

UNIVERSITY OF CAPE TOWN



HALF DISSERTATION PRESENTED FOR THE DEGREE OF MASTER OF
SCIENCE

DEPARTMENT OF STATISTICAL SCIENCES

**Evolutionary algorithms for optimising
reinforcement learning policy
approximation**

Author

Blake CUNINGHAM

Supervisor

Prof. Bruce BASSETT

August 4, 2019

The copyright of this thesis vests in the author. No quotation from it or information derived from it is to be published without full acknowledgement of the source. The thesis is to be used for private study or non-commercial research purposes only.

Published by the University of Cape Town (UCT) in terms of the non-exclusive license granted to UCT by the author.

Declaration of authorship

I, Blake CUNNINGHAM, declare that this thesis titled, “Evolutionary algorithms for optimising reinforcement learning policy approximation” and the work presented in it are my own. I confirm that:

1. I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own.
2. I have used the Harvard convention for citation and referencing. Each contribution to, and quotation in, this dissertation from the work(s) of other people has been attributed, and has been cited and referenced. Any section taken from an internet source has been referenced to that source.
3. This dissertation is my own work, and is in my own words (except where I have attributed it to others).
4. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.

Signed by candidate

Signed: _____

Date: 4 August 2019

Acknowledgements

I would like to thank the following people for their invaluable assistance in producing this dissertation:

- Prof. Bruce Bassett, my supervisor, for his help guiding me toward a novel area of research, providing critical feedback throughout the research process, and including me in the weekly meetings at AIMS where I was able to discuss machine learning topics with some of the brightest minds in the field.
- Emmanuel Dufourq for his help guiding me within the field of evolutionary algorithms, and making his EDEN code available to incorporate into my own work.
- The UCT Statistics Department, and my excellent lecturers on machine learning topics for how well they prepared me to embark on this research: Dr. Sebnem Er; Dr. Ian Durbach; Dr. Miguel Lacerda.
- The many incredibly helpful guides, and code (particularly that made available by OpenAI), that allowed me to gain an understanding on advanced topics within the field of reinforcement learning, and quickly achieve state-of-the-art results in my own computing environment.
- My partner, Caitlin Smit, for allowing me to talk through my various plans and approaches throughout the research, and her understanding during some of the early mornings and weekends when this work was being conducted in addition to my full time job.

Thank you to all.

MSc Data Science: Dissertation

Evolutionary algorithms for optimal reinforcement learning policy approximation

Blake Cunningham CNNBLA001

August 4, 2019

Abstract

Reinforcement learning methods have become more efficient in recent years. In particular, the A3C (asynchronous advantage actor critic) approach demonstrated in [Mnih et al. \(2016\)](#) was able to halve the training time of the existing state-of-the-art approaches. However, these methods still require relatively large amounts of training resources due to the fundamental exploratory nature of reinforcement learning. Other machine learning approaches are able to improve the ability to train reinforcement learning agents by better processing input information to help map states to actions - convolutional and recurrent neural networks are helpful when input data is in image form that does not satisfy the Markov property. The specific required architecture of these convolutional and recurrent neural network models is not obvious given infinite possible permutations. There is very limited research giving clear guidance on neural network structure in a RL (reinforcement learning) context, and grid search-like approaches require too many resources and do not always find good optima. In order to address these, and other, challenges associated with traditional parameter optimization methods, an evolutionary approach similar to that taken by [Dufourq and Bassett \(2017\)](#) for image classification tasks was used to find the optimal model architecture when training an agent that learns to play Atari Pong.

The approach found models that were able to train reinforcement learning agents faster, and with fewer parameters than that found by OpenAI's model in [Blackwell et al. \(2018\)](#) - a superhuman level of performance.

Contents

Declaration	I
Acknowledgements	II
1 Introduction	1
2 Review of the literature	2
2.1 Introduction	2
2.2 Reinforcement learning	3
2.2.1 Introduction	3
2.2.2 Overview of RL	3
2.2.3 Policy gradient approaches	8
2.2.4 Actor-critic approaches	9
2.2.5 Asynchronous advantage actor-critic (A3C)	11
2.2.6 Note on Markov processes	15
2.2.7 Note on exploration vs. exploitation	16
2.2.8 Reward functions	17
2.2.9 Conclusions	19
2.3 Neural network design	20
2.3.1 Introduction	20
2.3.2 Neural networks	20
2.3.3 Loss functions	22
2.3.4 Backpropagation	23
2.3.5 Learning rates	24
2.3.6 Regularization	26
2.3.7 Gradient descent optimization	29
2.3.8 Other notable techniques	32
2.3.9 Activation functions	34
2.3.10 Convolutional layers	37
2.3.11 Recurrent neural networks	42
2.3.12 Conclusion	45
2.4 Parameter searching methods	46
2.4.1 Introduction	46
2.4.2 Grid search	47
2.4.3 Random search	47
2.4.4 Other methods	49
2.4.5 Auto ML	49
2.4.6 Conclusions	50
2.5 Genetic algorithms	50
2.5.1 Introduction	50
2.5.2 Overview	51
2.5.3 Fitness and selection	51
2.5.4 Genetic operations	53
2.5.5 Termination	54
2.5.6 EDEN	55
2.5.7 Use in reinforcement learning	57
2.5.8 Conclusions	58

3 Methodology	58
3.1 Introduction	58
3.2 High level overview	59
3.3 Task	60
3.4 Tournament score	62
3.5 Summary of experiment parameters	63
3.6 Conclusions	66
4 Results and discussion	66
4.1 Introduction	66
4.2 Reinforcement learning	68
4.3 Evolution	69
4.4 Neural network design and parameter selection	76
4.4.1 Summary parameters	76
4.4.2 Layers	78
4.4.3 Learning rate	80
4.4.4 Activation functions	81
4.4.5 Convolutional 2D filters	82
4.4.6 Model examples	84
4.5 Conclusions	86
5 Conclusion	86
Appendices	95
A Initial progress with vanilla policy gradient	96
B List of models	100
B.1 Generation 0	100
B.2 Generation 1	104
B.3 Generation 2	108
B.4 Generation 3	112
B.5 Generation 4	117
B.6 Generation 5	121
B.7 Generation 6	125
B.8 Generation 7	129
B.9 Generation 8	134
B.10 Generation 9	138
B.11 Generation 10	142

1 Introduction

The field of reinforcement learning (RL) has seen significant advancement in recent years, with well-known breakthroughs such as being able to achieve superhuman performance in Atari games ([Mnih et al., 2015](#)), and being able to beat the world's best chess-playing software through self-play only ([Gibbs, 2017](#)). Most recently, an RL agent developed by Google DeepMind was used to defeat the top human players in the highly complex StarCraft II computer game ([Grubb, 2019](#)).

The literature is evolving quickly, and many new results are changing what is thought of as best practice for optimal learning. In particular, deep neural networks have been successful in helping to train the state-action functions in RL - these are the functions that evaluate the current environment and then try to decide what action to take in order to maximise the chance of getting some future reward.

There are many components to this including, *inter alia*, how best to learn this reward (Q-learning vs. Policy Gradient vs. Actor-Critic approaches), how to model the reward (does one use "shaping" to augment the reward function, and how best to shape the reward function), how to initialise the parameters on the learning function (e.g. a neural network), how to best handle the exploration-exploitation trade-off, and what specific form a state-action or state-value function should take (given that one has already chosen to use a Policy Gradient approach, for example).

This dissertation aims to research the last issue mentioned - what might be the optimal function for approximating the optimal state-action policy. Whereas RL can be thought of as the third branch of machine learning (in addition to supervised learning and unsupervised learning), the approximation of this policy function is, in practice, a supervised learning task where we must choose a technique and structure to achieve the best test results (must be defined e.g. quicker learning; more consistent results etc.). In [Mnih et al. \(2015\)](#) a relatively simple convolutional neural network was used (two convolutional layers and two fully connected layers) that was decided on after initial tests with a handful of the available (and eventually tested) Atari games.

Finding good architectures and hyper-parameters for neural network models used in RL becomes more difficult depending on the size and complexity of the search space. After more than just a few combinations of discrete and continuous possible parameters, traditional methods like grid search and random search become computationally infeasible and do not learn how to narrow in on more promising parts of the search domain. This thesis therefore takes an evolutionary approach similar

to that used by [Dufourq and Bassett \(2017\)](#). The research aims to find good network structures that could be used to inform design decisions in future RL tasks. The dissertation will consider factors such as ability to converge, efficiency of training, and robustness within the learning environment - the classic Atari game of Pong.

The dissertation will begin by introducing the concepts and relevant literature on the main topics required for understanding the methodology undertaken in the experiment, with notes on previous research justifying some of the specific design choices made here (Section 2). Once an understanding of the key concepts is developed, the dissertation gives an overview of the methodology and details the specific design choices made (Section 3). The results are then presented and discussed showing how the dissertation's approach was used successfully, with commentary on specifically successful architectures and parameters (Section 4). Finally, the conclusion comments on the limitations of the research and possibilities of future related research (Section 5).

2 Review of the literature

2.1 Introduction

The literature review aims to achieve two objectives. First, it will give an overview of the relevant theory of each of the elements related to the dissertation, explaining each so that the reader is able to easily follow the methodology section (Section 3). Second, the literature review will review key aspects of recent literature relevant to the dissertation experiment, providing insight on some design choices based off of previous work. The sub-goal of this second objective is to show the rationale behind choosing an evolutionary algorithm for hyper-parameter search, and why this is relevant and novel in the literature.

