live gameplay from ViZDoom: Take Cover. The goal is to stay alive by dodging fireballs for as long as possible.

### 2.5.3 PommeTeamCompetition-v0

*PommeTeamCompetition-v0* is a custom OpenAI Gym environment. The game is called *Pommerman*[4], which is a variant of the famous 1993 Super Nintendo Entertainment System (SNES) game called Bomberman. It consists of a randomly drawn square board, with four players in each corner that can lay bombs to kill their opponents. The board contains wooden walls and rigid walls, where the wooden walls can be destroyed with bombs to make a passage towards the opponent. The main goal is to be the last player standing.

The variant of Pommerman used for this research is called *Pommerman Team Competition*, which teams up two players on opposite corners. The goal for the team is to

[3] https://gym.openai.com/envs/DoomTakeCover-v0/
[4] https://www.pommerman.com/

work together and kill both players from the other team first. The creators of Pommerman state that their purpose is to explore "how to train agents that can operate in environments with other learning agents, both cooperatively and adversarially".

Pommerman presents a more challenging environment than CarRacing-v0 and ViZDoom: Take Cover. The environment is more dynamic and stochastic than the former, with a lot more happening at once. For example the board changes as bombs go off and new passages are created, whereas in CarRacing-v0 the track remains static and ViZDoom: Take Cover the room is always the same. In Pommerman, the state of the player itself is also crucial, such as how many bombs it has, the strength of the bombs depending on whether the player has picked up power-ups, or whether the player has picked up the ability to kick a bomb.

Each player can take one discrete action at a time:

1. No action: 0.
2. Go up: 1.
3. Go left: 2.
4. Go down: 3.
5. Go right: 4.
6. Lay a bomb: 5.

PommeTeamCompetition-v0 is also different from CarRacing-v0 and ViZDoom: Take Cover because the state is represented as a tuple of several components represented as integers, rather than pixels. So the learning task is different in that the raw pixels are not used, but the integer coded representation of the state is. The state includes the partially observable board from the perspective of an agent, the amount of ammo the agent has available, list of enemies, and more[5].

The rewards received in PommeTeamCompetition-v0 are very sparse. Rewards are received only at the end of each game, when both players of a team are eliminated. The winning team receives a reward of 1 (for both players, regardless of whether one player died and the other still managed to win), and the losing team receives a reward of $-1$. If there is a tie, both teams receive a reward of $-1$. This is in stark comparison to CarRacing-v0 and ViZDoom: Take Cover, where a reward is received at every single timestep.

[5]Full state details for Pommerman can be seen at:
https://github.com/MultiAgentLearning/playground/tree/master/pommerman


Figure 2.4: Live gameplay from PommeTeamCompetition-v0 rendered as pixels for an external observer. The goal for a team of players is to kill the other team's players first. Each player can only observe a portion of the board.

# Chapter 3

# Methods

This chapter aims to detail the methodology of the research undertaken. The conceptual design and actual implementation are described. The tools used are listed. And problems and difficulties encountered through the process are discussed.

## 3.1 Experimentation Framework

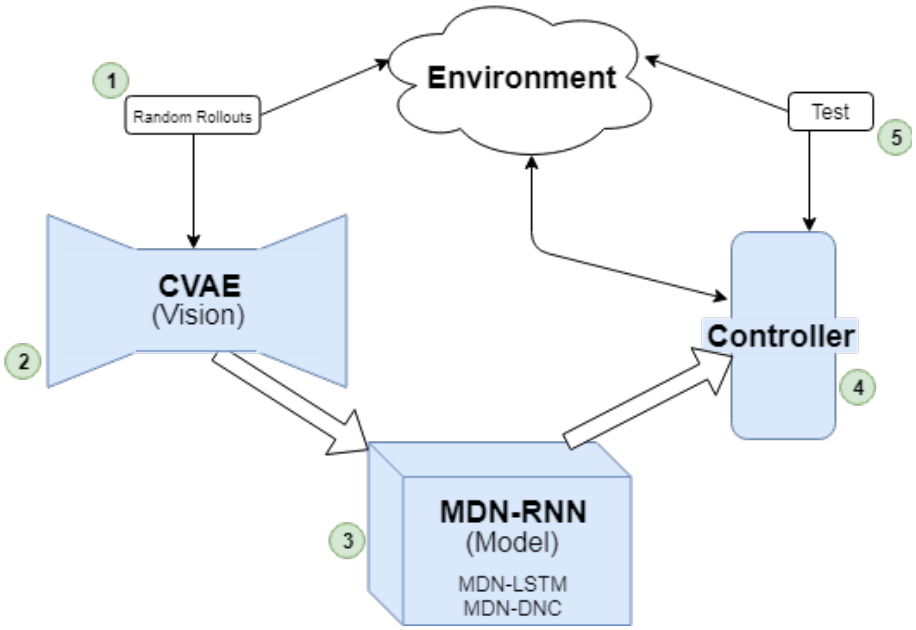



Figure 3.1: The experimentation framework created to carry out research. All steps are detailed in this chapter.

The experiments conducted in this research were carried out in the architecture rep-

resented in Figure 3.1. This architecture is based off of the World Models framework of Ha and Schmidhuber (2018). All components are described in the subsections below, in the order the steps are carried out to perform a Reinforcement Learning experiment using this architecture.

Implementing this architecture as a usable experimentation framework was a challenging feat, given the various components and their complexity. Each component of the experimentation framework was created as an individual Python script: (1) *random_rollouts.py*, (2) *vision.py*, (3) *model.py*, (4) *controller.py*, (5) *test.py*. Each script was executable with command line arguments to run it for a particular game, adjust hyperparameters, toggle GPU usage, and so on.

### 3.1.1 Step 1 - Collecting Random Rollouts

The first step was to gather $10,000$ rollouts (episodes) with an agent performing random actions, in the environment being learned. For this, a multiprocessing Python script, *random_rollouts.py*, was created. The number of child processes spawned were equal to the number of CPU cores available on the system, unless specified otherwise through command line arguments. This sped up data collection as multiple rollouts could be performed in parallel.

For each rollout, the following data was collected:

1. The state pixels from the gameplay at each timestep. In the case of PommeTeamCompetition-v0, it was instead the tuple of integers representing the state.

2. The random action performed at each timestep.

3. The reward received at each timestep.

A number of ways of encoding this data for storage were tested, including storing each frame as an individual image, and storing actions and rewards as CSV. NumPy's compressed *.npz* format was ultimately chosen, as it provided the fastest access to data, with low disk space usage (for CarRacing-v0, $10,000$ rollouts occupied $7GB$ of disk space, almost half of the disk space using other methods).

How actions were selected randomly had important consequences. For example, initially for CarRacing-v0, actions were sampled from a uniform random distribution, and because multiple actions can be performed all at once (steer, break, accelerate), essentially all actions for all frames would equate to a rapid succession of "steer left,

steer right, break, accelerate" at the same time. The car would remain almost stationary, or inch forward in a straight line very slowly. It was clear that this would not lead to a diverse selection of states and rewards being recorded.

After much deliberation, the algorithm for sampling random actions was refined, as shown in Algorithm 1. First, the algorithm repeats the previous action with a $9/10$ probability. This was due to the fact that in the games being learned, frames progressed very rapidly, and an action performed just once during a certain timestep had minimal effects on the state. Secondly, an action can be "forced", meaning that "no action" is not desired, which was useful particularly in CarRacing-v0. And lastly, the algorithm can balance actions – this was useful in ViZDoom: Take Cover, because $a_t = [0,0]$ and $a_t = [1,1]$ have the same meaning: "no action". If these were not balanced among the other actions ($a_t = [1,0]$ and $a_t = [0,1]$), the random agent would be performing "no action" more often than the others. After these improvements were made, a more diverse collection of states were observed during the random rollouts collection phase.

### 3.1.2 Step 2 - Training Vision: CVAE

The second step was to train *Vision* with the data collected from the random rollouts – the frame pixels from each individual state in the case of the pixel based learning task for CarRacing-v0 and ViZDoom: Take Cover, and the raw state integer tuple for PommeTeamCompetition-v0. For CarRacing-v0 and ViZDoom: Take Cover, this involves training a Convolutional Variational Autoencoder (CVAE) with the frame pixels gathered. The idea is to compress each high dimensional state frame $s_t$ down to a low dimensional latent vector $z_t$. In the case of the pixel based learning tasks, each frame is resized to a $64x64x3$ image, which equates to a total dimension of 12288 per state frame. This is further compressed down to a latent $z_t$ dimension of 32 for CarRacing-v0 and 64 for ViZDoom: Take Cover (more details in Chapter 4).

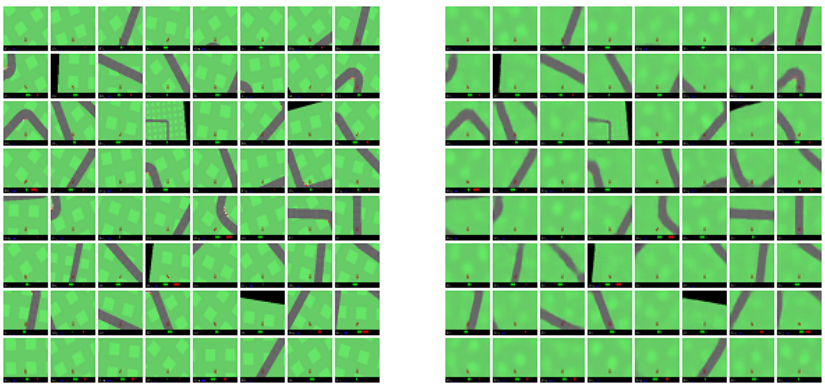
The CVAE was constructed as described by Ha and Schmidhuber (2018), where the same number and size of convolutional layers were used, with the same activation functions at each layer, and so on. It was coded into a script named *vision.py*. Throughout training of the CVAE, samples were generated to allow a visual check of how the network was performing. Randomly selected frames were encoded into $z$ vectors, and then decoded back (reconstructed) to frames, and saved as a collage into a results folder. Examples of these can be seen in Figure 3.2.

After training the CVAE, all states frames from the random rollouts would be pre-

**Algorithm 1** Randomly sampled action algorithm, for Random Rollouts data collection.

**function** GENERATE_ACTION(low, high, prev_action, balance_no_actions, force_actions)
▷ **low**: Array of lowest possible values for all actions, where $a = index(low)$
▷ **high**: Array of highest possible values for all actions, where $a = index(high)$
▷ **prev_action**: Action at previous timestep $t$. At $t_0$, $prev\_action \leftarrow -1$
▷ **balance_no_actions**: Balance $a_t.all() = 0$ and $a_t.all() = 1$ as the same action
▷ **force_actions**: Do not allow $a_t.all() = 0$
▷ **return**: Random action $a_t$
    $repeat \leftarrow (randint(0,9)$ **modulo** $10)$
    **if** $repeat = 0$ and $prev\_action > -1$ **then**
        **return** $prev\_action$
    $action\_len \leftarrow low.length$
    $a_t \leftarrow \vec{0}$
    **while** true **do**
        **for** $i \in range(0, action\_len)$ **do**
            $r \leftarrow randint(low[i], high[i])$
            **if** ($r$ **modulo** $action\_len$) = 0 **then**
                $a_t[i] \leftarrow r$
        **if** $balance\_no\_actions$ **and** $a_t.all() \in \{0,1\}$ **and** randint(0,1)=0 **then**
            $a_t \leftarrow \vec{0}$
            **continue**
        **if** $force\_action$ **and** $a_t.all() \in \{0,1\}$ **then**
            **continue**
        **break**
    **return** $a_t$

processed by *vision.py*, which would encode each frame using the trained Vision to pre-compute the $\mu$ and $\sigma$ for each frame – so the latent could later be sampled as $z \sim \mathcal{N}(\mu, \sigma)$. The collection of $\mu$ and $\sigma$ were saved alongside the random rollout dataset on disk, again using NumPy's compressed .npz format. This pre-computation allowed for faster training in Model and Controller down the pipeline.



(a) Training frames (b) Reconstructed frames

Figure 3.2: CarRacing-v0 Vision samples generated during training of the CVAE.

### 3.1.3 Step 3 - Training Model: MDN-RNN

At this step in the experimentation pipeline, the MDN-RNN for the *Model* component was trained. The goal was to train the MDN-RNN using the data collected from the random rollouts using teacher forcing, where the inputs are the latent $z_t$ and outputs are $z_{t+1}$. More specifically, the network was trained using the ordered sequence of compressed frames for each random rollout (episode) collected in Step 1. The input was $[z_t + a_t]$ – the compressed latent $z_t$ at timestep $t$, concatenated with the random action $a_t$ taken at timestep $t$. The output was $z_{t+1}$, which was simply the next latent $z$ in the sequence for the rollout. This allowed the RNN to learn a predictive model – what frames follow in a sequence, and what affect an action has on the upcoming frame.

The output of the network were the parameters for the mixture density model (the Gaussian mixture model (GMM) in this case): for each mixture, $m$, a set of coefficients $\alpha$, a set of means $\mu$, and a set of variances $\sigma$. The network was trained using the following negative log likelihood as the loss function (Bishop, 1994):

$$\mathcal{L} = -ln\left\{ \sum_{i=1}^{m} \alpha_i(z_t) \frac{1}{\sqrt{2\pi\sigma_i(z_t)}} exp\left\{ -\frac{\|z_{t+1} - \mu_i(z_t)\|^2}{2\sigma_i(z_t)} \right\} \right\} \tag{3.1}$$

$$\sum_{i=1}^{m} \alpha_i(z_t) = 1 \tag{3.2}$$

As the MDN-RNN outputs the GMM parameters $\alpha$, $\mu$, and $\sigma$ given $[z_t + a_t]$ as input, $z_{t+1}$ was sampled from the mixtures. A mixture $m$ was chosen with a probability given the coefficients $\alpha$, and then $z_{t+1}$ was sampled as $z_{t+1} \sim \mathcal{N}(\mu_m, \sigma_m)$. To choose the

mixutre, a weighted random choice can be implemented as shown in Algorithm 2.

**Algorithm 2** Algorithm for choosing a mixture given a set of *coefficients* $\alpha$.

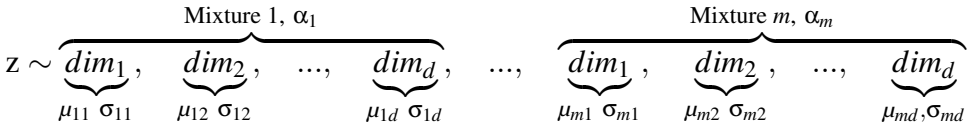
```
function GET_MIXTURE_INDEX(coefficients)
    ▷ coefficients: Array of coefficient α for each mixture
    ▷ returns: Index of mixture selected randomly with probabilities according to coefficients
    sum ← 0.0
    r ← randuniform(0.0, 1.0)
    index ← 0
    for i ∈ range(0, coefficients.length) do
        sum+ = coefficients[i]
        if r ≤ sum then
            index ← i
            break
    return index
```

Training the MDN-RNN was built into *model.py*. The latent $z$'s were re-sampled at every epoch using the pre-computed $\mu$ and $\sigma$ from Vision in Step 2 – to prevent overfitting to specific sampled $z$'s.

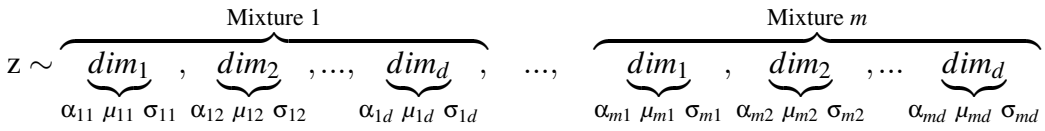
For training the RNN, Truncated Backpropagation Through Time (TBPTT) was used, and some preliminary experiments were conducted to find an ideal sequence length at which to perform TBPTT. A sequence length of 128 seemed to give the best performance. In the case of ViZDoom: Take Cover, the MDN-RNN was made to output an additional prediction on whether the player had been killed by a fireball, called the "*done*" state. Done was added as an additional separate output from the network along with the GMM parameters, and a Sigmoid cross-entropy loss was added to the GMM loss (Equation 3.1) during training.

The GMM coefficients $\alpha$ can be modeled in one of two ways. For a pre-selected number of mixtures $m$ to model a latent $z$ of dimensionality $d$, one coefficient per mixture can be chosen. So if there are 5 mixtures ($m = 5$), then there would be 5 coefficients total. Alternatively, the GMM can be setup to have an individual coefficient for each individual dimension of the latent $z$, therefore the network could learn $m$ x $d$ coefficients in total. An illustration of these two setups is provided in Figure 3.3. It was hypothesized that giving each dimension $d$ for input vector $z$ its own coefficient $\alpha$ would allow the model to be more flexible, so this method of modeling the GMM was

chosen.




(a) Coefficient per whole mixture.




(b) Coefficient per individual dim per mixture.

Figure 3.3: Possibilities for setup of Gaussian Mixture Model parameters output by the MDN-RNN.

When the MDN-RNN was utilized during training of the Controller, it was discovered that the sampling for $z_{t+1}$ was a bottleneck in speed, as it contained several nested math-heavy *for* loops, which can be slow in Python. Several techniques of speeding it up were explored, and eventually Numba[1] was used to compile the code to C++ using the *JIT* (just in time) method. The use of Numba mitigated the speed issue.

Similar to training in Vision, samples of how the MDN-RNN was performing were generated throughout, and saved in a results folder. A sequence of compressed frames and actions from a saved random rollout were input to the network, and the next predicted frame was sampled in order and saved as a collage. This provided a visual spot-check of how the MDN-RNN performed as training progressed.

Furthermore, an option to generate a simulated rollout using the trained MDN-RNN was built in. The simulated rollout would use an initial compressed frame from a real rollout collected in Step 1, and perform pre-configured actions in the simulation up to a number of timesteps. To create a simulation, $[z_t + a_t]$ were input to the MDN-RNN to sample $z_{t+1}$, which was input back into the MDN-RNN with $a_{t+1}$. An illustration of how a simulated rollout was created is shown in Figure 3.4. The simulated rollout would be saved as an animated GIF, and was highly useful in testing the robustness and predictive power of the trained MDN-RNN.

The implementation of the MDN created for this research was attempted on a toy task, as a proof of concept and to ensure it was working correctly. Details of this toy task can be found in Appendix A.1.

[1] https://numba.pydata.org/

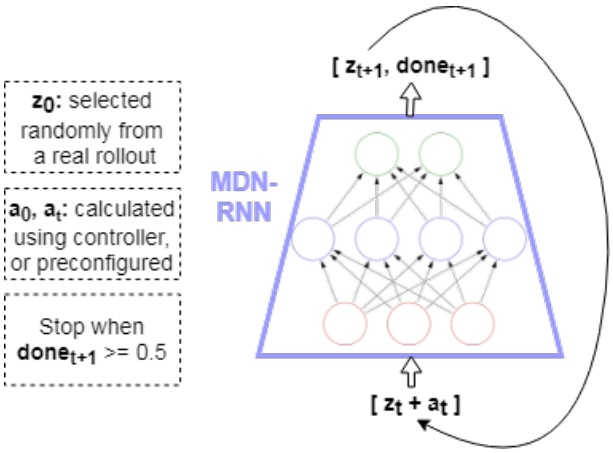



Figure 3.4: Illustration of how a simulated rollout is created for testing the MDN-RNN or training the controller.

### 3.1.4 Step 4 - Training Controller: CMA-ES

The Controller uses Vision to encode frames from an environment into a compressed latent $z$, Model to get the prediction for the next state, and Covariance Matrix Adaptation Evolution Strategy (CMA-ES) to train its own set of parameters, $W$, to perform optimal actions in the environment to maximize the reward.

Everything to do with training the Controller was built into *controller.py*. At the core of it, an implementation of $(\mu/\mu_{I,W},\lambda)$-CMA-ES algorithm was created in Python using the equations and pseudo-code published by Hansen (2016). $(\mu/\mu_{I,W},\lambda)$-CMA-ES was tested on a toy task to ensure it was working correctly. Details of this toy task can be found in Appendix A.2. For brevity, $(\mu/\mu_{I,W},\lambda)$-CMA-ES is simply referred to as CMA-ES from here on in this thesis.

With CMA-ES, an initial seed parameter was sampled from $\mathcal{N}(0,I)$, a "multivariate normal distribution with zero mean and unity covariance matrix" (Hansen, 2016). Mutations to Controller parameters $W$ were created according to covariance matrix $C$ from $\mathcal{N}(m,C)$, where C was adapted as generations progressed. In *controller.py*, a core CMA-ES loop was made responsible for creating a population of these mutations ($\lambda$ mutations in total), and launching processes to test these mutations in parallel. Each sub-process would test the mutation in the environment (or in a simulated environment), with each mutation tested for a number of trials, and return the mean cumulative rewards achieved by the mutation across all trials. The core CMA-ES loop would then adjust the master copy of the Controller parameters $W$ towards the mean of the mutations achieving the top 25% of rewards in that generation.

When testing a mutation $m$ at generation $g$, to get an action $a_t$ at timestep $t$, the action was calculated using the parameters of the Controller $W_{g,m}$ as $a_t = W_{g,m} \cdot [\text{some input}]$. The input was typically the compressed frame $z_t$ (from the environment or simulation), concatenated with the hidden state $h_t$ from the recurrent network in the MDN-RNN. More details for the parameters $W$, inputs chosen, and methods for calculating the action are provided in *Experiments* in Chapter 4 as they varied per environment.

Each mutation was tested multiple times over a number of trials, and the average cumulative reward was passed back to the core CMA-ES loop. This was done to be certain how the mutation performed, due to the stochastic nature of the environments. Otherwise it may be that the mutation got very lucky over a single run and achieved a high cumulative reward, but was not actually fit to perform well generally. This allowed the Controller to learn robust strategies.

The fitness test procedure in *controller.py* for Controller parameter mutations was set up to support two types of fitness tests: using the live environment, or using a simulation of the environment generated with the MDN-RNN (Figure 3.4). When evolving the Controller parameters in a simulated rollout, an initial frame was used from a randomly selected rollout saved to disk in Step 1. This compressed frame $z_0$ was input to the MDN-RNN model along with a first action $a_0$ calculated using the Controller parameters. As the MDN-RNN was trained to produce the mixture parameters for $z_{t+1}$ given input $[z_t + a_t]$, each output $z_{t+1}$ (along with other information such as the hidden state $h_t$ from the MDN-RNN) was used to calculate the next $a_{t+1}$ using the Controller parameters $W_{g,m}$. This allowed for a full simulation of the environment, without ever using the real environment. The simulation would be terminated when the "*done*" flag output by the MDN-RNN reached a certain threshold.

As the Controller evolved, snapshots were saved to disk every few generations (configurable through command line arguments), to ensure no data was lost in case of an issue or glitch. This snapshot included the current Controller parameters $W$, the covariance matrix $C$ and other ancillary scalars and matrices used for CMA-ES, among other information such as generation number and past rewards. The snapshot was saved in the NumPy compressed .npz format, and *controller.py* would automatically resume from the latest snapshot (unless told otherwise through command line arguments), which also allowed seamlessly restarting Controller training on different servers.

Using CuPy[2], a GPU-optimized numerical library similar to NumPy, GPU support (if available) was also built into controller.py to speed up training. This allowed for faster action calculation with the dot product of the Controller's parameters $W$ with inputs from the environment, along with faster processing of the Vision and Model neural networks used in testing mutations in the environment.

#### 3.1.4.1 Distributed Evolution

A strong selling point of Evolution Strategies is that they are very parallelizable (Salimans et al., 2017). As testing each mutation to Controller parameters $W_{g,m}$ in a generation can be costly (computation heavy and slow, particularly for CarRacing-v0), parallelization of the Controller's evolution was sought. A simple strategy to parallelize was created to test each parameter mutation from the whole population in a subprocess of its own. The number of sub-processes spawned were equal to the number of CPU cores available on the computer, unless specified otherwise through command line arguments. This allowed each mutation to be tested in parallel.

Alternatively, it was conceived that multiple machines, or an entire compute cluster, could be used to test each mutation from the entire population of $\lambda$ mutations. As each mutation is tested over a number of trials, and if mapping each trial to a CPU core, a maximum parallel benefit could be achieved using $\lambda * trials$ CPU cores in total. For example, for $\lambda = 64$ and $trials = 16$, a maximum benefit could be achieved with $64 * 16 = 1024$ total CPU cores spread across a cluster of machines.

To that effect, a simple client/server architecture, to perform distributed CMA-ES over a cluster of computers, was devised and built into *controller.py*. A single machine would be assigned as a Dispatcher, and would perform the core loop of CMA-ES to create and distribute mutations and receive results (mean cumulative rewards in the environment being tested), and evolve the master Controller parameters $W$. Workers would receive mutations from the dispatcher, test them in the environment, and return the mean cumulative reward over a number of trials. The mutations and results were passed back and forth over TCP/IP, using NumPy compressed .npz format. An illustration of this distributed evolution architecture can be seen in Figure 3.5.

For this to work, the same technology stack was setup on each node–(a) Git to receive updates for the codebase for this research, (b) Conda to setup the same Python environment, and (c) OpenAI Gym. The dispatcher itself was also made a worker, to

[2]https://cupy.chainer.org/

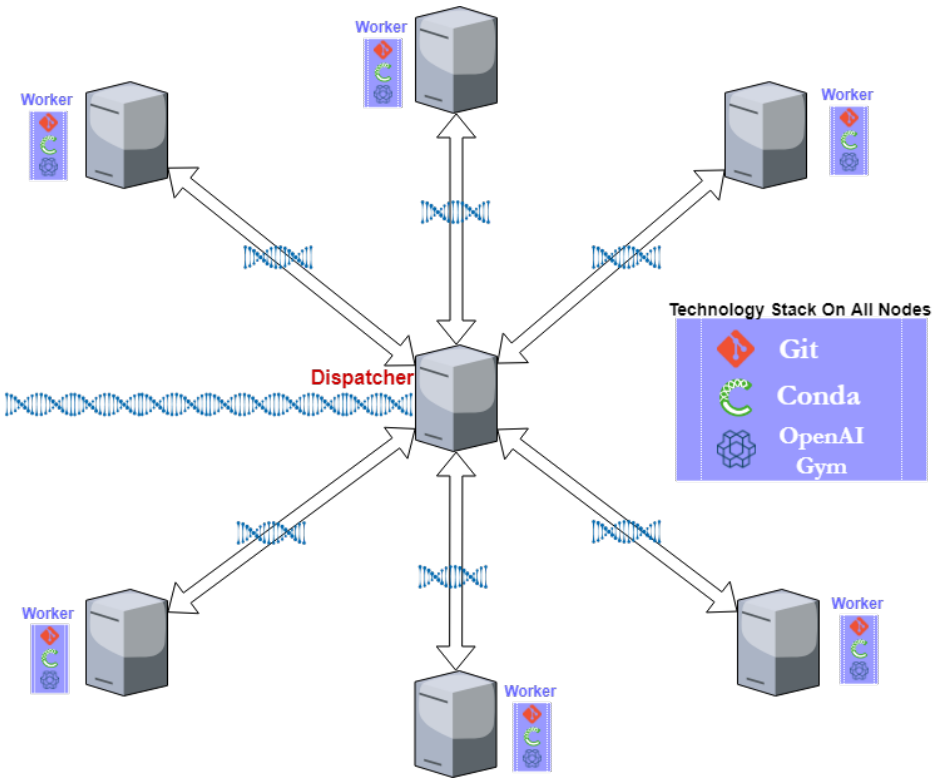



Figure 3.5: Distributed CMA-ES cluster. Dispatcher runs core CMA-ES algorithm, and passes mutations to all workers over TCP/IP.

make use of its own CPUs as well. A cluster of 12 servers with 40-48 CPUs per server was used with distributed evolution architecture, and orders of magnitude speedup in wall-clock time was observed for training of the Controller.