The chapter will start with the overall concept of reinforcement learning (Section 2.2), then introduce neural networks (Section 2.3) in order to understand how policy models are approximated. Next, the problem of finding optimal architectures and hyper-parameters is introduced (Section 2.4) with an overview of traditional techniques. As an improvement over traditional techniques, evolutionary algorithms are explained (Section 2.5) with particular emphasis on previous work by [Dufourq and Bassett \(2017\)](#).

2.2 Reinforcement learning

2.2.1 Introduction

RL (Reinforcement Learning) is a branch of machine learning that according to [Sutton and Barto \(2018\)](#) is “much more focused on goal-directed learning from interaction than are other approaches to machine learning”. Supervised learning focuses on predicting the value or category of an outcome based on a specified range of inputs ([Hastie et al., 2009](#)) - RL can make use of this technique while exploring an environment to create associated rewards with pairs of states and actions.

This section aims to give a very brief overview of RL for the purposes of understanding the experiment in the dissertation - several key concepts in the field are therefore omitted if they do not contribute directly to ultimately understanding A3C (Asynchronous Advantage Actor Critic) (Section [2.2.5](#)) where the state is fully observed, but the process does not have the Markov property of memorylessness (Section [2.2.6](#)).

Section [2.2.2](#) gives an overview of RL, introducing the key concepts as a base for understanding RL in general. Subsequent sections describe nuanced aspects of RL that pertain to this dissertation, leading to a description of A3C and how it was applied here.

2.2.2 Overview of RL

As mentioned above, RL can make use of supervised learning (Section [2.2.8](#)). More specifically, for some observed state \mathbf{o}_t , supervised learning could help predict which action, \mathbf{a}_t , was taken - assuming the supervised model was trained with pairs of states and actions. The model that gives the probabilities for which action is likely to result in the smallest loss (according to some loss function of the supervised learning method) is given by $\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$, which is the policy of which action to take based on a particular observation. In a fully observed state, such as the one used in this dissertation, the policy is $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ where \mathbf{s}_t is the full state - here the policy defines which action to take based off which state is observed at the time of the action. This is shown in Figure [1](#).

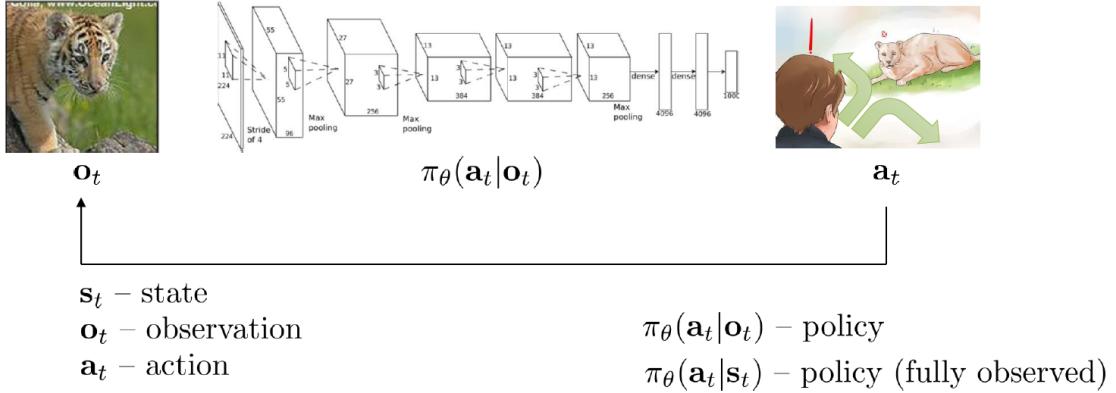


Figure 1: Mapping observed state to actions with a neural network. The image of a tiger is seen and a set of possible actions are assigned values such that the action with the highest likely value is chosen with higher probability by the agent. The mapping of the observation to likely action values is learned by the neural network (policy approximation function) over time from the agent’s experience within the environment (Levine, 2018b).

In Figure 2 we see a special case of RL where the actions and observations are already paired based on the experience of an expert, which could be an observed human. In this case, the problem is much closer to supervised learning in that there is very little room for exploration, and taking potentially incorrect actions - the pairs are assumed to be correct, and are not linked to some reward function. It is important to understand the point of departure of RL from supervised learning - generally it is this aspect of exploration and some reward function that indirectly helps the learning process work out which actions are best. The concept of exploring versus more efficiently optimising is called the exploration-exploitation trade-off - see Section 2.2.7.

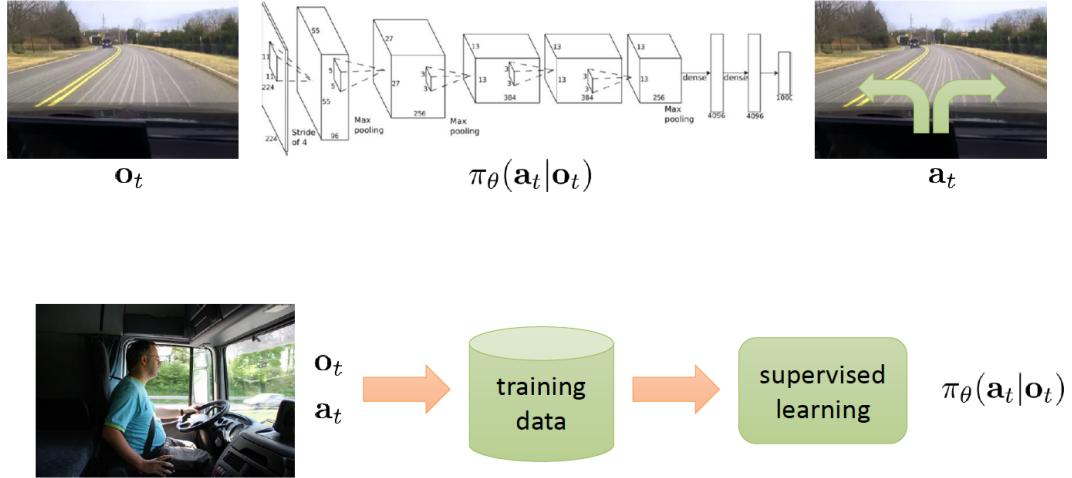


Figure 2: Mapping observed state to actions in imitation learning - some known “good” set of state-action pairs are used to train a supervised machine learning model ([Levine, 2018b](#)).

The reward function is defined by $r(s, a)$ and defines the utility associated with certain state-action transitions. Lastly, the probability of moving into the next state s' (or s_{t+1}) is given by $p(s'|s, a)$. These components (states, actions, reward function, and probability of the next state) define a Markov decision process in agent-environment interaction. The process is shown in Figure 3 - the agent is trying to maximise its reward over time based on its actions within the environment at particular states.

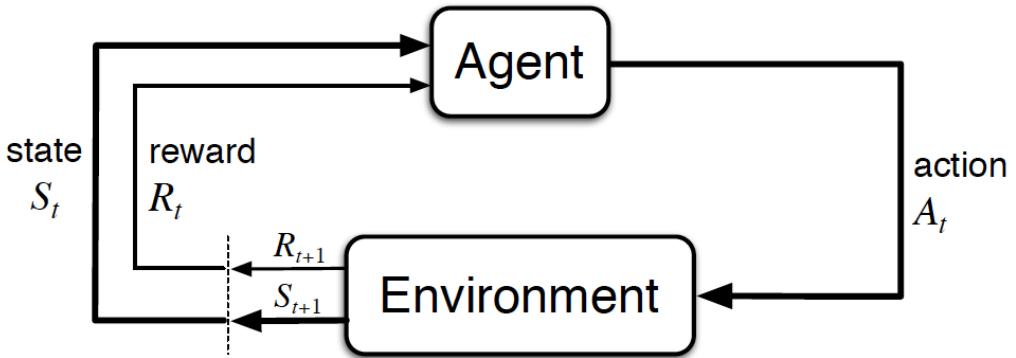


Figure 3: Agent-environment interaction in a Markov decision process. An agent takes some action after observing some state, then gets feedback via a reward signal from its environment. The agent then learns based off this information, observes the next state, and takes an action in the environment and gets the new reward signal ([Sutton and Barto, 2018](#)).

A way to define the probability of a new state given the previous state and the associated action is given by Equation 1 - this is known as the transition operator.

$$\tau_{i,j,k} = p(\mathbf{s}_{t+1} = i | \mathbf{s}_t = j, \mathbf{a}_t = k) \quad (1)$$

The transition operator describes all the probabilities that successive states will occur given previous states and potential actions. Equation 2 describes the policy, where the parameters are defined by θ , for the agent in the environment.

$$\pi_\theta(\tau) = p_\theta(\mathbf{s}_1, \mathbf{a}_1, \dots, \mathbf{s}_T, \mathbf{a}_T) = p(\mathbf{s}_1) \prod_{t=1}^T \pi_\theta(\mathbf{a}_t | \mathbf{s}_t) p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t) \quad (2)$$

The goal of the agent is to maximise its reward over time, and so we are interested in finding the set of parameters θ^* that allow for the policy with the greatest amount of reward over the time the agent is in the environment. This process is described in Equation 3 (Levine, 2018b).

$$\theta^* = \operatorname{argmax}_\theta E_{\tau \sim p_\theta(\tau)} \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right] \quad (3)$$

However, neither \mathbf{r} nor \mathbf{s} nor \mathbf{a} are available to begin with. The agent must interact with the environment as a way to gather this information and learn by updating θ , then taking actions based on this new policy. The process repeats as the agent becomes more competent and θ gets closer to θ^* . This is illustrated in Figure 4.