#### 3.1.4.2 Curriculum Learning

It was observed that Controller evolution was slow if full rollouts were used for testing a mutation to Controller parameters $W$ for every single generation. For example, for CarRacing-v0, the agent would spend the majority of its time pointlessly in the grass, in early generations of evolution. This added to wall-clock time for Controller training. It was hypothesized that the Controller could benefit from shorter rollouts early on, and progressively build up to longer episodes once it learned to maximize its reward in shorter episodes. This would allow it to learn a shorter (perhaps *easier*) task first, before moving on to mastering longer tasks. And consequently, it would also speedup wall-clock time of training as shorter rollouts would be used early on.

Thus, *controller.py* was built to optionally apply an elementary and simple form

of curriculum learning for the task being learned. An initial number of maximum timesteps (*max_timesteps*) could be configured, along with a step size (*step_size*). For every generation of evolution, the past mean cumulative reward returned from all mutations was logged – particularly, the highest one only (if a future generation had a lower reward, it was ignored). And *max_timesteps* was increased by *step_size* strictly only if the mean cumulative reward increased from the previous generation. Each trial for a mutation would run up to *max_timesteps* and terminate, returning the cumulative rewards received up until that point.

#### 3.1.4.3 PommeTeamCompetition-v0 adaptations

As PommeTeamCompetition-v0 presented a very different task than CarRacing-v0 and ViZDoom: Take Cover, the Controller had to be adapted to handle the differences. The team competition involves teams of two players, and hence a pair of two Controller parameters had to be created – one for each player. Each of the parameters had their own copy of the ancillary CMA-ES scalars and vectors, and evolved "independently", but both were tested as a team in the live environment. For trials, each of the mutations (from two populations) to each of the two parameters were paired, their actions calculated independently, but performed in the same instance of the game. Their cumulative rewards were collected and reported back to the core CMA-ES loop, which then would use the paired parameter mutations that achieved the top 25% of rewards in each generation, to adjust the master copies of the pair towards the mean of the winning mutations.

Each individual agent in the team can be placed in any one of four corners (with the condition that the partner is in the kitty-corner diagonally across the square board). Thus, when running a trail, the locations of the agents were randomized, within the team, and on the board, so an agent would not overfit to being exactly in the same corner always.

The Controller was adapted to use self-play when running a trial with a pair of mutations, where the opposing team was an older "frozen" version of the pair of parameters. The frozen pair was updated every few generations (treated as a hyperparameter). Thus the complexity of the Controller codebase increased even more, having to track two individual pairs of parameters. A version of the environment trial was also made to use a deterministic team of agents as the opposing team, instead of using self-play. The PommeTeamCompetition-v0 codebase includes an agent called SimpleAgent, which turns out to be not very simple – it contains $\sim$ 500 lines of code filled

with logic, rules, and even a version of Djikstra's graph search algorithm[3].

### 3.1.5 Step 5 - Testing

The final step in the experimentation framework was to test the trained Controller in the real environment over a number of rollouts, to get the mean reward (along with the standard deviation) it was able to achieve in the environment. For that purpose, a *test.py* script was created, which would run the Controller through the environment. *test.py* used functions from *controller.py* to calculate the action at every timestep.

The script was configured to run in parallel, using a number of subprocesses equal to the CPU cores on the system (unless specified otherwise through command line arguments). An optional command line argument allowed recording each rollout as an animated GIF and saving it to disk. After evaluation, the script reported back the mean cumulative reward and standard deviation over all rollouts, along with the cumulative rewards received per episode. These results were also recorded to disk in a NumPy compressed .npz format.

This setup made it easy to test a trained Controller and even observe visually how the agent performed in the real environment[4].

## 3.2 Differentiable Neural Computer

As the primary objective of this research was to investigate the use of the Differentiable Neural Computer (DNC) to learn a predictive, probabilistic model of an environment, an implementation of the DNC in Chainer that could plug in neatly with the codebase for this research was necessary. Furthermore, it was thought to be beneficial to test the implementation and get a "feel" of the DNC by testing it in simpler toy tasks first, and observe how it behaves and performs in comparison to traditional LSTM. Thus, a thorough effort was undertaken on implementing and testing the DNC separately before using it in this research.

[3] https://github.com/MultiAgentLearning/playground/blob/master/pommerman/agents/simple_agent.py

[4] An example recorded rollout of a trained Controller in CarRacing-v0 can be seen at: https://raw.githubusercontent.com/AdeelMufti/WorldModels/master/asset/CarRacing-v0_rollout.gif

### 3.2.1 Implementation

There was an implementation of a DNC coded in Chainer, available publicly on GitHub by a user named *yos1up*[5]. This codebase for the DNC was useful as a starting point, particularly given the care the author took to keep it simple and match the names of variables and operations to equations laid out in the DNC publication by Graves et al. (2016). However, there were a couple of problems with this implementation, with the main one being speed of training in wall-clock time.

The primary reason for the slowness of *yos1up*'s implementation of the DNC was the use of nested *for* loops in Python to interface with NumPy in non-vectorized ways, which is a well-known cause for drastic slowdown in the execution of code. NumPy vectorization means that operations on NumPy arrays are done through compiled C code containing *for* loops. For example, a vectorized NumPy dot product offers a 70x speedup over a Python *for* loop version (van der Walt et al., 2011).

With some testing, it was estimated if this DNC implementation was plugged into the Model component in the experimentation framework, it would take 52 days per single epoch of training, where the approach of Ha and Schmidhuber (2018) calls for 20 epochs. So training a full Model would be completely unfeasible. Therefore, to get around this speed issue, an optimized implementation was created by using *yos1up*'s code as a starting point, and baseline. A final optimized implementation ultimately achieved a **50x** speedup over *yos1up*'s baseline.

### 3.2.2 Optimization

Cython and Numba can be used to translate Python code to C/C++ and compiled to run much faster. Already this research project had employed the use of Numba to speedup sampling from the MDN-RNN. So as a starting point, parts of the DNC code, particularly the *for* loops in the inner computation mechanism, were (1) ported to Cython or (2) decorated with Numba JIT (after some modifications). But only a moderate speedup was observed, likely due to the fact that the compiled *for* loops were still calling further *for* loops (internal to the NumPy library) when manipulating the NumPy arrays. So other approaches were sought.

Alternatively, Rae et al. (2016) lay out an approach for implementing a "Sparse Differentiable Neural Computer". It uses Compressed Sparse Row (CSR) and Compressed Sparse Column (CSC) formats for some of the DNC's internal matrices (for

[5] https://github.com/yos1up/DNC

example the temporal memory linkage matrix which happens to be sparse), along with a sparse memory management scheme that tracks memory usage. Rae et al. (2016) observed a 400x speedup using a memory matrix with 2,000 slots. This approach was investigated but not pursued because the SciPy (for CPU) and CuPy (for GPU) CSR and CSC matrix formats did not plug into Chainer.

For writing optimized NumPy code in Python, van der Walt et al. (2011) suggest "vectorizing calculations, avoiding copying data in memory, and minimizing operation counts". So this approach was closely followed. Ultimately what led to the most significant speedup was to vectorize the code and avoid any *for* loops in the inner computation mechanism of the DNC altogether. This was achieved with some careful and creative manipulation of NumPy arrays and the necessary mathematical operations, using only vectorized NumPy functions. Essentially, major parts of the implementation had to be recreated and tested to support these changes.

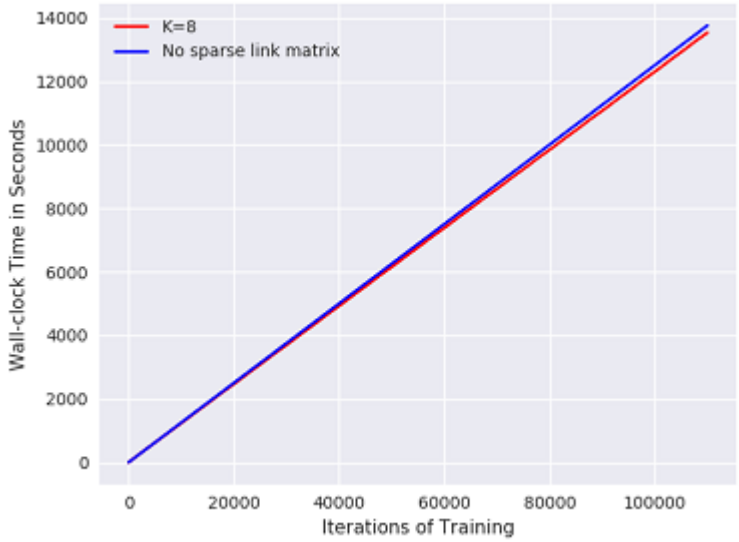



Figure 3.6: Sparse link matrix optimization benefits in wall-clock time for training the DNC.

Graves et al. (2016) point out a way to use a sparse link matrix (which connects related items stored in memory together), rather than a dense matrix. For this, a CSR/CSC matrix was not necessary, and benefits could be gained just through NumPy slicing of the matrix to ignore the sparse bits. With the proposed changes, computation costs related to the link matrix could be reduced from $O(n^2)$ to $O(n \log n)$, without sacrificing performance in error/accuracy. This improvement was implemented, and sped

up the implementation a bit. It is predicted that for large memory sizes and long training requirements, this improvement can make a significant dent in wall-clock time savings. A graph of the speedup achieved can be seen in Figure 3.6, where a small memory size was used for testing on a toy task.

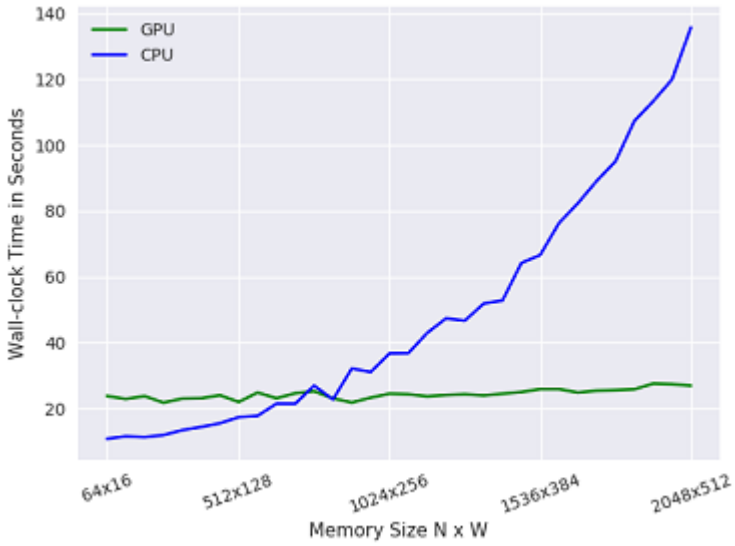



Figure 3.7: CPU vs GPU benefits in wall-clock time of training the DNC as memory size increases.

GPU acceleration can speedup neural network training by orders of magnitude. A major reason for this speedup is that GPUs can perform massively parallel matrix computations especially on batches of matrices together, which is common in mini-batch stochastic gradient descent training of neural networks. However, the DNC can only be trained with a batch size of one (the reason follows below). Still, it was predicted that performing many of the matrix operations for a single input while training the DNC could benefit from GPU acceleration. Fortunately, CuPy (Chainer's GPU accelerated numerical library) is meant to be a drop-in replacement for NumPy, so wherever NumPy was used, CuPy could simply be swapped in. Though this did not work out exactly perfectly; some changes still needed to be made. In the end, GPU support did not give any benefit over CPU for smaller memory matrix sizes for the DNC (in fact, it degraded performance) – but it did offer a significant speedup as the memory size increased. A graph of the speedup achieved with GPU acceleration as memory size increased, can be seen in Figure 3.7.

The reason the DNC needs to be trained with a mini-batch size of one is because for

every single input passed through the network to calculate the gradient during training, the internal state of the DNC needs to be manipulated. Particularly, the controller inputs the external input, and its own data $r_t$ (where $r_0 = 0$) read from the external memory at the end of the previous timestep (i.e. the previous input in a sequence). Still, mini-batch processing was attempted by meticulously implementing it – changes to every single matrix operation in the DNC were required. Though a major wall-clock speedup was achieved with larger batch sizes (2, 8, 16, 32), especially with GPU acceleration, a major degradation in performance in error/accuracy were also observed. This is likely because the data contained at timestep 1 in $r_1$ would not be representative of the actual timestep in the input sequence as a batch of inputs were processed in the previous pass. So this approach was abandoned.

With all the optimizations to the DNC (the most significant one being vectorized NumPy matrix operations), an average speedup of around 50x was observed on some toy tasks, and this would allow training the MDN-RNN a lot more feasible for these experiments. Details of the toy tasks used to test the final optimized implementation can be found in Appendix A.3.

### 3.2.3 Integration With Model

Integrating the DNC in the Model component was straightforward. Where the traditional LSTM was previously used in hidden layer for the MDN-RNN, it was simply switched to the DNC implementation, which was possible because the DNC was extended from the Chainer *Chain* class. Hence, DNC support was built in to the experimentation framework, and the MDN could be toggled to switch between LSTM and DNC as its recurrent layer. This toggle could be done from command line arguments from which the hyperparameters (such as memory size) for the DNC could also be provided. *model.py*, *controller.py*, and *test.py* were all adapted to be able to switch between MDN-LSTM and MDN-DNC.

## 3.3 Challenges and Difficulties

There were a number of challenges and difficulties faced when creating these implementations and carrying out the experiments, some of which are described below.

### 3.3.1 Large Datasets

$10,000$ random rollouts were collected by a random agent and saved to disk. For ViZDoom: Take Cover, and PommeTeamCompetition-v0, each rollout had $\sim 500$ states, and $1,000$ states in the case of CarRacing-v0. Choosing how to store this data was an important decision for ease of use, speed, and disk space. Ultimately NumPy's compressed .npz format was chosen.

For training the Vision and Model components, the data needed to be loaded into memory. However, a major obstacle was that the computer on which training was being done (even AWS instances with large RAM) would run out of memory if attempting to load all $10,000$ rollouts at once into memory from disk. The problem was resolved by implementing a custom data loader class used by *vision.py* and *model.py* that allowed loading data in batches instead. Chainer and the code to train the CVAE and MDN-RNN would be agnostic of the batched loading, and would simply iterate through the data in the custom data loader, which would handle loading from disk in chunks. The data loader had to maintain its own internal state, and carefully keep the vast amount of data "aligned" as well – when training Model, it was paramount to have each rollout's compressed frames ordered in a sequence, and matched with the right action, which was made difficult by the fact that the $10,000$ rollouts would be shuffled during training. Implementing all this was a challenging task.

Loading the data from disk was also very slow, as it needed to be preprocessed and stacked into arrays. So the custom data loader was parallelized, by spawning subprocesses to do the loading and computation.

### 3.3.2 Controller Action Calculation

In the World Models approach of Ha and Schmidhuber (2018), actions can be calculated in the following ways:

1. $a_t = tanh(W \cdot [z_t + h_t] + b)$
2. $a_t = tanh(W \cdot [z_t + h_t + c_t])$

where $W$ (weights) and $b$ (bias) are the parameters of the controller, $z_t$ is the compressed latent vector received from the environment or simulated environment, and $h_t$ and $c_t$ are the hidden and cell states of the MDN-RNN. In method (1), there is a scalar output for each individual action. For method (2), a singular scalar reduced to $(-1,1)$ is output which is treated as a number falling in the range $[-1,1]$, which is divided into

parts equal to the number of actions, to get the ultimate action. For example if there are 3 actions, $a_t$ can be calculated as:

1. $a_t = 0 : -1 < tanh(W \cdot [z_t + h_t + c_t]) <= -0.33$

2. $a_t = 1 : -0.33 < tanh(W \cdot [z_t + h_t + c_t]) <= 0.33$

3. $a_t = 2 : 0.33 < tanh(W \cdot [z_t + h_t + c_t]) <= 1$

PommeTeamCompetition-v0 uses 6 actions, whereas CarRacing-v0 and ViZDoom: Take Cover both have 3, which presented a problem with these methods of action calculation. With #1, there were too many parameters in the Controller for CMA-ES, which slowed it down. With #2, empirically $W \cdot [z_t + h_t + c_t]$ produced values far in the negative, or far in the positive, and rarely close to 0. This was a problem because after $tanh()$, values were close to $-1$ or 1, which caused the controller to only perform two actions. For ViZDoom: Take cover, this was not an issue because there were really only two actions (go left or go right), the third action was "no action" – so the problem was modeled as when closer to -1, go left, when closer to +1, go right.

Thus for Pommerman, a number of approaches for action calculation were attempted, including using sigmoid with adjusted slope (Ivanikovas et al., 2009) instead of tanh, or using sine as an activation function. Though in the end, approach #1 was used with the trade-off that training the Controller was slower in wall-clock time.

### 3.3.3 Technical

There were some purely technical issues that had to be overcome.

1. The ViZDoom wrapper for OpenAI Gym had been deprecated and did not plug in with the latest OpenAI Gym version[6]. Hence, ViZDoom has to be compiled on its own separately, and a custom wrapper was created just for this research.

2. There was a bug in CarRacing-v0 render code which caused pixels not to get rendered when not in human-mode. This bug was tracked down and a workaround was created[7].

[6] https://github.com/ppaquette/gym-doom/

[7] The workaround was submitted to: https://github.com/openai/gym/issues/976#issuecomment-395486438

3. ViZDoom and some related libraries would not compile on the University of Edinburgh Informatics Scientific Linux environment, due to other missing libraries and packages. Without root access, nothing could be done. Rather than chase down the right person who potentially may not allow these libraries to be installed on the compute and GPU clusters anyway, being pressed for time, a Virtual Machine of Scientific Linux was created locally. The point was just to be able to compile the libraries and ViZDoom, and copy them to the Informatics servers. The C++ code of the libraries had to be modified slightly so they would work out of a relative path rather than absolute path (which would require them to be placed in folders with root access only).

## 3.4 Tools and Technical Setup

Python was chosen as the programming language to create the implementations. This is because Python is widely used and supported in the Machine Learning / Artificial Intelligence research community. There are several optimized Deep Learning frameworks available for Python. NumPy, a popular fundamental package for numeric computing, is built for Python. And in Python it is easy to explore new ideas and code prototypes quickly given its minimal syntax and coding overhead.

Chainer was chosen as the Deep Learning framework to implement and conduct this work. The primary reason for this choice was recommendation from others in the scientific community. According to Chainer's website[8], Chainer is a "powerful, flexible, and intuitive framework for neural networks", and "supports various network architectures including feed-forward nets, convnets, recurrent nets and recursive nets."

PyCharm is a powerful Integrated Development Environment (IDE) for Python, and was exclusively used for prototyping and development in Python. It has many features that make programming easier, such as: code completion hints, seamless version control integration, code organization, and more. BitBucket offers free Git (a code version control system) repository hosting, and was used extensively through the project.

MSc Informatics students at the University of Edinburgh have access to a GPU cluster, and a compute cluster. The GPU cluster consists of around 200 NVidia 1060 Ti GPUs. And the compute cluster, part of the Extreme Computing course at the University of Edinburgh, consists of 12 servers with 40 or 48 Intel Xeon E5-2650

[8] https://chainer.org/

CPUs on each machine. When either of these resources was not available, paid cloud computing services from Amazon Web Services (AWS) were used – particularly, the c5.xxx instances for CPU compute, and g4.xxx instances for GPU compute.

To facilitate experimentation, a series of shell scripts were created for various tasks such as copying code to servers, copying data and results to and from the servers, copying data between servers, and so on. This made an optimized pipeline to quickly execute experiments.

While there exist tools to organize and log experiments, such as *Sacred*, experiments were organized and logged in a simple text document instead. Outputs of the training logs, and the results themselves (usually trained models) from each experiment were kept organized in folders.

# Chapter 4

# Experiments

This chapter covers the details of the experiments that were carried out for this research. Since all experiments were based on a particular environment, the chapter is broken down into sections for each environment.

As the objective of the research was to investigate the benefits of an external memory augmented neural network, particularly the Differentiable Neural Computer (DNC), for each environment two categories of experiments were conducted:

1. **MDN-LSTM:** The experimentation framework defined in Chapter 3 using Long Short Term Memory (LSTM) as the recurrent layer for the Mixture Density Network (MDN) in the Model component. These would form a baseline for comparison to what improvements were offered by the MDN-DNC.

2. **MDN-DNC:** Instead of LSTM, this set of experiments used the DNC as the recurrent layer for the MDN in the Model component.

Note that in some cases multiple experiments had to be carried out to fine-tune certain aspects of a component, and are alluded to in the Results chapter but omitted here for the sake of brevity. This was particularly true for PommeTeamCompetition-v0.

## 4.1 CarRacing-v0

### 4.1.1 MDN-LSTM

#### 4.1.1.1 1. Collect Random Rollouts

**Experiment Tag:** [cr0 / mdn-lstm / random_rollouts]

Using the random action policy defined in Algorithm 1, 10,000 random rollouts were collected. For action sampling, *force_actions*=true was used to bias the car to move to collect a variety of scenarios. The frames were resized to 64x64x3 and saved to disk, along with the actions and rewards, in NumPy compressed .npz format. The rollouts were collected on a server containing a total of 48 Intel Xeon E5-2650 CPU cores, and hence 48 parallel subprocesses were spawned.

#### 4.1.1.2 2. Train Vision

**Experiment Tag:** [cr0 / mdn-lstm / vision]

The CVAE in Vision was trained using the settings and hyperparameters defined in Table 4.1. Training was done using a GPU. The data loader shuffled all frames and loaded data into memory in batches of 200 rollouts.

| Setting | Value |
|---|---|
| Epochs | 1 |
| Latent dim | $z \in \mathbb{R}^{32}$ |
| Mini-batch size | 100 |
| Optimizer | Adam, lr=0.001 |

Table 4.1: [cr0 / mdn-lstm / vision] settings.

#### 4.1.1.3 3. Train Model

**Experiment Tag:** [cr0 / mdn-lstm / model]

The MDN-LSTM in Model was trained using the settings and hyperparameters defined in Table 4.2. Training was done using a GPU. The data loader shuffled all 10,000 rollouts per epoch, but preserved the sequence of latent $z$'s and actions per individual rollout. Data was loaded into memory in batches of 1,000 rollouts.

#### 4.1.1.4 4. Train Controller

**Experiment Tag:** [cr0 / mdn-lstm / controller]

The Controller was trained using CMA-ES run across a distributed compute cluster (Figure 3.5) consisting of 12 machines with 40-48 Intel Xeon E5-2650 CPU cores per

| Setting | Value |
|---|---|
| Epochs | 20 |
| LSTM hidden dim | 256 |
| Gaussian mixtures | 5 |
| TBPTT sequence length | 128 |
| Optimizer | Adam, lr=0.001 |
| Predict *done* | False |

Table 4.2: [cr0 / mdn-lstm / model] settings.

machine (a total of 520 CPU cores). Ideally if each machine had a GPU, it could be utilized when processing Vision and Model, and calculating the actions – but no GPUs were available on this compute cluster. The controller was trained using the settings and hyperparameters defined in Table 4.3.

| Setting | Value |
|---|---|
| CMA-ES population size | $\lambda = 64$ |
| CMA-ES fittest genes | $\mu = 1/4$ |
| CMA-ES target fitness | Mean cumulative reward of 900 |
| Trials per mutation | 16 |
| Number of actions | 3 |
| Action dim | $a \in \mathbb{R}^3$ |
| Parameters dim | $W \in \mathbb{R}^{864}$, $b \in \mathbb{R}^a$ (867 in total) |
| Action calculation | $tanh(W \cdot [z_t + h_t] + b)$ |
| Curriculum | initial max timesteps=50, step=5 |
| MDN temperature* | $\tau = 1.0$ |
| Environment type | OpenAI Gym |

Table 4.3: [cr0 / mdn-lstm / controller] settings.

*Ha and Schmidhuber (2018) use *temperature* ($\tau$) when sampling from the MDN-RNN to increase or decrease uncertainty. The temperature adjusts the GMM coefficients to make the dominant mixture most probable with $\tau < 1.0$, or make all mixtures more equally probable with $\tau > 1.0$. And the standard deviation is narrowed with $\tau < 1.0$, and widened with $\tau > 1.0$.

#### 4.1.1.5 5. Test

**Experiment Tag:** [cr0 / mdn-lstm / test]

The final step was to test how well the trained agent performed in the real environment. This entailed putting the trained Vision, Model, and Controller through the environment and logging the cumulative rewards for each trial, over 100 trials – the final mean and standard deviation of all cumulative rewards over all trials would serve as a measure of how well the trained agent was performing in general.

The mean cumulative rewards achieved by the agent could be compared against those received by a random agent. A random agent was set up as follows. At every timestep in the environment, the agent calculates its action as $a_t = tanh(W \cdot [z_t + h_t] + b)$, using $W$ and $b$ parameters evolved through CMA-ES. To test how a random agent would perform, the Controller's pararameters $W$ and $b$ were initiated from a random unit normal distribution, thus only meaningless actions would be calculated at any timestep. The random agent was also run over 100 trails in the environment, and the mean and standard deviation over all cumulative rewards were recorded for comparison. Note that while a random Controller was used, the trained Vision and Model were used.

### 4.1.2 MDN-DNC

Experiments with MDN-DNC were started for CarRacing-v0, but due to limited time available for conducting this research, and slow training of the DNC, they were halted. It was guessed that maximum benefit and ideal testing for the predictive modeling capabilities of the MDN-DNC would be gained in a simulated training environment generated by the MDN-DNC (possible using *model.py* as described in Section 3.1.3 and illustrated in Figure 3.4). Since experiments using simulated training were to be done for Doom, that is where the MDN-DNC experimentation efforts were focused.

## 4.2 ViZDoom: Take Cover

### 4.2.1 MDN-LSTM

#### 4.2.1.1 1. Collect Random Rollouts

**Experiment Tag:** [dtc / mdn-lstm / random_rollouts]

Using the random action policy defined in Algorithm 1, $10,000$ random rollouts were collected. For action sampling, *balance_no_actions*=true was used to ensure that "no

action" ($[0,0]$ and $[1,1]$) were balanced against taking an action at a probability of $1/3$. The frames were resized to 64x64x3 and saved to disk, along with the actions and rewards, in NumPy compressed .npz format. The rollouts were collected on a server containing a total of 48 Intel Xeon E5-2650 CPU cores, and hence 48 parallel subprocesses were spawned.