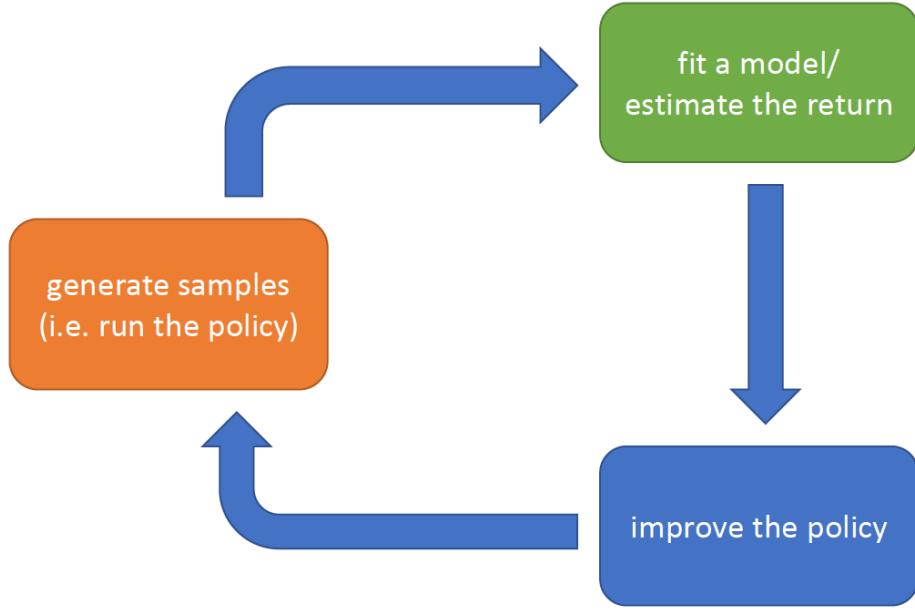


Figure 4: Loop for updating a policy based off environment interaction. The agent takes actions based off observed states in an environment, then calculates a better model based off new state, action, and reward information. Then the agent uses this model to improve its policy given some state, and uses this policy to take actions in the environment thus generating more samples to learn from and improve (Levine, 2018b).

After interacting with the environment, a model is fitted - this is known as the Q-function. It gives the predicted rewards of taking the set of available actions \mathbf{a}_t in the state \mathbf{s}_t . The Q-function is shown in Equation 4 (Levine, 2018b).

$$Q^\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^T E_{\pi_\theta}[r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t, \mathbf{a}_t] \quad (4)$$

The Q-function can be shown to give an expected value for a given state - this is expressed as a value function in Equation 5.

$$V^\pi(\mathbf{s}_t) = E_{\mathbf{a}_t \sim \pi(\mathbf{a}_t | \mathbf{s}_t)} [Q^\pi(\mathbf{s}_t, \mathbf{a}_t)] \quad (5)$$

This same value function can be formulated in another way which gives the *Bellman equation* (Equation 6). Here the value in \mathbf{s}_t is shown to be dependent on the value (given the same policy)

in \mathbf{s}_{t+1} , or \mathbf{s}' . This equation introduces a new variable γ (the discount factor) that decreases the impact of future rewards). The variable may be set to 1 where there is no difference between rewards now and in the future, but it is typically set to < 1 .

$$V^\pi(\mathbf{s}_t) = \sum_{a_t} \pi(\mathbf{a}_t | \mathbf{s}_t) \sum_{\mathbf{s}_{t+1}, \mathbf{r}_t} p(\mathbf{s}_{t+1}, \mathbf{r}_t | \mathbf{s}_t, \mathbf{a}_t) [\mathbf{r}_t + \gamma V^\pi(\mathbf{s}_{t+1})] \quad (6)$$

The value function and Q-function are used in different ways in various RL methods. In actor-critic approaches the Q-function is fit at the point of the green box in Figure 4. This model is used to update the policy at the point of the blue box, after which the policy is run in the environment in order to generate new samples. The model fitting and policy updating are separate in actor-critic - this will be explained in more detail in Section 2.2.4.

2.2.3 Policy gradient approaches

While the majority of the literature review aims to focus on methods used in the dissertation experiment, it is worth spending some attention on policy gradient approaches. Firstly, by understanding this *simpler* approach it gives a more enhanced understanding of RL before discussing actor-critic approaches. Secondly, the initial experiments that informed the basis of the dissertation were undertaken with vanilla policy gradients building on work by [Karpathy \(2016a\)](#) - see Appendix A for details.

The policy gradient method learns a parametrized policy that can select actions without consulting a value function ([Sutton and Barto, 2018](#)). We can describe the policy taken by an agent with Equation 7:

$$\pi(a|s, \theta) = Pr\{A_t = a | S_t = s, \theta_t = \theta\} \quad (7)$$

where π is the policy describing the action taken given the state s and the policy parameters θ . The goal is to find parameters θ such that the policy π maximises the expected reward received over the course of the task / game. The parameters are updated using gradient ascent:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)} \quad (8)$$

where $J(\theta)$ is some performance function, and $\widehat{\nabla J(\theta_t)}$ is an estimate of the gradient of the performance function with respect to θ at time t . The learning rate is given by α . See Section 2.3.3 for more detail and examples on performance functions.

The overall idea is to update the policy parameters directly based on the rewards calculated from the sampled observations. The approach is simpler conceptually than many other methods, but has been noted to be slow and difficult to get working by [Karpathy \(2016a\)](#). He suggests using a version of policy gradients known as Trust Region Policy Optimization ([Schulman et al., 2015](#)) that offers more robust performance - monotonic improvements with little need to tune hyper-parameters.

2.2.4 Actor-critic approaches

The actor-critic approach (often described as “advantage actor-critic” according to [Sutton and Barto \(2018\)](#)) builds on the policy gradient approach in Section 2.2.3, as well as the concepts of a value function and a Q-function in Section 2.2.3.

A key concept is the introduction of an advantage function, shown by Equation 9, where $Q^\pi(\mathbf{s}_t, \mathbf{a}_t)$ is given by Equation 4 and $V^\pi(\mathbf{s}_t)$ is given by Equation 5.

$$A^\pi(\mathbf{s}_t, \mathbf{a}_t) = Q^\pi(\mathbf{s}_t, \mathbf{a}_t) - V^\pi(\mathbf{s}_t) \quad (9)$$

The advantage function measures the difference between the observed actions and rewards, and the expectations of the actions and rewards. The larger the difference, the greater the changes to the parameters of the policy because the policy is likely further away from the optimum than when the difference is small (this is analogous to gradient descent - see Section 2.3.7). The changes to the policy parameters are given by Equation 10.

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) A^\pi(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \quad (10)$$

The policy is then updated in a similar way to Equation 8, shown by Equation 11.

$$\theta = \theta + \alpha \nabla_{\theta} J(\theta) \quad (11)$$

Of course, before applying Equation 11, the following equations need to be fit with the observed data: 4 and 6, then 9.

Levine (2018a) outline the basic actor-critic algorithm as follows, where ϕ are the parameters of the value function:

Algorithm 1: The actor-critic algorithm

- 1 sample $\{\mathbf{s}_i, \mathbf{a}_i\}$ from $\pi_{\theta}(\mathbf{a}|\mathbf{s})$ after running the policy
 - 2 fit $\hat{V}_{\phi}^{\pi}(\mathbf{s})$ as per Equation 6
 - 3 evaluate $\hat{A}^{\pi}(\mathbf{s}_i, \mathbf{a}_i) = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \hat{V}_{\phi}^{\pi}(\mathbf{s}'_i) - \hat{V}_{\phi}^{\pi}(\mathbf{s}_i)$
 - 4 Calculate $\nabla_{\theta} J(\theta)$ as per Equation 10
 - 5 $\theta = \theta + \alpha \nabla_{\theta} J(\theta)$
-

In Algorithm 1, $y_{i,t} \approx r(\mathbf{s}_i, \mathbf{a}_i) + \hat{V}_{\phi}^{\pi}(\mathbf{s}_{i,t+1})$ and the loss function when fitting the value function is given by Equation 12.

$$\mathcal{L}(\phi) = \frac{1}{2} \sum_i \|\hat{V}_{\phi}^{\pi}(\mathbf{s}_i) - y_i\|^2 \quad (12)$$

The overall process can also be visualized with Figure 5. The agent operating in an environment generates samples of actions, states, and rewards. These values are used to fit a critic function (V) that is used to update the actor policy parameters (θ). The new agent takes more actions in the environment and the process continues while the agent learns and takes better actions over time.

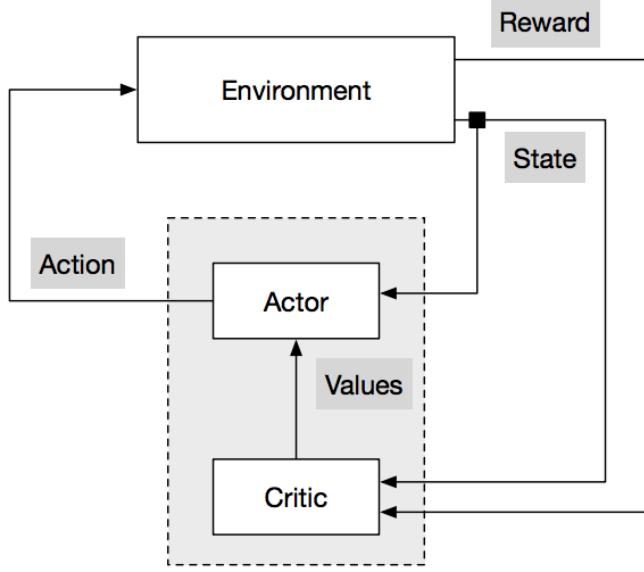


Figure 5: The actor-critic process - an agent takes actions in an environment based off input from a critic. These actions result in some reward signal which is used to learn whether the action was beneficial or not such that the actor and critic make more competent future actions when confronting a similar state ([Gašic, 2017](#)).