#### 4.2.1.2 2. Train Vision

**Experiment Tag:** [dtc / mdn-lstm / vision]

The CVAE in Vision was trained using the settings and hyperparameters defined in Table 4.4. Training was done using a GPU. The data loader shuffled all frames and loaded data into memory in batches of 200 rollouts.

| Setting | Value |
|---|---|
| Epochs | 1 |
| Latent dim | $z \in \mathbb{R}^{64}$ |
| Mini-batch size | 100 |
| Optimizer | Adam, lr=0.001 |

Table 4.4: [dtc / mdn-lstm / vision] settings.

#### 4.2.1.3 3. Train Model

**Experiment Tag:** [dtc / mdn-lstm / model]

The MDN-LSTM in Model was trained using the settings and hyperparameters defined in Table 4.5. Training was done using a GPU. The data loader shuffled all 10,000 rollouts per epoch, but preserved the sequence of latent $z$'s and actions per individual rollout. Data was loaded into memory in batches of $1,000$ rollouts.

At the end of training, animated GIFs were generated in a simulation with predefined deterministic actions, to get a visual test of how the model was performing (whether it was responsive to the actions, whether it had learned the dynamics of the environment, and so on).

#### 4.2.1.4 4. Train Controller

**Experiment Tag:** [dtc / mdn-lstm / controller]

| Setting | Value |
|---|---|
| Epochs | 20 |
| LSTM hidden dim | 512 |
| Gaussian mixtures | 5 |
| TBPTT sequence length | 128 |
| Optimizer | Adam, lr=0.001 |
| Predict *done* | True |

Table 4.5: [dtc / mdn-lstm / model] settings.

The Controller was trained using CMA-ES run across a distributed compute cluster (Figure 3.5) consisting of 12 machines with 40-48 Intel Xeon E5-2650 CPU cores per machine (a total of 520 CPU cores). Ideally if each machine had a GPU, it could be utilized when processing Vision and Model, and calculating the actions – but no GPUs were available on this compute cluster. The Controller was trained using the settings and hyperparameters defined in Table 4.6.

Training was significantly different from [cr0 / mdn-lstm / controller] because it was performed using simulations of the environment instead of in the real environment itself. In the simulation, a +1 reward was given for every timestep alive. The game was considered done (agent killed) when the additional *done* flag output by the MDN-LSTM reached a certain threshold. To prevent a run-away simulation that kept going forever, the simulation was terminated after a certain number of timesteps.

Through training generations, animated GIFs were generated in a simulation using actions calculated by the trained Controller. This allowed visually observing what the controller had learned to do in the simulated environment, which was useful down the line.

#### 4.2.1.5 5. Test

**Experiment Tag:** [dtc / mdn-lstm / test]

The final step was to test how well the trained agent performed in the real environment. This entailed putting the trained Vision, Model, and Controller through the environment and logging the cumulative rewards for each trial, over 100 trials – the final mean and standard deviation of all cumulative rewards over all trials would serve as a measure of how well the trained agent was performing in general. The mean cumulative rewards achieved by the agent could be compared against those received by a random

| Setting | Value |
|---|---|
| CMA-ES population size | $\lambda = 64$ |
| CMA-ES fittest genes | $\mu = 1/4$ |
| CMA-ES target fitness | Mean cumulative reward of 2100 |
| Trials per mutation | 16 |
| Number of actions | 3 |
| Action dim | $a \in \mathbb{R}^1$ |
| Parameters dim | $W \in \mathbb{R}^{1088}$, no bias |
| Action calculation | $tanh(W \cdot [z_t + h_t + c_t])$ (Section 3.3.2) |
| Curriculum | initial max timesteps=500, step=10 |
| MDN temperature | $\tau = 1.15$* |
| Environment type | Simulation |
| Simulation max length | 2100 |
| Simulation done threshold | 0.5 |
| Initial $z$ noise | $z+ = \mathcal{N}(0, 0.5)$* |

Table 4.6: [dtc / mdn-lstm / controller] settings. *Multiple values were tried, but ultimately the value provided was chosen. More details in Results, Chapter 5.

agent. The random agent's mean cumulative rewards were collected as described in Section 4.1.1.5.

Since the Controller was being trained in simulations of the environment generated by the MDN, an individual 100 trial test was performed with the Controller as it evolved over the generations, to see how it performed in the real environment. The test was done at every $5^{\text{th}}$ generation and mean cumulative rewards and standard deviations were logged.

### 4.2.2 MDN-DNC

#### 4.2.2.1 1. Collect Random Rollouts

The same dataset of $10,000$ random rollouts from [dtc / mdn-lstm / random_rollouts] was used for this set of experiments.

#### 4.2.2.2 2. Train Vision

As the primary reason for conducting these experiments was to test how Model and Controller performed when augmented with the DNC, it was not necessary to retrain the Vision, and more useful to keep it consistent. Hence the same trained Vision from [dtc / mdn-lstm / vision] was used.

#### 4.2.2.3 3. Train Model

**Experiment Tag:** [dtc / mdn-dnc / model]

The MDN-DNC in Model was trained using the settings and hyperparameters defined in Table 4.7. Training was done using a GPU. The data loader shuffled all 10,000 rollouts per epoch, but preserved the sequence of latent $z$'s and actions per individual rollout. Data was loaded into memory in batches of $1,000$ rollouts.

While training was planned for 20 epochs, it was stopped early due to time constraints given the fact that the DNC is slower to train for reasons detailed in Section 3.2.2.

| **Setting** | **Value** |
|---|---|
| Epochs | 20 |
| LSTM hidden dim* | 512 |
| Gaussian mixtures | 5 |
| TBPTT sequence length | 128 |
| Optimizer | Adam, lr=0.0001 |
| Predict *done* | True |
| DNC memory slots | $N = 256$** |
| DNC memory width | $W = 64$** |
| DNC read head | $R = 4$** |

Table 4.7: [dtc / mdn-dnc / model] settings. *The DNC's controller consists of an LSTM. **These were chosen as they were used in the majority of experiments by Graves et al. (2016).

At the end of training, animated GIFs were generated in a simulation with predefined deterministic actions, to get a visual test of how the model was performing.

#### 4.2.2.4 4. Train Controller

**Experiment Tag:** [dtc / mdn-dnc / controller]

Training the controller was mostly the same as done for [dtc / mdn-lstm / controller]. The major difference was that instead of using a MDN-LSTM, a MDN-DNC was used for the Model component. Another difference was that instead of directly injecting Gaussian noise into the initial frame $z$ used for the simulation ($z+ = \mathcal{N}(0, 0.5)$), the MDN temperature was set to $\tau = 4$. And similar to the other experiment, animated GIFs were generated in a simulation using actions calculated by the Controller as it evolved, to visually observe what it had learned to do in the simulations.

#### 4.2.2.5 5. Test

**Experiment Tag:** [dtc / mdn-dnc / test]

Tests over 100 trials were performed in the real environment, at every $5^{th}$ generation of Controller evolution, as done for [dtc / mdn-lstm / test].

## 4.3 PommeTeamCompetition-v0

### 4.3.1 MDN-LSTM

PommeTeamCompetition-v0 was significantly different than the other tasks – state was presented as a tuple of integers rather than pixels, and the task involves multi-agent learning. Thus, educated guesses and some trial and error with preliminary experiments had to be made to design these experiments.

#### 4.3.1.1 1. Collect Random Rollouts

**Experiment Tag:** [ptc / mdn-lstm / random_rollouts]

Using the random action policy defined in Algorithm 1, $10,000$ random rollouts were collected. PommeTeamCompetition-v0 returns 4 full states, with 1 from the partially observed perspective of each of the 4 players. Thus, in fact, each individual rollout can be treated as 4 rollouts, and the $10,000$ rollouts can be divided logically as $40,000$ individual rollouts.

The states were stored on disk using Python Pickle (an object serialization method), compressed with gzip. The rollouts were collected on a server containing a total of 48 Intel Xeon E5-2650 CPU cores, and hence 48 parallel subprocesses were spawned.

#### 4.3.1.2 2. Train Vision

The custom data loader had to be adapted to convert the state tuple consisting of sets of integers into a "frame" – all the integers in the tuple flattened and concatenated into a single integer vector. Each frame had the dimensionality $\mathbb{R}^{375}$, which was much smaller than the other tasks where every frame was of dimensionality $\mathbb{R}^{12288}$ (64x64x3 image).

Since the frames for this game were represented not as pixels, a Convolutional VAE was not appropriate to use in Vision. A simpler VAE and even a traditional autoencoder (AE) could be used. The choice a trying a non-variational autoencoder was based on the fact that the state space was so small for this task (375 vs 12288), small variations in the state could potentially have more dramatic effects compared to a few pixels being off otherwise.

Thus, two versions of Vision were trained to use with experiments further down the experimentation pipeline.

**Experiment Tag:** [ptc / mdn-lstm / vision-vae]

Several variations of a VAE architecture were attempted in preliminary experiments, and ultimately the one represented in Table 4.8 was used.

| Setting | Value |
|---|---|
| Epochs | 10 |
| Latent dim | $z \in \mathbb{R}^{16}$ |
| Mini-batch size | 100 |
| Optimizer | Adam, lr=0.001 |
| Hidden layers | 1 |
| Hidden dim | 512 |

Table 4.8: [ptc / mdn-lstm / vision-vae] settings.

**Experiment Tag:** [ptc / mdn-lstm / vision-ae]

Several variations of an AE architecture were attempted in preliminary experiments, and ultimately the one represented in Table 4.9 was used. Note that with an autoencoder, the outputs from the encoder are not the $\mu$ and $\sigma$ for a Gaussian distribution from which $z$ is sampled, but the encoded $z$ vector directly.

These two Vision architectures were tested against each other by using each to

| Setting | Value |
|---|---|
| Epochs | 10 |
| Latent dim | $z \in \mathbb{R}^{16}$ |
| Mini-batch size | 100 |
| Optimizer | Adam, lr=0.001 |
| Hidden layers | 2 |
| Hidden dim | $2048 \rightarrow 1024$ |

Table 4.9: [ptc / mdn-lstm / vision-ae] settings.

encode state frames into a latent $z$, and then reconstructing the latent $z$ and measuring how similar it was to the original state frame. As the state consists of integers, the measure of similarity was simply just to see if the reconstructed frame contained the same integer values for certain parts of the state. The parts included the position of the agent on the board, which was thought to be very important during game play, and had the most variety of values (121 – anywhere on an 11 x 11 board).

#### 4.3.1.3 3. Train Model

The Vision based on an AE (autoencoder) performed the best (more details in *Results*, Chapter 5) on the state frame reconstruction tests described earlier. So this Vision was used to train a Model. Furthermore, a version of the Model was trained with no Vision altogether – this was possible as the frames for Pommerman were small enough on their own with dimensionality $\mathbb{R}^{375}$, and did not potentially need to be compressed to a latent $z$. So the Model was modified to optionally take raw frames as input, to predict the next frame.

**Experiment Tag:** [ptc / vision-ae / mdn-lstm / model]

The settings and hyperparameters listed in Table 4.10 were used to train a version of the Model using latent $z$'s encoded using Vision trained with an AE.

**Experiment Tag:** [ptc / no-vision / model-lstm]

A version of the Model was trained using the raw state frames collected in [ptc / mdn-lstm / random_rollouts], instead of latent $z$'s encoded by Vision. Furthermore, the MDN was removed as the output of the network, and it was trained using Mean Squared Error to instead output raw state frames $s_{t+1}$ directly. The reason for this

| Setting | Value |
|---|---|
| Data | AE encoded $z \in \mathbb{R}^{16}$ |
| Epochs | 10 |
| LSTM hidden dim | 512 |
| Gaussian mixtures | 5 |
| TBPTT sequence length | 128 |
| Optimizer | Adam, lr=0.001 |
| Predict *done* | False |

Table 4.10: [ptc / vision-ae / mdn-lstm / model] settings.

choice was that this version of Model was used to solely test the predictive power and accuracy of a model learned in a LSTM network versus a DNC network, without adding in the probabilistic benefits that would be useful to train a Controller. As time was limited, this setup made sense to perform some quantitative tests for the core objectives of this research. More details follow below in Section 4.3.2.

| Setting | Value |
|---|---|
| Data | Raw state frames $s \in \mathbb{R}^{375}$ |
| Epochs | 10 |
| LSTM hidden dim | 512 |
| Gaussian mixtures | 5 |
| TBPTT sequence length | 128 |
| Optimizer | Adam, lr=0.001 |
| Predict *done* | False |

Table 4.11: [ptc / no-vision / model-lstm] settings.

#### 4.3.1.4 4. Train Controller

**Experiment Tag:** [ptc / vision-ae / mdn-lstm / controller]

As detailed in Section 3.1.4.3, the controller was adapted to handle pairs of two individual parameters, treating each as an individual agent part of the team. Using this setup, the Controller pair was evolved using the settings and hyperparameters listed in Table 4.12.

A version of the experiment was also carried out with the same exact setup in

| Setting | Value |
|---|---|
| CMA-ES population size | $\lambda = 64$ |
| CMA-ES fittest genes | $\mu = 1/4$ |
| CMA-ES target fitness | Mean cumulative reward of 15* |
| Trials per mutation | 16 |
| Number of actions | 6 |
| Action dim | $a \in \mathbb{R}^6$ |
| Parameters dim | $W \in \mathbb{R}^{8394}$, $b \in \mathbb{R}^a$, 8400 in total, x2 |
| Action calculation | $tanh(W \cdot [s_t + h_t + c_t])$ |
| Curriculum | initial max timesteps=50, step=25 |
| MDN temperature | $\tau = 1.0$ |
| Environment type | OpenAI Gym |
| Self-play freezing interval** | 10 generations*** |

Table 4.12: [ptc / vision-ae / mdn-lstm / controller] settings. *This value was chosen because the final reward of a team is $+1$ if it wins an individual instance of a game, and for these experiments the total number of wins over the number of trails is returned, and used as the fitness metric for CMA-ES. So if a team wins 15 games out of 16 trails, it has won $\sim 93.75\%$ of games. **See Section 3.1.4.3. ***Multiple values were tried, but ultimately the value provided was chosen.

Table 4.12, except instead of using self-play, the opposing team was set up using the deterministic Pommerman SimpleAgent.

#### 4.3.1.5 5. Test

**Experiment Tag:** [ptc / vision-ae / mdn-lstm / test]

For testing the Controller consisting of a paired team of two agents, the live environment was used, and the team was tested against the deterministic Pommerman SimpleAgent, over 100 total trials. The final test metric was the percentage of games the team won over the total number of trials.

### 4.3.2 MDN-DNC

The random rollouts and trained Vision from the non-DNC set of experiments could be used to train a MDN-DNC in the Model, and then train a Controller. However, full experiments, to create a final MDN-DNC based trained agent to test in the actual

environment, were not conducted due to limited time available.

Since the primary objective of this research was to investigate how well of a predictive model could be learned in a DNC versus traditional LSTM, only one experiment was carried out to train a DNC based model. This was so that tests could be conducted by simply comparing the predicted states between a DNC based model and LSTM based model, to test which one had learned to predict the next state $s_{t+1}$ more accurately, given the current state and action $[s_t + a_t]$.

#### 4.3.2.1 Model

**Experiment Tag:** [ptc / no-vision / model-dnc]

The Model was trained using the settings and hyperparameters defined in Table 4.13.

| Setting | Value |
|---|---|
| Data | Raw state frames $s \in \mathbb{R}^{375}$ |
| Epochs | 10 |
| LSTM hidden dim | 512 |
| Gaussian mixtures | 5 |
| TBPTT sequence length | 128 |
| Optimizer | Adam, lr=0.001 |
| Predict *done* | False |
| DNC memory slots | $N = 256$ |
| DNC memory width | $W = 64$ |
| DNC read head | $R = 4$ |

Table 4.13: [ptc / no-vision / model-dnc] settings.

#### 4.3.2.2 Test

**Experiment Tag:** [ptc / no-vision / dnc-vs-lstm-test]

To test the DNC based model against the LSTM one, a different test was created. Since the idea was to check which architecture learned a better predictive model, future states predicted from the models were compared with the future states observed in the actual environment. A random rollout using random actions would be performed in PommeTeamCompetition-v0 in OpenAI Gym. At every state $s_t$ observed in the environment, and every random action $a_t$ taken at that state, $[s_t + a_t]$ would be input to the

models to receive $s_{t+1}$, and this prediction of the state at the next timestep would then be compared with the actual state at the next timestep observed from the environment.

This test was performed over $1,000$ new random rollouts. Rollouts stored on disk with [ptc / mdn-lstm / random_rollouts] could have been used, but the models had been trained on those, so fresh rollouts were used. The accuracy between predicted $s_{t+1}$ (from both the LSTM and DNC models), and real $s_{t+1}$, included comparing the integer values for equality. The following parts of the state were compared: position coordinates, whether the position coordinates matched the position on the board, amount of ammo, blast strength, can kick (true/false or 1/0), teammate, enemies, bomb blast, bomb life, and alive (which players are alive). The percentage of correct parts of each state were recorded for both the LSTM and DNC models for comparison.

# Chapter 5

# Results and Analysis

In this chapter, the results and observations from all experiments laid out in Chapter 4 are listed and analyzed.

## 5.1 CarRacing-v0

### 5.1.1 MDN-LSTM

The random rollouts collected for [cr0 / mdn-lstm / random_rollouts] took approximately 12 hours to collect, and amounted to a total of 7*GB* of diskspace. Some of the rollouts were randomly selected for visual inspection, and frames turned into animated GIFs. A variety of scenarios were observed with the use of the improved random action sampling method (Algorithm 1) – the car was observed going in all directions.

Training [cr0 / mdn-lstm / vision] on a GPU took approximately ten hours to complete, including encoding each frame into a latent $z$ at the end and saving it to disk, pre-processed for Model. Sample frames were encoded and decoded using the CVAE as it trained and saved to disk for visual inspection of what the network was learning. A clear improvement in the quality of reconstructed frames was observed towards the end of training.

Training [cr0 / mdn-lstm / model] on a GPU also took approximately ten hours to complete with a truncated backprop through time (TBPTT) sequence length of 128. Training was much faster with longer sequence lengths, but for lengths longer than 128, a drastic degradation in predictions was observed, due to exploding gradients (gradient norms were output through training to confirm). Gradient clipping was attempted but training on a sequence length of 128 without clipping performed the best. Performance

was measured by observing the loss values for erratic behavior, and through samples of rollouts recreated using the model under training, and saved to disk as a collage for visual inspection at every epoch.

Training [cr0 / mdn-lstm / controller] was by far the slowest. On a compute cluster with 520 CPU cores, the experiment was halted after completing 400 generations of evolution over five days. In those five days, a solid increase in performance was observed – the mean cumulative rewards received in the environment steadily increased. A graph of the mean cumulative rewards over the generations can be seen in Figure 5.1.

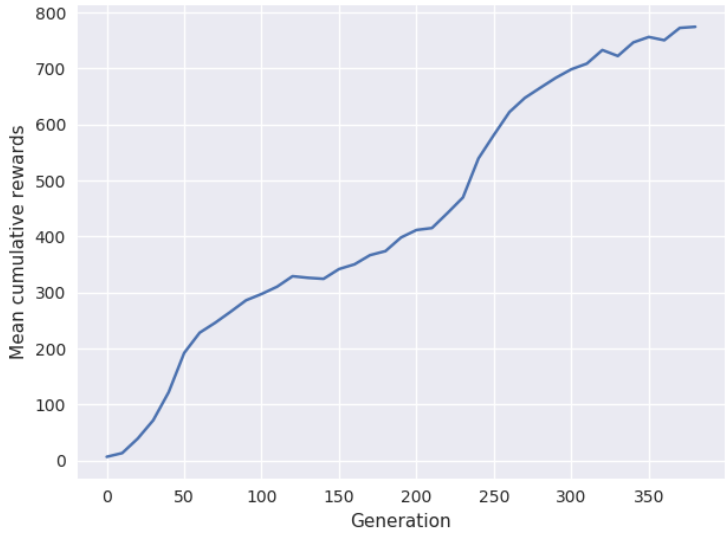



Figure 5.1: [cr0 / mdn-lstm / controller]: performance.

Tests for the trained Controller were performed as described in [cr0 / mdn-lstm / test]. The final cumulative rewards over 100 rollouts are listed in Table 5.1. The agent had clearely learned to perform well on the CarRacing-v0 task.

| **Agent** | **Rewards** |
|---|---|
| Random agent | $-72 \pm 3$ |
| Trained agent | $753 \pm 13$ |

Table 5.1: [cr0 / mdn-lstm / test]: mean and standard deviation of cumulative rewards received by agents over 100 rollouts.

## 5.2 ViZDoom: Take Cover

### 5.2.1 MDN-LSTM

[dtc / mdn-lstm / random_rollouts] took only two hours to collect, likely because the ViZDoom engine runs much faster. The data amounted to $13GB$ of diskspace. Again, some of the rollouts were randomly selected for visual inspection, where frames were saved as animated GIFs.

Training [dtc / mdn-lstm / vision] on a GPU took approximately ten hours to complete, including encoding all frames from random rollouts into latent $z$. Sample frames were encoded and decoded using the CVAE and saved to disk for visual inspection of what the network was learning as training progressed. A clear improvement in the quality of reconstructed frames was observed towards the end of training.

Training [dtc / mdn-lstm / model] on a GPU took approximately ten hours to complete. Performance was measured by observing the loss values, and through visual inspection of samples of rollouts recreated using the model under training. Furthermore, animated GIFs of simulated rollouts with predefined actions were created at the end of training, to observe how the model responded to actions (for example, whether the player moved left with a sequence of repeated $[1,0]$ actions), and to observe if fireballs were forming and following a plausible trajectory[1]. At times, the model was observed to get "stuck", where a $[z_t + a_t]$ input produced a $z_{t+1}$ which was almost exactly identical to $z_t$, and the simulation would fall into an endless loop. Adjusting the temperature helped with the issue but not entirely. There were also other flaws in the model observed at times through these animated GIFs, where fireballs would appear in the middle of the screen out of nowhere. The additional "done" flag output by the MDN-LSTM for this experiment was also tested. For each simulated rollout, the final frame where $done >= 0.5$ was recorded, and in the vast majority of frames observed, a fireball was spotted in the middle of the screen, meaning the model had learned that the agent dies when it is hit by a fireball.

[dtc / mdn-lstm / controller] was trained on a compute cluster with 520 CPU cores. Initially the Controller was trained without adding Gaussian noise to the initial frame. This sometimes resulted in simulated rollouts where the agent was stuck as described earlier – no fireballs would form and the agent would be in an endless loop. This also hindered CMA-ES from learning, because for simulated training rewards were based

[1] An example of a simulated rollout in ViZDoom: Take Cover, with a predefined sequence of actions: https://github.com/AdeelMufti/WorldModels/blob/master/asset/DoomTakeCover_dream_rollout.gif

on how long the agent was alive, and being in an endless loop with no fireballs, the 0.5 threshold for "done" never being reached meant that the reward was very high. Adding noise to the initial frame mitigated the issue. Training was stopped after approximately two days and 200 generations of evolution.

The [dtc / mdn-lstm / test] experiment was used to check how well the Controller performed, which was especially important because it was trained in the simulated environment. The final cumulative rewards over 100 rollouts are listed in Table 5.2[2].

| Agent | Rewards |
|---|---|
| Random agent | $223 \pm 105$ |
| Trained agent | $680 \pm 411$ |

Table 5.2: [dtc / mdn-lstm / test]: mean and standard deviation of cumulative rewards received by agents over 100 rollouts.

### 5.2.2 MDN-DNC

Training [dtc / mdn-dnc / model] was extremely slow. Without the optimizations mentioned in Section 3.2, a single epoch would take 52 days. With the optimizations, a single epoch took one-and-a-half days. As time was running out for this research, though training of the MDN-DNC was planned for a full 20 epochs as was done for the MDN-LSTM counterpart, just a single epoch of training was used. However, with just a single epoch, the MDN-DNC appeared to be performing remarkably well with some preliminary testing. A series of animated GIFs were created in simulations, by running them through pre-defined sequences of actions (notably, left for 100 timesteps, then right for 100 timesteps, repeatedly). The movements were smooth and responsive to the actions, and fireballs were forming and following correct trajectories. There were fewer signs of the simulation getting "stuck." The MDN-DNC trained for 1 epoch appeared to be more robust than what was observed for the MDN-LSTM trained for 20 epochs on the same data. Regardless, quantitative improvements could only be gathered through training and testing the Controller. [dtc / mdn-dnc / controller] was trained for approximately two-and-a-half days for 250 generations.

As described in *Experiments* (Chapter 4) the [dtc / mdn-dnc / test] experiment differed from previous tests of the Controller in that a 100 rollout test in the real en-

[2] A recording of the trained Controller in the real environment can be seen at: https://github.com/AdeelMufti/WorldModels/blob/master/asset/DoomTakeCover_rollout.gif

vironment was run from a snapshot at every fifth generation of evolution. This was done to be certain how the MDN-DNC performed in the real environment (since it was being trained in simulations) as it evolved, but was very compute heavy and hence a lot more costly to perform.

In order to compare the performance of the MDN-DNC based Model and Controller against the MDN-LSTM Model and Controller, a fair assessment would be to use a MDN-LSTM also trained for just one epoch rather that 20 epochs. Thus, a snapshot at one epoch of the MDN-LSTM from the [dtc / mdn-lstm / model] experiment was used as the Model, and another MDN-LSTM based Controller was trained in parallel for 250 generations as well. This Controller was tested similarly, with a 100 rollout test in the real environment run from a snapshot at every fifth generation of evolution.

The MDN-DNC Controller outperformed the MDN-LSTM Controller by a margin. A graph comparing the mean cumulative rewards for each of the two Controllers, over 100 rollouts, at every fifth generation, can be seen in Figure 5.2.

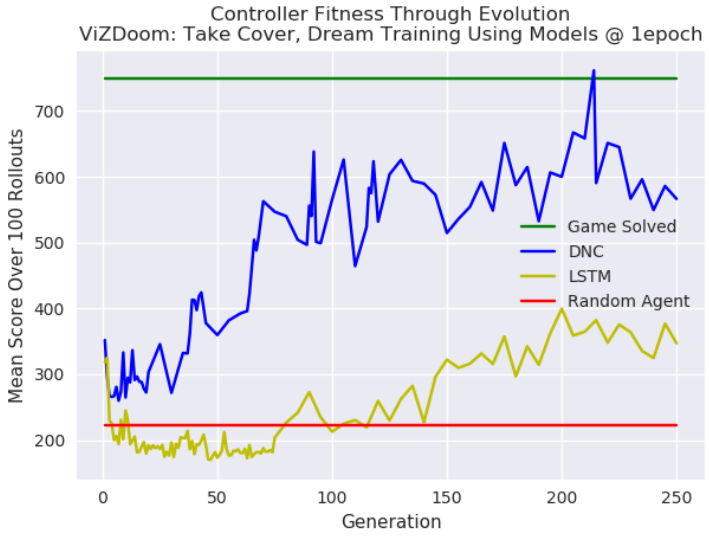



Figure 5.2: MDN-DNC Controller versus MDN-LSTM Controller.

The Controller based on MDN-DNC (trained at one epoch) even outperformed the fully trained MDN-LSTM (20 epochs) Controller from the [dtc / mdn-lstm / controller] experiment – the MDN-LSTM achieved a mean cumulative reward of $680 \pm 411$ over 100 rollouts, whereas the MDN-DNC achieved a mean cumulative reward of $761 \pm 452$ over 100 rollouts around generation 210.