2.2.5 Asynchronous advantage actor-critic (A3C)

Mnih et al. (2016) introduced a variant on the actor-critic approach they called *Asynchronous Advantage Actor-Critic* (A3C). At the time, this approach was shown to surpass the current state-of-the-art methods on Atari games (such as Pong) while using far fewer computational resources.

Figure 6 is helpful for understanding each of the components of A3C at a conceptual level. Firstly, there are the two A's that represent advantage actor-critic which is present in each of the light blue boxes. These are the standard actor-critic components detailed in Section 2.2.4. The third A is for asynchronous which is represented by the way the multiple workers connect with the global network.

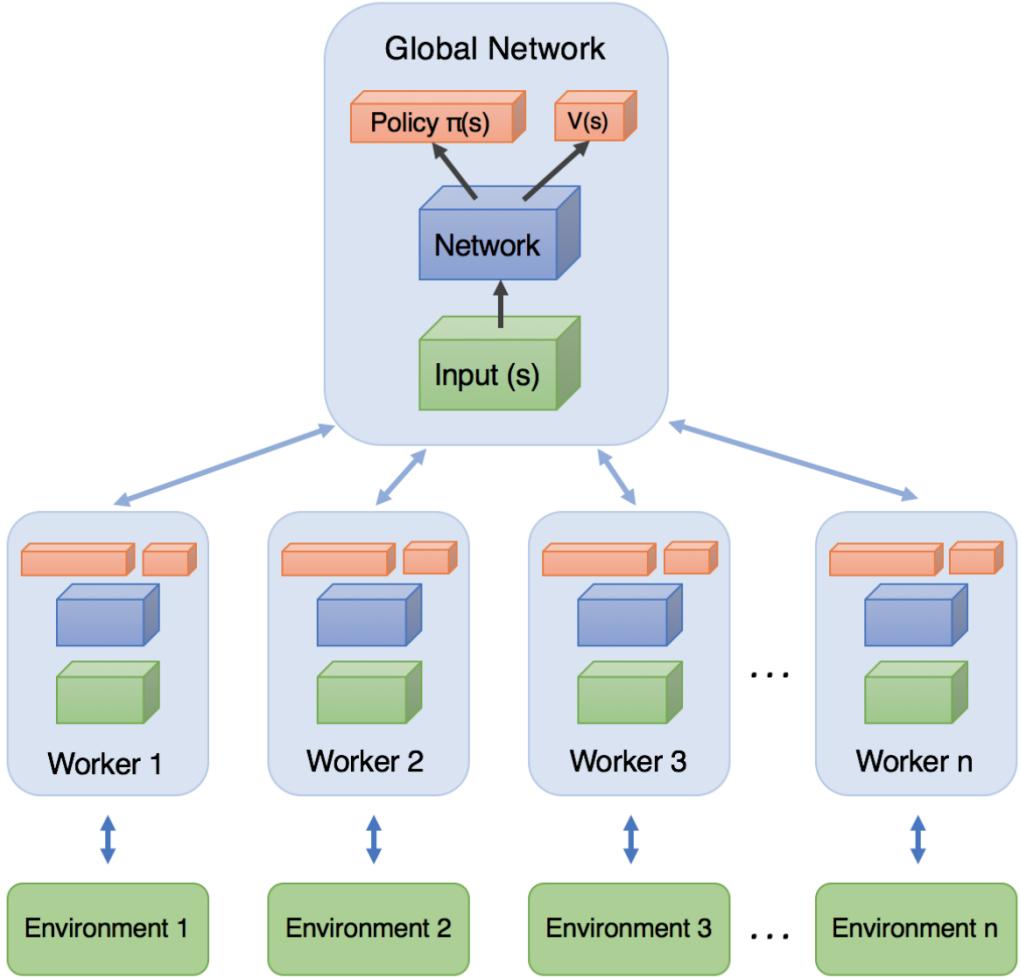


Figure 6: The three A's of A3C are shown in this diagram. The advantage actor-critic is represented by the overall structure of each of the blue boxes (both globally and at the worker level) where an advantage function is used to update an actor with input from a critic (value function). The asynchronous aspect is represented by each worker independently taking actions in their respective environments and providing parameter updates to the global model, and resetting to the global model, at different times to other workers (Juliani, 2016).

Each worker interacts with their environment and generates samples as per Step 1 in Algorithm 1. These workers are interacting with the environment independently. However, the θ that is being updated as per Step 5 is part of the global policy - it is receiving parameter updates *asynchronously* as each worker finishes some defined episode of operations. After each update, the workers then sync with the global network (i.e. initialise to the global parameters) and continue learning in their individual environments. Because each worker is getting a slightly different version of the global

network based off their own parameter updates and some order of updates from previous workers, it then takes this unique policy back to its environment allowing for a wider range of exploration. A simple diagram of the process is shown in Figure 7. One of the primary advantages of the A3C system in practice is that it can be computationally efficient as workers can scale in parallel. This allows for an almost linear ability to accelerate the training time.

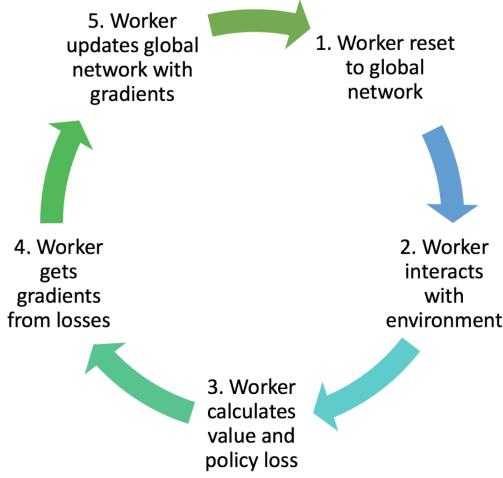


Figure 7: A simple representation of the A3C process, which is described in more detail in Algorithm 2. During each cycle a worker both updates the global parameters, and is updated by the global parameters. This cycle is repeated for each worker (Juliani, 2016).

Algorithm 2 details A3C more comprehensively at the worker thread level. Each of the worker boxes in Figure 6 undergoes these steps during training. It is worth emphasizing the asynchronous nature of this loop: Each worker will encounter a different set of steps and episodes because they are operating in different environments, with their slightly different parameters taking different decisions even when encountering the same states. Episodes will have different lengths so that the terminal s_t will occur at different times creating an effectively random order of updates and synchronisations with the global parameters θ and θ_v .

Algorithm 2: The asynchronous actor-critic algorithm Mnih et al. (2016)

```

// Assume global shared parameter vectors  $\theta$  and  $\theta_v$  (actor and critic / value function,
respectively) and global shared counter  $T = 0$ 

// Assume thread specific parameter vectors  $\theta'$  and  $\theta'_v$ 

Initialize thread step counter  $t \leftarrow 1$  repeat
    Reset gradients:  $d\theta \leftarrow 0$  and  $d\theta_v \leftarrow 0$ .
    Synchronise thread-specific parameters  $\theta' = \theta$  and  $\theta'_v = \theta_v$ 
     $t_{start} = t$ 
    Get state  $s_t$ 
    repeat
        Perform  $a_t$  according to policy  $\pi(a_t|s_t; \theta')$ 
        Receive reward  $r_t$  and new state  $s_{t+1}$ 
         $t \leftarrow t + 1$ 
         $T \leftarrow T + 1$ 
    until terminal  $s_t$  or  $t - t_{start} == t_{max}$ ;
     $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \end{cases}$ 
    for  $i \in \{t - 1, \dots, t_{start}\}$  do
         $R \leftarrow r_i + \gamma R$ 
        Accumulate gradients w.r.t.  $\theta' : d\theta + \nabla_{\theta'} \log \pi(a_i|s_i; \theta') (R - V(s_i; \theta'_v))$ 
        Accumulate gradients w.r.t.  $\theta'_v : d\theta_v \leftarrow d\theta_v + \delta (R - V(s_i; \theta'_v))^2 / \delta \theta'_v$ 
    end
    Perform asynchronous update of  $\theta$  using  $d\theta$  and of  $\theta_v$  using  $d\theta_v$ .
until  $T > T_{max}$ ;

```

This asynchronous version of actor-critic was shown to surpass the current state-of-the-art in the Atari domain by Mnih et al. (2016). As previously mentioned, not only was performance improved, but training time was halved. Additionally, new tasks were able to be mastered that had eluded previous approaches. Figure 8 shows the A3C performance on multiple tasks compared to other prominent methods.

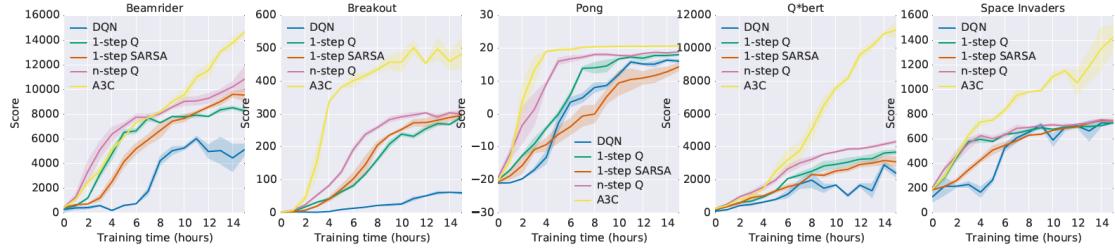


Figure 8: A3C compared to other popular RL methods on multiple Atari games. In the middle chart the game of Pong is shown to be more quickly mastered with A3C than with other methods, after about 4 hours on the specific setup at Google DeepMind. On other tasks A3C is shown to result in even wider margins of performance over other methods (Mnih et al., 2016).