Another interesting observation about the MDN-LSTM trained for one epoch was that the simulated rollouts generated as animated GIFs for visual inspection were jittery, and not as responsive to actions as the MDN-DNC was observed to be. It was clear that the model of the environment it had learned was not as robust. During Controller training, animated GIFs of the Controller performing in a simulation were created, to observe what the Controller had learned. Surprisingly, the Controller had learned to exploit weaknesses in the MDN-LSTM model. It would perform a certain sequence of actions that would lead the Model to getting "stuck" in a loop. And since training in the simulation is done based on the number of timesteps the agent is alive ($done < 0.5$), CMA-ES would see the genes that led to this policy as the fittest, as they would survive the longest due to the endless loop. A similar kind of adversarial policy was also observed by Ha and Schmidhuber (2018). Interestingly though, this did still carry into the real environment, because the MDN-LSTM did better than a completely random agent. A guess is that as the adversarial policy involves a certain sequence of actions, they luckily allowed the agent to live longer than a random one, though it failed to generalize beyond a certain point (a mean cumulative reward of 400).

Figure 5.3: Controller discovers an adversarial policy by exploiting weaknesses in a MDN-LSTM Model trained for one epoch. After a certain sequence of actions in the simulation, a rogue fireball appears on the screen, and every $[z_t + a_t]$ produces a $z_{t+1}$ that is almost identical to the original $z_t$ (with the same fireball stuck in place), causing the simulation to fall in an endless loop.

## 5.3 PommeTeamCompetition-v0

### 5.3.1 MDN-LSTM

[ptc / mdn-lstm / random_rollouts] were the fastest to collect, likely because the state

is not rendered as pixels. $10,000$ rollouts were collected in one hour on a Lenovo Thinkpad W520, amounting to $1.6GB$ of diskspace.

[ptc / mdn-lstm / vision-vae] and [ptc / mdn-lstm / vision-ae], Vision with VAE and Vision with AE as specified in Section 4.3.1, were both very fast to train, completing under an hour on a GPU. The VAE saved pre-computed $\mu$'s and $\sigma$'s for each state frame to disk, whereas the AE saved the latent $z$'s. To test the performance of the two networks, new state frames were collected over 100 rollouts and encoded and subsequently decoded ("reconstructed"), and checked if certain parts of the reconstructed state were equal to the original state. The results are presented in Table 5.3.

| **State** | **VAE** | **AE** |
|---|---|---|
| Position | 47% | 62% |
| Ammo | 52% | 64% |
| Teammate | 82% | 75% |

Table 5.3: [ptc / mdn-lstm / vision-vae] versus [ptc / mdn-lstm / vision-ae]: percentage of reconstructed frames matching the original frame for certain parts of the state.

As the Vision with AE showed the best overall reconstruction accuracy, it was used moving forward. [ptc / vision-ae / mdn-lstm / model] was trained and subsequently used in training [ptc / vision-ae / mdn-lstm / controller]. As self-play was being used to train the team, a pattern was observed in the mean cumulative rewards over the generations where the rewards would increase for ten generations, but drop down again when the "frozen" version of the team being used as the opponent was updated with the latest. At 50 generations of training the Controller, [ptc / vision-ae / mdn-lstm / test] was performed to check how well it generalized by having it play against SimpleAgent over 100 trials. The team performed poorly – no better than a random team. As time was limited, the Controller training was halted altogether and a version was created to play against a team of SimpleAgents rather than using self-play. After 325 generations of training against SimpleAgent, the team was tested over 100 rollouts and was observed to win 18% of games, whereas a randomly initialized team would win only 12% of games. The Controller experiments for PommeTeamCompetition-v0 were halted at that point as the time available to conduct this research had run out.

The primary objective of this research was to investigate how well of a predictive model could be learned in a DNC versus traditional LSTM. Thus, a final experiment, [ptc / no-vision / model-lstm], was carried out to train a MDN-LSTM model in the

PommeTeamCompetition-v0 environment.

### 5.3.2 MDN-DNC

[ptc / no-vision / model-dnc] (DNC based model) was trained to test against [ptc / no-vision / model-lstm] (LSTM based model) using the test methods described for [ptc / no-vision / dnc-vs-lstm-test] (Section 4.3.2.2). The results are graphed in Figure 5.4.

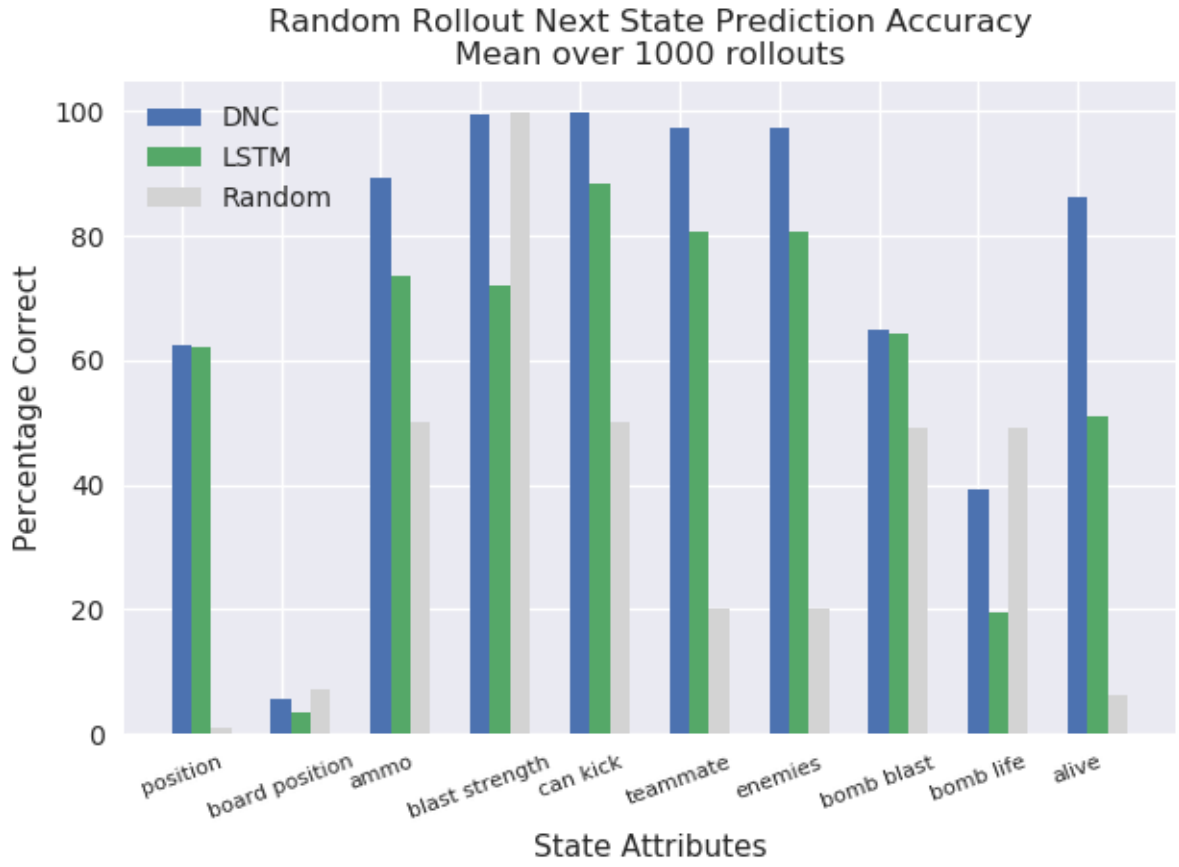



Figure 5.4: [ptc / no-vision / dnc-vs-lstm-test]: testing the predictive power of MDN-DNC over MDN-LSTM. The *Random* percentages are simply random uniform guesses at the possible integer values in the individual components of the state, that were guessed correctly.

It was observed that a MDN-DNC outperformed a MDN-LSTM on learning to predict the PommeTeamCompetition-v0 environment. Given a state $s_t$ and an action $a_t$, the MDN-DNC got better accuracy in predicting $s_{t+1}$ when compared with the actual $s_{t+1}$ observed from the environment.

## 5.4 Analysis

On the baseline MDN-LSTM tasks for CarRacing-v0 and ViZDoom: Take Cover, the goal was to replicate the results of Ha and Schmidhuber (2018), though the final rewards achieved were not as high. This is thought to be because Controller training had not yet completed. For example, the mean rewards for CarRacing-v0 were still increasing (Figure 5.1) when training was halted due to time constraints, to make way for other experiments on the same resources. Perhaps with more training of the Controller, higher rewards might be achieved. For ViZDoom: Take Cover, the temperature hyperparameter for MDN-LSTM was perhaps not ideal, and could be adjusted to achieve better rewards (along with more generations of training the Controller). The guess for adjusting the temperature is made because when sampling from the MDN, the coefficients of the Gaussian mixture models can be dramatically affected as the temperature is varied. Furthermore, the MDN-LSTM did not learn a robust model of the environment and would often get stuck in an endless loop, and it was observed that higher temperature values would mitigate the loop issue.

As seen through these experiments, the DNC outperforms LSTM when learning a predictive model of the environment. This does not come as a surprise, as the DNC is intended to be a Neural Turing Machine, which can learn algorithms on highly complex tasks according to Graves et al. (2016). But aside from the external memory and a complex internal setup to mimic a Turing machine that can learn, the DNC did not necessarily have any other unfair advantage over LSTM in these experiments – as far as additional capacity when it comes to neural network architectures (number of hidden layers, number of hidden units, and so on). The DNC's controller consists of LSTM, which can be set to contain a certain number of hidden units. These hidden units were kept the same for all experiments between the MDN-DNC and MDN-LSTM. Essentially both variations had the same number of layers and units whether LSTM or linear. Thus, the external memory and algorithm learning capability of the DNC truly set it apart.

PommeTeamCompetition-v0 presented a more challenging environment and task over CarRacing-v0 and ViZDoom: Take Cover, for reasons such as sparse rewards, and other reasons detailed in Section 2.5.3. So it was no surprise that the Controller had difficulty in learning the task. This perhaps had to do with how the Controller was set up for this task – how two independent sets of parameters were paired and evolved separately in the same environment, or how the action calculation methodology was

set up. The controller was simple – a mere $8,400$ parameters whereas the CVAE has $4,348,547$. Perhaps a simple, small, linear model is not enough to learn more complex tasks, and more powerful Controller architectures are necessary.

# Chapter 6

# Conclusions and Further Work

## 6.1 Conclusions

For this thesis, research was carried out to test the DNC as a predictive, probabilistic model used for RL. The DNC was used as the recurrent layer (the RNN) in the MDN-RNN used for learning a model of an environment. Tests were carried out in a new architecture, the *MDN-DNC*. Simulations of the ViZDoom: Take Cover environment were created as animated GIFs to observe how it responded to pre-defined sequences of actions. The MDN-DNC appeared to have learned a better model of the environment when compared to its MDN-LSTM counterpart, and was less prone to getting stuck in loops or having other glitches such as fireballs appearing randomly out of sequence. But the ultimate test was to train a Controller with RL, using the learned model, and compare the final cumulative rewards it received in the real environment. A MDN-DNC based model trained for only 1 epoch was used to simulation-train a Controller, which was able to outperform a Controller using a MDN-LSTM model (also trained for one epoch), when tested in the real environment. The MDN-DNC was also tested against MDN-LSTM for learning the PommeTeamCompetition-v0 environment, where the MDN-DNC was able to predict future states more accurately than its counterpart.

Thus it can be concluded that the MDN-DNC is able to learn a more robust model of an environment, at least for the environments tested for this research. MDN-DNC based models could be used to run robust simulations for RL, when using the real environment may be too costly, dangerous, or otherwise expensive to use repeatedly. And so it follows that the DNC is perhaps beneficial in Deep Reinforcement Learning, particularly when model-based methods are used, and could potentially serve as a useful

tool in the ladder towards Artificial General Intelligence (AGI).

## 6.2 Further Work

The MDN-DNC was able to outperform a MDN-LSTM in the ViZDoom: Take Cover task when training a full agent (including the Controller). However, the performance of a fully trained agent with a MDN-DNC was not tested in PommeTeamCompetition-v0, due to time constraints and due to the fact that the MDN-LSTM "baseline" agent performed underwhelmingly. This was likely due to reasons presented in the results analysis (Section 5.4) – mainly, the Controller not having been adapted ideally to the Pommerman task as it is so different than the others due to having to perform multi-agent learning. While the MDN-DNC versus MDN-LSTM tests on PommeTeamCompetition-v0 showed superior predictive power of the MDN-DNC, a complete test would include a fully trained RL agent that makes use of the MDN-DNC. Thus, the MDN-DNC remains to be tested with a trained Controller in more complex environments other than ViZDoom: Take Cover.

The action calculation, when there are numerous actions, presents a challenge. When using the approach to split a single scalar, produced by taking the dot product of the input with the Controller's parameters, into a range (Section 3.3.2), leads the Controller to be entirely biased towards two actions, which hinders training. Perhaps some other approaches for calculating the actions could be explored, such as using a different activation function rather than tanh, normalizing the inputs to the Controller, adjusting the initial seed of the Controller's parameters, and so on.

Another area that could be explored when using a MDN-DNC to train the Controller, is to add parts of the DNC's memory to the Controller's inputs. The Controller typically receives as input the compressed latent $z_t$ from the environment or the simulation, the hidden state of the LSTM $h_t$, and the cell state of the LSTM $c_t$. Perhaps the Controller could also benefit by additionally receiving the DNC's memory as input. The DNC contains a number of *read heads*, which return parts of the memory read to the DNC's own controller (LSTM layer) at every timestep – this memory is cruical to the DNC's performance. For these experiments, the number of read heads were four, with a memory width of 64, which means that the DNC's LSTM layer receives a $4x64 = 256$ dim input at every timestep. This is a reasonably small amount of additional input that could be presented to the Controller.

The World Models approach is limited in that the Vision and Model can only go

as far as to perform well on scenarios collected by the random rollouts, which are performed using a random policy. It is not very plausible that the random policy will be able to get to all scenarios or parts of the environment (such as all levels of a long game). Thus, the World Models approach could be extended to work in an iterative manner. During the first iteration, a random policy could be used to collect the rollouts, to train Vision, Model, and Controller. The Controller may potentially only be able to get up to a certain point in the environment (such as past just the first level in a game). Afterwards, with all future iterations, new rollouts could be collected using the policy learned by the Controller combined with randomness, with the idea that new scenarios could be encountered in the environment and logged to disk. The Vision, Model, and Controller could be trained again with the new set of "random" rollouts and potentially perform well further into the environment.

## 6.3 Contributions

There were a few notable contributions made through this work.

At the time the research was planned and carried out, a search across Google, Google Scholar, and research publications archives such as arXiv, did not yield any existing work on using a DNC to learn a predictive model of an environment in the RL context. Thus, this research makes the first attempt to do so, and finds positive results in favor of the DNC.

The distributed evolution of CMA-ES (Figure 3.5) that scales dynamically to the size of a compute cluster is another contribution this research offers. The algorithm and methodology to distribute the genes worked well in practice.

The elementary form of curriculum learning applied to the Controller is also a new contribution. This sped up training of the Controller in wall-clock time, though it is not obvious if it helped with the performance on the task itself.

This work offers a complete implementation of the World Models framework[1]. When the research was planned and started, no complete implementation was available publicly. Since then, Ha and Schmidhuber (2018) released their code in Tensorflow. So the efforts from this research offer an alternative complete implementation in Chainer.

And lastly, another contribution is a highly optimized implementation of the Differentiable Neural Computer in Chainer[2].

[1] https://github.com/AdeelMufti/WorldModels

[2] https://github.com/AdeelMufti/DifferentiableNeuralComputer

# Appendix A

# Appendix A

## A.1 MDN Toy Task

Consider the following inverted sinusoidal function, where $\mathcal{U}$ is the real number uniform random distribution:

$$y = \mathcal{U}[0,1), \;\; y \in \mathbb{R}^{500} \tag{A.1}$$

$$x = sin(2\pi y) \; + \; 0.2\mathcal{U}[0,1) \; * \; cos((2\pi y) + 2) \tag{A.2}$$

This function presents a one-to-many mapping – a single input x value mapping to multiple output y values – and can be used for checking a Mixture Density Network (MDN) implementation. If graphed, the functions looks as shown in Figure A.1.

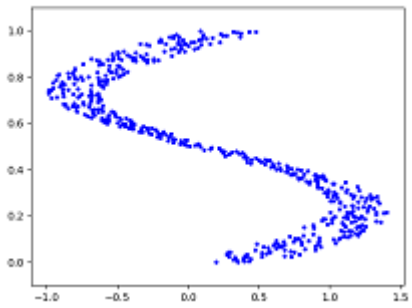



Figure A.1: The graph of inverted sinusoidal function in Equations A.1, A.2.

If a simple feedforward neural network is trained using Mean Squared Error (MSE) to model $y = Wx + b$, the network learns to predict the mean of the $y$ values given an $x$ value, as seen in Figure A.2.

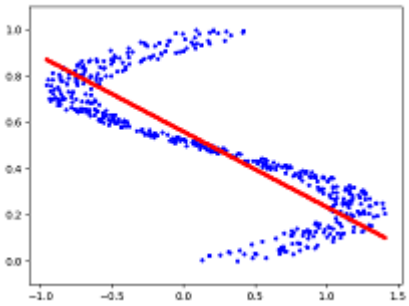


Figure A.2: Graph of inverted sinusoidal function (blue) overlaid with predictions (red) from a neural network trained with MSE.

However, if the feedforward network instead outputs a Gaussian Mixture Model (GMM), it can be sampled from to predict a $y$ value given an $x$ value. This network can learn to predict multiple $y$ values given a single $x$ value accurately, as seen in Figure A.3.

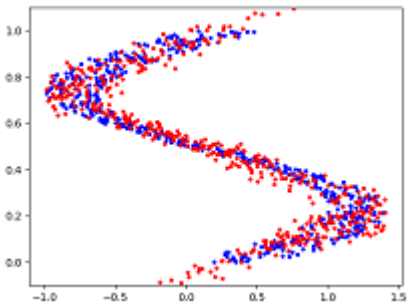


Figure A.3: Graph of inverted sinusoidal function (blue) overlaid with predictions (red) sampled from a neural network trained to output a GMM.

## A.2 CMA-ES Toy Task

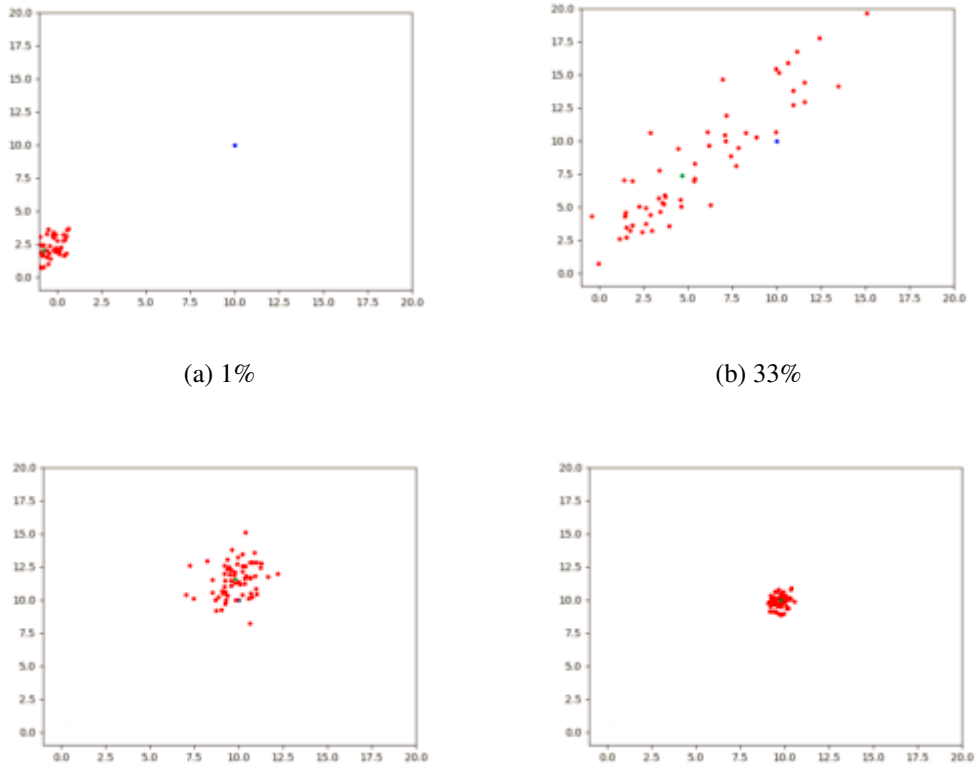
Consider a shifted Schaffer function with global minimum ("solution") at point $(10,10)$, as seen in Equation A.3.

$$f(x,y) = 0.5 + \frac{sin^2[(x-10)^2 - (y-10)^2] - 0.5}{[1+0.001[(x-10)^2 + (y-10)^2]]^2}, \quad f(10,10) = 0 \tag{A.3}$$

The CMA-ES task can be modeled to simply learn these solution points, and hence in this case evolve solution $s \in \mathbb{R}^2$. The fitness function $F()$ for CMA-ES can be treated as the negative square error between the solution being tested, and the actual solution, against the Schaffer function, as follows:

$$F(s_1, s_2) = -(f(s_1, s_2) - f(10, 10))^2$$

Therefore, the task for CMA-ES is to find the solution $(s_1 = 10, s_2 = 10)$. Given the right population size $\lambda$ and the right $\mu$ for CMA-ES, it eventually converges to a solution. With $\lambda = 64$ and $\mu = 0.25$, a visualization of CMA-ES as it evolves a population over generations can be seen in Figure A.4.



(a) 1%

(b) 33%

(c) 66%

(d) 99%

Figure A.4: CMA-ES progression on the shifted Schaffer function toy task with solution at $f(10, 10)$. Blue dot represents the solution. Red dots the entire population being tested. The green dot the mean of the population (i.e. the solution $s$ in the current generation).

## A.3 DNC Toy Tasks

To test the Differentiable Neural Computer, a few toy tasks can be used, including:

1. **Repeat:** Simply echo a row-wise sequence of randomly generated 0's and 1's.
2. **Addition:** Sum a randomly generated row-wise sequence, with a 1 in each row representing a number based on the column (position) it is contained in. Example below.
3. **Priority sort:** Harder task. Essentially involves repetition as well. A row-wise sequence of randomly generated 0's and 1's are to be sorted according to a priority assigned to each row, with the priority sampled uniformly from [-1,1]. This task was used in the Neural Turing Machines (DNC's predecessor) paper (Graves et al., 2014).

While the authors of the DNC tested it on much more complex problems (Graves et al., 2016), these tasks are easy to implement and useful for quickly testing the DNC implementation.

```
[[0. 0. 1. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 1.]]
```

(a) Input (2+0+2+0+4+0+4)

```
[[ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [ 0.]
 [12.]]
```

(b) Target

```
[[-0.14442]
 [-0.30143]
 [-0.24193]
 [-0.16146]
 [0.08639]
 [0.01805]
 [0.03887]
 [12.01316]]
```

(c) Output

Figure A.5: DNC *addition* toy task input/output example.

An additional test for the DNC is to check its generalizing ability, to handle longer sequences than it was trained on. This can be tried on any task, and compared against a traditional LSTM similarly trained on the same task with the same data and sequence lengths. This generalization test was done using the addition task with two networks (LSTM, and DNC with a 256$x$64 memory matrix – both with 256 hidden units). Both were trained with the same data and sequence lengths, but tested after training on increasingly longer sequence lengths. The results can be found in Figure A.6.

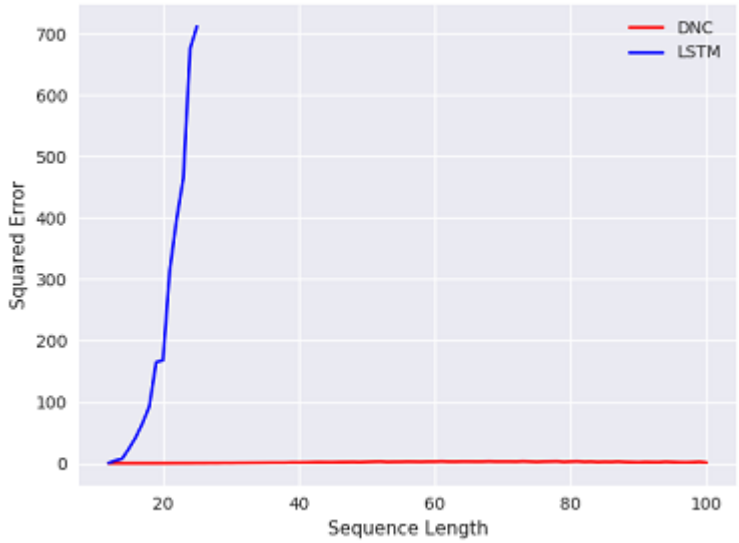



Figure A.6: Error over increasing sequence lengths on the addition task with traditional LSTM versus DNC trained for 50k iterations. Both were trained using a random sequence length between two and 12, and had never seen longer sequence lengths during training. The traditional LSTM network quickly diverge on sequences greater than 12, while the DNC network is very robust to generalizing on longer sequences.

# Bibliography


Bishop, C. M. (1994). Mixture density networks. Technical report, Citeseer.

Goertzel, B. and Pennachin, C. (2007). *Artificial general intelligence*, volume 2. Springer.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Graves, A., Wayne, G., and Danihelka, I. (2014). Neural turing machines. *CoRR*, abs/1410.5401.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., et al. (2016). Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476.

Ha, D. and Schmidhuber, J. (2018). World models. *CoRR*, abs/1803.10122.

Hansen, N. (2016). The CMA evolution strategy: A tutorial. *CoRR*, abs/1604.00772.

Ivanikovas, S., Dzemyda, G., and Medvedev, V. (2009). Influence of the neuron activation function on the multidimensional data visualization quality. In *ASMDA. Proceedings of the International Conference Applied Stochastic Models and Data Analysis*, volume 13, page 299. Vilnius Gediminas Technical University, Department of Construction Economics & Property.

Levine, S. and Koltun, V. (2013). Guided policy search. In *International Conference on Machine Learning*, pages 1–9.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Rae, J. W., Hunt, J. J., Harley, T., Danihelka, I., Senior, A. W., Wayne, G., Graves, A., and Lillicrap, T. P. (2016). Scaling memory-augmented neural networks with sparse reads and writes. *CoRR*, abs/1610.09027.

Salimans, T., Ho, J., Chen, X., Sidor, S., and Sutskever, I. (2017). Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*.

Silver, D. (2016). Deep reinforcement learning, a tutorial at ICML 2016. `https://icml.cc/2016/tutorials/deep_rl_tutorial.pdf/`.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement learning: An introduction*. MIT press Cambridge.

Sutton, R. S. and Barto, A. G. (2017). *Reinforcement learning: An introduction*. Unpublished, second edition.

Turing, A. M. (1937). On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265.

van der Walt, S., Colbert, S. C., and Varoquaux, G. (2011). The numpy array: a structure for efficient numerical computation. *CoRR*, abs/1102.1523.