Other advantages of A3C noted by Mnih et al. (2016) include the ability to achieve the result with multiple CPUs instead of a GPU. Given the good results and low resource requirements of A3C, it was considered a good candidate to be used as the RL framework in this dissertation.

2.2.6 Note on Markov processes

As noted in Section 2.2.2, a Markov decision process (MDP) comprises a reward function $r(\mathbf{s}, \mathbf{a})$, the state \mathbf{s}' (or \mathbf{s}_{t+1}), and the probability of moving into the next state $p(\mathbf{s}'|\mathbf{s}, \mathbf{a})$. The MDP is represented visually in Figure 3. However, some Markov processes have a special property called the *Markov property of memorylessness*. To quote Sutton and Barto (2018), “A function f has the Markov property if and only if any two histories h and h' that are mapped by f to the same state ($f(h) = f(h')$) also have the same probabilities for their next observation.” In other words, previous states do not matter - only the current state does.

In the dissertation experiment, it is important to note that the task does not have the Markov property. A simple way to think about this is by visualizing a frame, as in Figure 34. The ball is at a specific point in the frame but it is unknown which direction it is travelling in and therefore an RL agent would potentially take a different action based on whether it knew this information or not. Hence, the Markov property is not satisfied because the state does not contain all the information and it is important to have some information about previous states.

The experiment deals with the lack of the Markov property by making use of a recurrent neural network (specifically, a LSTM) as described in Section 2.3.11.

2.2.7 Note on exploration vs. exploitation

As the RL agent learns better parameters with respect to achieving a higher score on the task, it is possible that it gets stuck and behaves deterministically such that it takes the same action given some state that may not be optimal overall. In certain cases this can lead to a complete failure of the RL learning process where the task score is worse than that achieved by taking random actions. This was observed in Appendix A where the running score of the agent can be seen to collapse to the minimum even after significant progress was made. Several strategies are employed as a way to combat this effect. From a neural network perspective, gradient-clipping, lowering the learning rate, and the Adam optimiser were all tried (with Adam and learning rate adjustments eventually being used in the experiment), but a simple RL strategy is to force a particular exploration vs. exploitation trade-off. As a simple definition, [Sutton and Barto \(2018\)](#) put it well: “exploration means trying actions that improve the model, whereas exploitation means behaving in the optimal way given the current model.” Furthermore, they write: “we want the agent to explore to find changes in the environment, but not so much that performance is greatly degraded.”

When using a neural network model, the initialization of parameters should lead to a random model at first. In the case of the Pong task, a random model has an EMA (Exponential Moving Average) of approximately -20.4 . A model that collapses (i.e. is unable to find parameters that help stop it from deterministically failing) will eventually converge on -21.0 as a running score. When this happens, an alternative to neural network training techniques could be to use a strategy such as the ϵ -greedy approach described in Equation 13.

$$\mathbf{A} = \begin{cases} \text{argmax}_a Q(\mathbf{a}) & \text{with probability } 1 - \epsilon \\ \text{a random action} & \text{with probability } \epsilon \end{cases} \quad (13)$$

The approach is simple. For some proportion of actions, selected randomly with probability ϵ , select an action randomly from the set of possible actions. The technique ensures that it is impossible to only choose a single action over and over again - for example, always choosing to move downwards in Pong. The goal is to continue some exploration no matter what so that the model eventually has some positive rewards (by chance) to help it adjust its parameters and eventually be able to make new action choices when choosing an action that it thinks maximises future rewards.

Different values of ϵ can be chosen, and it is itself a hyper-parameter that could play a role in an experiment such as the one undertaken in this dissertation. The different values may have an impact similar to that shown by [Sutton and Barto \(2018\)](#) in Figure 9. Lower values of ϵ allow for less random actions, but mean the model may find an optimal solution slower (or perhaps never at all, depending on the task). But, a higher ϵ that helps the model improve at a quicker rate initially may prevent a higher longer-term optimum. A solution suggested by [Sutton and Barto \(2018\)](#) is to decrease ϵ over time.

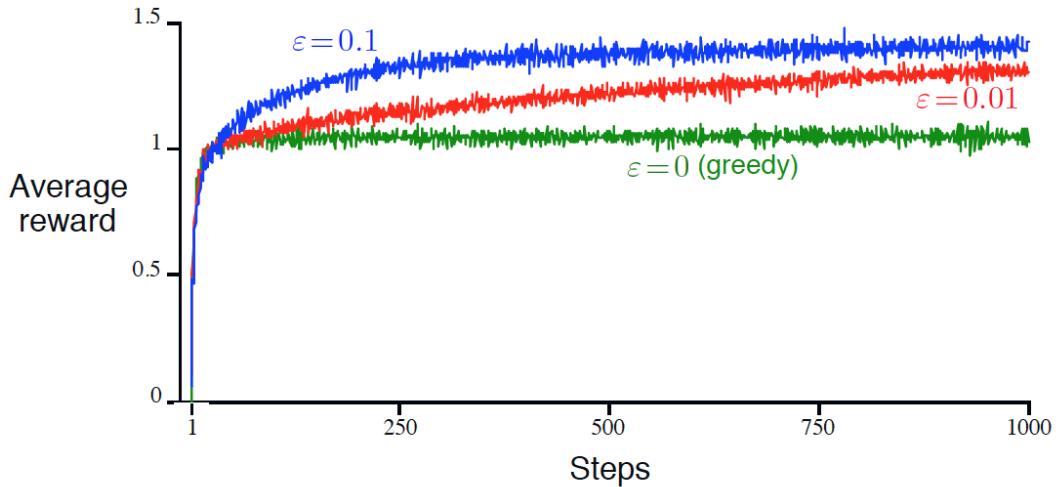


Figure 9: Different levels of ϵ result in different paths to optimality when using an ϵ -greedy method to manage the exploration-exploitation trade-off. A lower value shows a slower rate of improvement, but does seem more likely to eventually surpass the higher value ([Sutton and Barto, 2018](#)).

Ultimately, it was not necessary to use the ϵ -greedy approach in this dissertation, but it is worth noting for its simplicity and effectiveness - as well as being an explicit and intuitive representation of the trade-off between exploration and exploitation.

2.2.8 Reward functions

A challenge in RL, as compared to more classic supervised learning problems, is the inability to directly link an action to a reward in many cases. In supervised learning there are examples, each with a set of features \mathbf{X} and some label \mathbf{Y} - an example may be an image of a cat, with the label “cat”. Or, some set of attributes of a house with the label being the price of the house. A model would then learn to map each set of input features to the annotated label. In RL the challenge is

twofold:

- In the beginning of training, a model is still doing more exploring than exploiting the known ways to act given a state.
- Rewards can be sparse such that most actions are not followed directly by a reward from the environment, but a set of state-action pairs may result in an environment reward at some point in the future. Figure 10 illustrates this concept well.

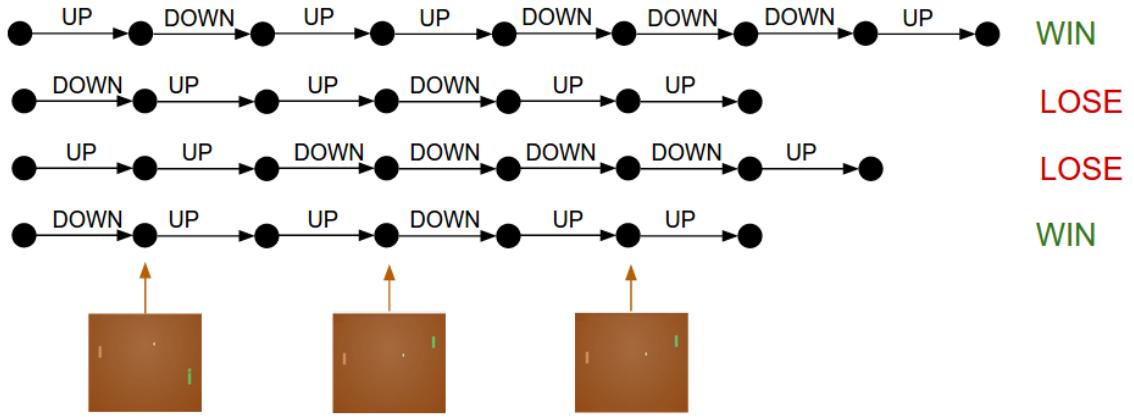


Figure 10: Different actions from an agent acting in the Pong environment will result in an eventual reward. Either a set of actions will result in losing a round and the reward would be negative, or a positive reward could result from winning the round. However, the key point is that the reward belongs to a set of actions as opposed to a direct action-reward relationship from the environment (Karpathy, 2016a)

The way the first challenge is dealt with is by limiting the size of the parameter updates to account for the fact that many actions sub-optimally lead to some positive reward, or many good actions ultimately lead to some negative reward. Hence, the learning rate is a critical hyper-parameter in the experiment.

The second challenge is dealt with by discounting future rewards. The simplest way of representing the total reward G_t of the state-action pair at time t is with Equation 14 (Sutton and Barto, 2018).

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \dots + R_T \quad (14)$$

However, the equation above treats the rewards of each time period exactly the same. For example, if I took an action now and it resulted in a positive reward in 20 moves time, it would result in the same total rewards for the state-action pair as a move that may result in a positive reward in just 5 moves time. But, we would prefer to allocate a higher reward to the action that resulted eventually in receiving the reward after less moves. We deal with this via discounting - Equation 15 show a smaller modification where we introduce the parameter γ , where $0 \leq \gamma \leq 1$.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{T-1} R_T \quad (15)$$

Future rewards are worth less than more immediate rewards, and the RL model will optimise for those actions which result in sooner positive rewards, and more distant negative rewards. In this dissertation γ was fixed at 0.99.

Other aspects of reward functions beyond the scope of this dissertation include techniques such as reward shaping. According to Ng et al. (1999) this is a “method used in reinforcement learning whereby additional training rewards are used to guide the learning agent.” This is particularly helpful in further dealing with the sparse reward problem described above where future rewards are just so far ahead, and require very complex sequences of actions in order to achieve them. For example, in the game of Pong an additional reward may be to gain some reward when the ball is hit, or when the paddle is closer to the ball. In more complex domains with several thousands of moves as a time horizon (as well as other issues including: partially observed states; high-dimensionality and continuous action and observation spaces) this has proved useful, for example in OpenAI (2018). However, in some seemingly complex domains like the game of Go the number of total moves is typically less than 150 and reward shaping was not required (Silver et al., 2016).

2.2.9 Conclusions

Reinforcement learning was introduced as an extension of supervised learning where an agent is required to explore an environment as a way to improve on a task with a specific quantifiable goal. The main concepts in RL were touched on, including Q-functions and value functions, with a brief overview of the policy gradient approach as a grounding for further understanding. Actor-critic approaches introduced the advantage function, and the specific version of actor-critic, the asyn-

chronous advantage actor-critic (A3C), used in this dissertation was described. Further clarification on Markov processes, the trade-off between exploration and exploitation, and how rewards are handled, was given.

2.3 Neural network design

2.3.1 Introduction

Neural networks are a particular model class within machine learning that are particularly adept with image data (the state input type within the dissertation experiment). They are widely used within the realm of supervised learning where the task is concerned with learning how to predict some vector \mathbf{y} based off the input of some vector \mathbf{x} . In general, supervised machine learning involves observing examples of \mathbf{x} with an associated \mathbf{y} and learning to predict \mathbf{y} from \mathbf{x} by estimating $p(\mathbf{y}|\mathbf{x})$. This procedure is performed during RL once the loss function and labels are defined (one can think of RL as a technique to convert data observed from exploration into a supervised learning task), so neural networks will be discussed in a supervised learning context.

This section gives a high-level introduction to neural networks (Section 2.3.2), touching on key concepts such as backpropagation (Section 2.3.4), loss functions (Section 2.3.3), activation functions (Section 2.3.9), regularization (Section 2.3.6), and learning rates (Section 2.3.5). More advanced topics relevant to the dissertation are then introduced with the aim of ensuring the reader is familiar with the essential concepts employed in the experiment methodology. Topics include convolutional networks (Section 2.3.10), recurrent networks (Section 2.3.11), parameter updating methods (Section 2.3.7), and gradient clipping (Section 2.3.8). Particular emphasis will be placed on aspects of the experiment that were the most important in finding good model designs.

2.3.2 Neural networks

According to [Hastie et al. \(2009\)](#), a neural network is a two-stage model that can be represented by a diagram such as the one in Figure 11. When the problem is that of classification with K classes there is a Y_k for all $k = 1, \dots, K$ with each either being labelled a 0 or a 1. The Z layer with the derived features Z_m is created from linear combinations of the layer of inputs X . Then Y_k is a linear combination of Z_m . This is represented mathematically by Equation 16:

$$\begin{aligned}
 Z_m &= \sigma(\alpha_{0m} + \alpha_m^T X), m = 1, \dots, M, \\
 T_k &= \beta_{0k} + \beta_k^T Z, k = 1, \dots, K, \\
 Y_k &= f_k(X) = g_k(T), k = 1, \dots, K
 \end{aligned} \tag{16}$$

σ represents an activation function which is typically some non-linear function such as a sigmoid shown in Equation 32. A neural network therefore differs from regression in that Equation 16 would be a regression if $g_k(T) = T_k$, and the activation function σ was linear. The middle layer which consists of the Z_m features is known as a *hidden layer* given that it is not directly observed as an output. There can be many hidden layers, and typically the term *deep* is used to describe neural networks where the number of hidden layers is > 1 .

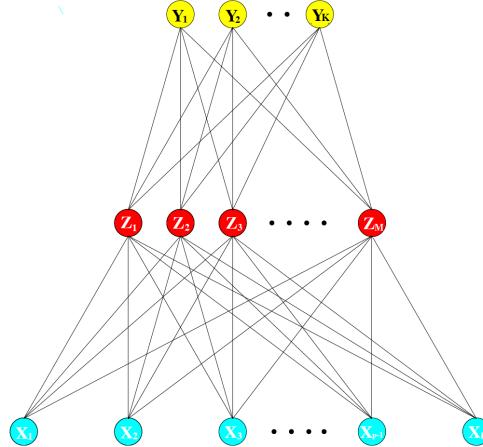


Figure 11: A diagram of a single hidden layer neural network. The bottom row shows the inputs of \mathbf{X} feeding forward to the top row output \mathbf{Y} after some transformation layer \mathbf{Z} [Hastie et al. \(2009\)](#).

The overall process of developing neural networks, and many other machine learning techniques, that accurately predict outcomes based on some input is shown in Figure 12. While the goal is to be able to perform inference based off the trained network (shown in the bottom half of the figure), the steps taken to train the network are represented by the top half of the figure. Typically, a large set of examples are passed through the network where some predicted output is observed and then compared to the actual known output (in this case it is the predicted label “dog” where the known output is actually “human face”). When there is a difference between the predicted value

and the actual labelled value then there is some non-zero error calculated. This error is calculated using some loss function (see Section 2.3.3) and then used to adjust the parameters (also known as weights) of the network via a process known as backpropagation (see Section 2.3.4). The process is iterative and many examples may be used during any one training step (depending on the batch size) with the learning rate (see Section 2.3.5) used to regulate the size of the adjustments made to the parameters. Other methods are also used to adjust the size of the parameter updates, including regularization (Section 2.3.6), gradient descent optimization (Section 2.3.7), and gradient clipping (described in Section 2.3.8 but not used in the dissertation experiment).

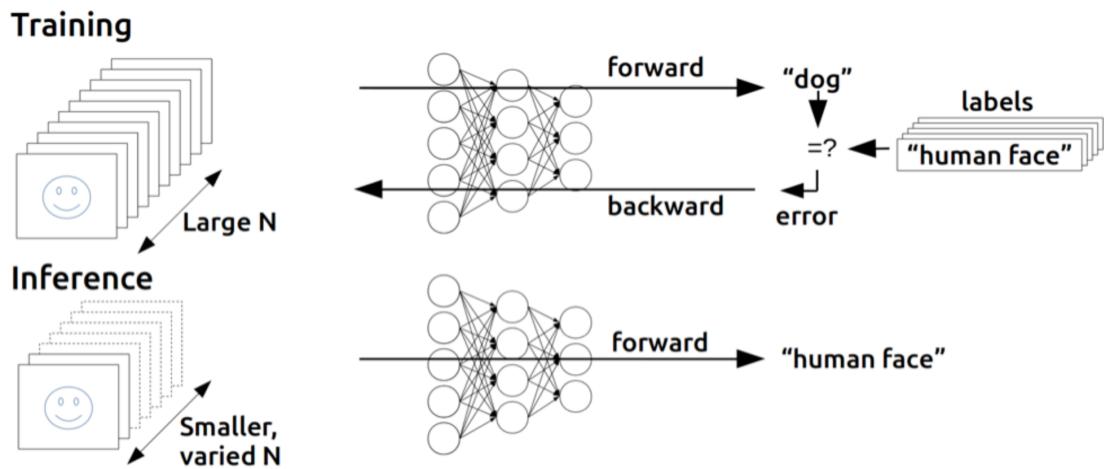


Figure 12: An overview of neural network training and inference. In the top half a neural network is trained after having many example inputs passed forward through the network and observing predicted outcomes, then comparing the outcome to a known labelled value, calculating an error, and using the error to work backwards through the network updating parameters by means of backpropagation. In the bottom half of the diagram a trained network makes accurate predictions with unseen inputs (Andersch, 2015).

2.3.3 Loss functions

Loss functions allow us a mechanism to measure how far away the predicted value is from the true value. When predicted value and true value are the same, then the loss should be 0. There are different ways of measuring loss, in particular for regression tasks (the output is continuous), and for classification tasks (the output is discrete).

Regression loss is typically calculated using the sum of squared errors between the estimated output $f_k(x_i)$ and the true output y_{ik} , as per Equation 17. When the goal is to minimise the

loss from the network, larger deviations are penalised more due to the square. Neural network regressions minimise this loss by means of gradient descent, although it is commonly minimised analytically when doing linear regression ([Hastie et al., 2009](#)).

$$J(\theta) = \sum_{k=1}^K \sum_{i=1}^N (y_{ik} - f_k(x_i))^2 \quad (17)$$

For classification tasks where there are K categories, Equation 18 is typically used. It can be shown that when a certain prediction for category k for example x_i when the true value is 1 will result in a value of 0 for the inner part of the summation. Conversely, a certain prediction that x_i is not k when the true result is 0 will also result in a value of 0 for the inner part of the summation. However, when the prediction is completely wrong, 1 vs. 0, then inner summation will be equal to 1 ([Hastie et al., 2009](#)).

$$J(\theta) = - \sum_{k=1}^K \sum_{i=1}^N y_{ik} \log f_k(x_i) \quad (18)$$

A special version of this categorical loss is used within the actor-critic approach to reinforcement learning. The specific gradients applied to the parameters were shown in Equation 10, but the loss itself is represented by Equation 19 where log of the vector of policy actions taken is multiplied by the advantage in the inner part of the summation. In this case the advantage function is representing the true value analogous to the y_{ik} in Equation 18.

$$J(\theta) = \frac{1}{N} \sum_{i=1}^N \sum_{t=1}^T \log \pi_\theta(\mathbf{a}_{i,t} | \mathbf{s}_{i,t}) A^\pi(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \quad (19)$$

2.3.4 Backpropagation

As [Hastie et al. \(2009\)](#) notes, the general approach to iteratively minimizing the $J(\theta)$ values of the loss functions is by means of gradient descent, which is achieved through a technique called backpropagation. The gradient of the loss function with respect to each parameter in the model can be calculated by using the chain rule of differentiation. For parameters later on in the model

(closest to the final layer) the parameter gradients are calculated with respect to the loss function directly, whereas earlier parameters are then calculated with respect to the gradients after them, all the way to the earliest parameters. [Hastie et al. \(2009\)](#) give a comprehensive overview and example of backpropagation on page 396 using the sum of squared errors loss function.

Ultimately, the point of calculating these gradients is to update the parameters as per the general Equation 20.

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} J(\theta) \quad (20)$$

Here the gradient applied to the parameters is regulated by α which is known as the *learning rate*. As the training process iterates, the size of the gradients typically get smaller until the model reaches some local optimum and no further updates are required (this is consistent with a minimised loss function).

2.3.5 Learning rates

The learning rate in neural network training is a constant when used most simply i.e. a set number that does not change throughout the training process. However, a specific learning rate early in the training process may not be the right rate later on in the process, and so may be effectively adjusted by either a set rate of decrease over time, or other methods used in this dissertation such as gradient descent optimization.

When setting a constant learning rate, there is no universally optimal number. Depending on the task, a specific learning rate may be too small such that it never escapes a local optimum to find a global one, or too large that it never settles on any optimum. This concept is clearly illustrated in Figure 13.

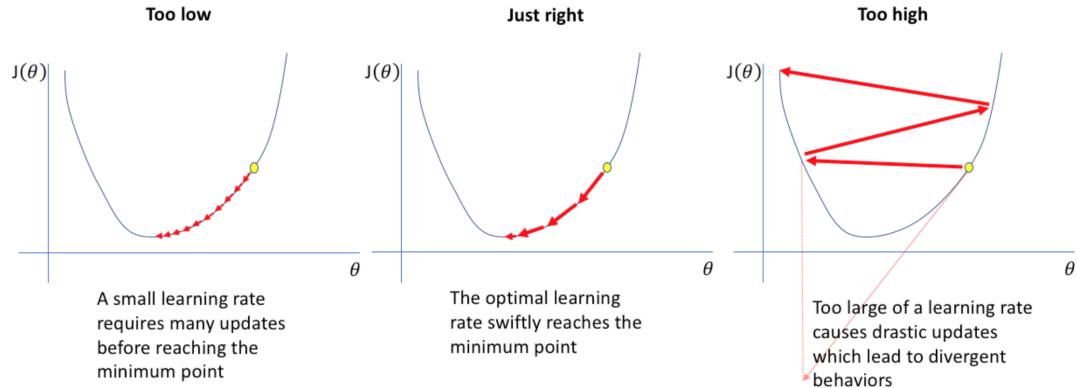


Figure 13: Setting a learning rate that is too low may take too long to update parameters of a network, or may never be able to escape a local optimum. A learning rate that is too high may diverge too much as it is never able to settle on any optima. A good learning rate allows reasonably sized parameter updates that help efficiently train and find good and stable optima for network parameters [Jordan \(2018\)](#).

Choosing the right learning rate can be achieved by attempting multiple values - similar to a one-dimensional hyper-parameter search. Figure 14 shows the result of having done this on an example network. A good learning rate for this network was on the order of 10^{-3} - large enough that training was efficient, but small enough to avoid divergence of the error.

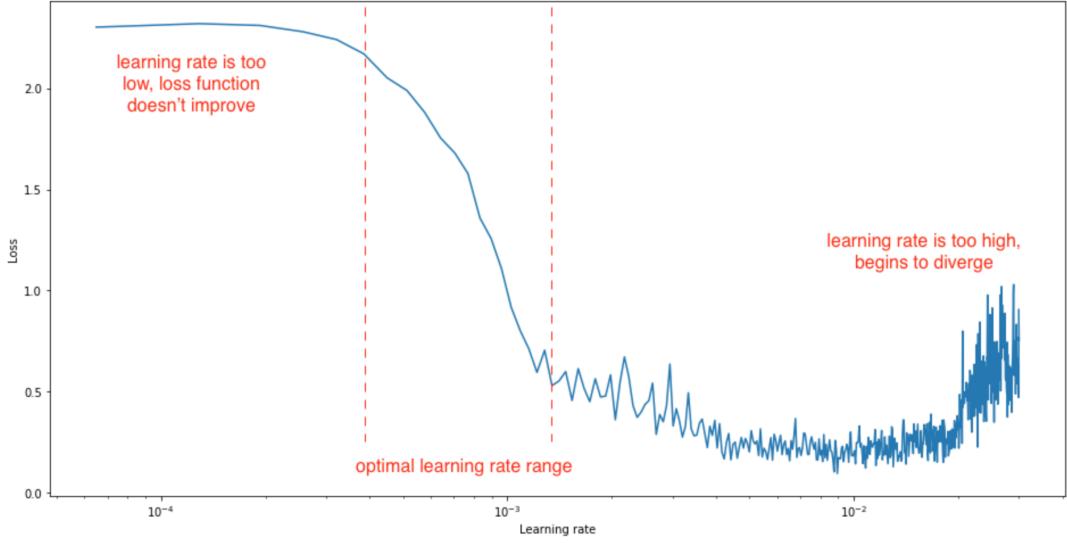


Figure 14: The loss after a certain amount of training for various learning rates. Lower learning rates show that they are unable to adequately reduce the loss in a given time, while higher learning rates show divergence - meaning that the loss is not stable, and the network parameters are likely not settling on local optima [Jordan \(2018\)](#).

In the dissertation experiment the benchmark OpenAI model ([Blackwell et al., 2018](#)) had a learning rate of 10^{-4} . [Mnih et al. \(2016\)](#) sampled from the distribution $\text{LogUniform}(10^{-4}, 10^{-2})$ when finding the optimal rate for their tasks. The results of their search are shown in Figure 15. The optimal learning rate for Pong (and most other tasks) was close to 10^{-3} .

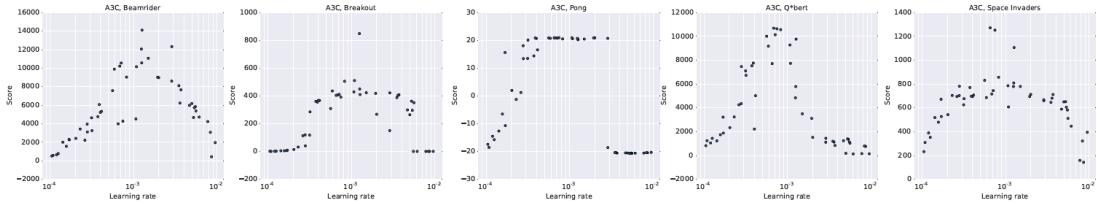


Figure 15: The results of A3C reinforcement learning across different tasks after sampling learning rates from $\text{LogUniform}(10^{-4}, 10^{-2})$. The best learning rate across most tasks seems to be close to 10^{-3} [Mnih et al. \(2016\)](#). In the charts the learning rate is on the x-axis, and the task scores are on the y-axis. The games, from left to right, are: Beamrider; Breakout; Pong; Q*Bert; Space Invaders.

2.3.6 Regularization

[Goodfellow et al. \(2016\)](#) note that “a central problem in machine learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs.” The topic

of over-fitting and under-fitting, generalization, bias and variance is large, and a more detailed examination is beyond the scope of this dissertation. However, the key concept is to ensure that a model performs consistently well on unseen (i.e. not part of the training examples) inputs.

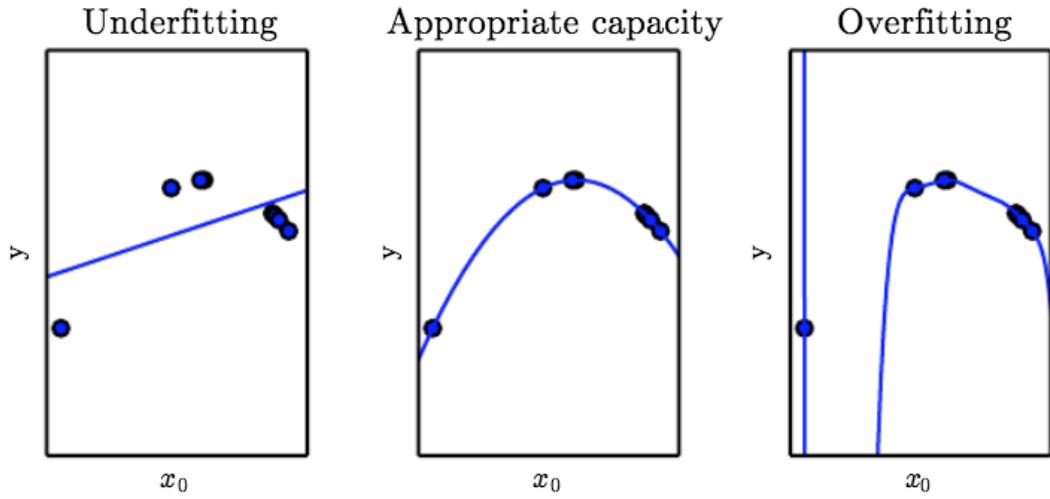


Figure 16: A demonstration of fitting a function to data. In the first panel a high bias model is fitted to the observed points such that there is some error between what is predicted and what is observed. However, in the right column the opposite problem is encountered where there is no error between what is predicted and what is observed on the training data, because the model is over-fitted and has high variance. A more optimal approach is that taken by the middle model which clearly has fewer parameters than the right model, and so is less able to overfit for the observed data, but it still results in a low error generally fitting the observations well ([Goodfellow et al., 2016](#)).

Regularization is the general technique where the capacity of a model is restricted, or penalised, in some way so that the ability to achieve a low error on training is limited, with the view that the model does well on unseen (test) data. Figure 17 shows this in a simple way - regularization helps find the model closest to the vertical red line.

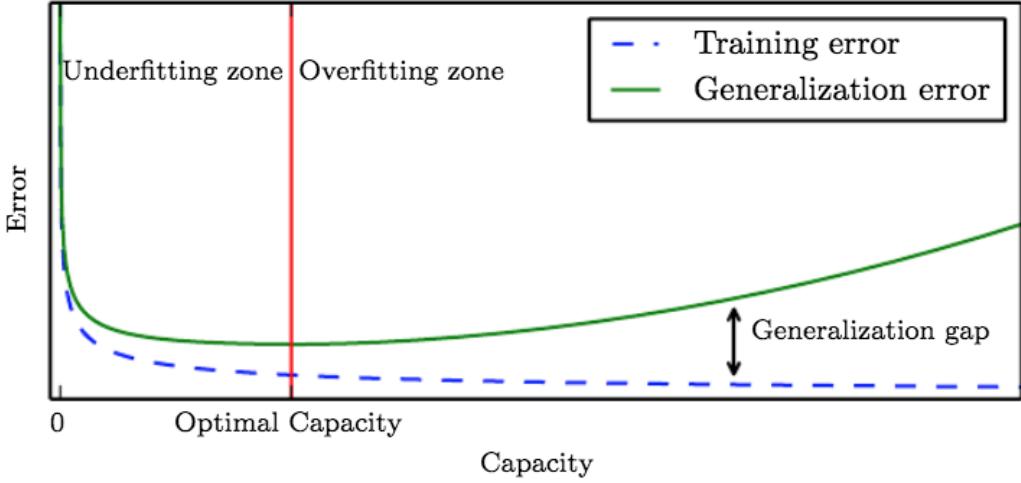


Figure 17: While Figure 16 gives specific examples of the generalization problem, this diagram shows it conceptually. The capacity of a model represents how much information it can contain (typically, the number of parameters or even their size). A more complex model may trivially fit the training data excellently by over-fitting for the observed cases, but could result in larger errors on unseen cases (the green generalization error line) (Goodfellow et al., 2016).

A general representation of regularization is shown in Equation 21 (Goodfellow et al., 2016). It is similar to the loss functions described in Section 2.3.3 but there is an extra term - $\eta\Omega(\theta)$ - known as the parameter norm penalty. The actual penalty $\Omega(\theta)$ is weighted by η which is just another hyper-parameter. The penalty may represent various measures of capacity and can be part of the training procedure (where it forms part of the gradients), or applied after a training procedure to compare different architectures.

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \eta\Omega(\theta) \quad (21)$$

In the dissertation experiment a form of regularization was incorporated in the tournament score function in Equation 46. While not directly penalising large / numerous weights, it does so indirectly by penalising networks that take longer to process. This regularization formed part of tournament selection and was not part of the gradients themselves. However, regularization has been shown to be effective (Goodfellow et al., 2016), with L^2 regularization (known as *ridge regression*) and L^1 regularization being two popular methods.

Another popular method (the most popular according to [Goodfellow et al. \(2016\)](#)) for regularizing neural networks is *early stopping*, which was implicitly used in the dissertation experiment. The method aims at stopping the training of a neural network after some point in time even as the training error begins to decrease, because it has been shown that validation error may begin to increase ([Goodfellow et al., 2016](#)).

2.3.7 Gradient descent optimization

For this section the learning rate will be represented by η in line with the most comprehensive source on this topic - [Ruder \(2016\)](#).

As previously noted, a simple constant learning rate may not be adequate. But there are other issues around making updates to parameters. Firstly there is the problem of the batch size. Using all the training data at once, as in Equation 22 (*Batch Gradient Descent*), is effective at getting towards a global optima, but does not allow additional examples when the model is *online* as is the case in many RL tasks. There may also be computational resource constraints when using batch gradient descent.

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta) \quad (22)$$

Alternatively, individual examples can be used to update the parameters as shown in Equation 23 - known as *Stochastic Gradient Descent* (SGD). The problem with SGD is that it may overshoot a local optimum based on the order of examples, and have volatile convergence towards an optimum.

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^i; y^i) \quad (23)$$

[Ruder \(2016\)](#) conclude that *mini-Batch Gradient Descent* as shown in Equation 24 is the best of both worlds - online learning while taking a smoother path to convergence. Additionally, the approach allows efficient use of computational resources by not requiring the calculation of gradients with respect to all examples at once.

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{i:i+n}, y^{i:i+n}) \quad (24)$$

However, even batch gradient descent is not adequate by itself as it does not guarantee that a model will converge on optimal parameters. Specifically, the challenges, according to [Ruder \(2016\)](#) include:

- Choosing an optimal learning rate can be challenging, as previously outlined in Section [2.3.5](#).
- Even using a learning rate schedule may not work as it needs to be designed for a specific task, and has its own set of hyper-parameters that need to be optimised.
- Even when getting a learning rate and its schedule right, the same rate applies to all parameters regardless of how close they are individually to their own optima.
- In a multi-dimensional space the parameters may get stuck at a saddle point where gradients are close to 0 - it could be difficult to escape this non-optimal minima.

While there are many methods that work to address these challenges, this section will outline three popular methods that build on each other: Momentum, RMSprop, and Adam (the method used in the dissertation experiment).

Momentum makes a simple adjustment to gradient descent by first adjusting the size of the gradients to include some weight of the previous gradients, in order to create less volatile gradient updates. This is shown by Equation [25](#) where γ is typically less than 1 (Note that this **not** the discount factor used in Section [2.2](#)). The benefits of Momentum are visually represented by Figure [18](#) - faster convergence and reduced oscillation ([Ruder, 2016](#)).

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \cdot \nabla_{\theta} J(\theta) \\ \theta &= \theta - v_t \end{aligned} \quad (25)$$

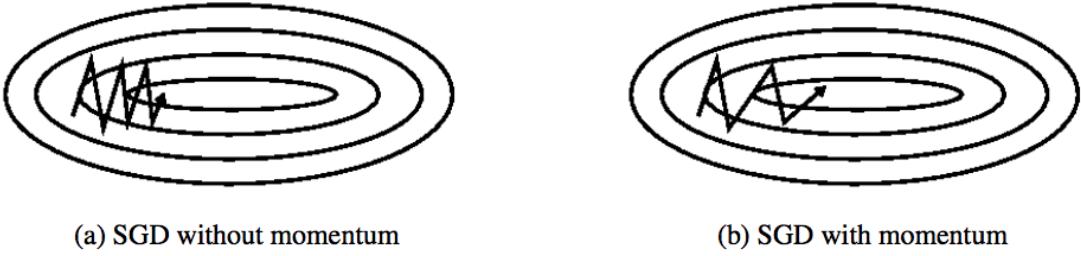


Figure 18: A visual representation of using Momentum to adjust gradients. The result is faster convergence and reduced oscillation ([Ruder, 2016](#)).

Momentum is known to give updates that may either over-correct, or not be as responsive as needed to changes in gradients according to [Ruder \(2016\)](#). RMSprop is a method originally developed by Geoffrey Hinton in an online course that is able address this. It is represented by Equation 26, and works by decaying the average of squared gradients over time. Geoffrey Hinton recommends setting γ to 0.9, and η to 0.0001.

$$\begin{aligned} E[g^2]_t &= 0.9E[g^2]_{t-1} + 0.1g_t^2 \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t \end{aligned} \tag{26}$$

Adaptive Moment Estimation (Adam) uses momentum adjustments as with the Momentum method, as well as decaying the squared gradient as with RMSprop. It tries to combine the best of both worlds by addressing the low responsiveness of pure Momentum, but sometimes over-responsiveness of RMSprop. However, both Adam and RMSprop are commonly used, and the choice depends on which might work better in practice. There are three steps in calculating the Adam gradients. Firstly, the momentum of the two moments are calculated (the mean and uncentered variance, respectively) as shown in Equation 27.

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned} \tag{27}$$

However, because the gradients are initialised at 0, there is some bias - hence the bias corrected estimates are calculated in Equation 28:

$$\begin{aligned}\hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t}\end{aligned}\tag{28}$$

The Adam update rule, Equation 29, is then calculated with the bias corrected estimates:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t\tag{29}$$

According to [Ruder \(2016\)](#) β_1 is typically chosen to be 0.9 and β_2 is chosen to be 0.999. Adam is known to provide the best performance across a wide range of tasks, and is a good default method of optimising gradient descent. Consequently, it was used as the method during the dissertation experiment.

2.3.8 Other notable techniques

A popular technique used in image classification as a further way to regularize the model (i.e. to sacrifice some training performance for better generalization) is the dropout technique introduced by [Srivastava et al. \(2014\)](#). Conceptually, the technique aims to transform the model into multiple ensemble networks by randomly turning off a specified percentage of network nodes during training as shown in Figure 19.

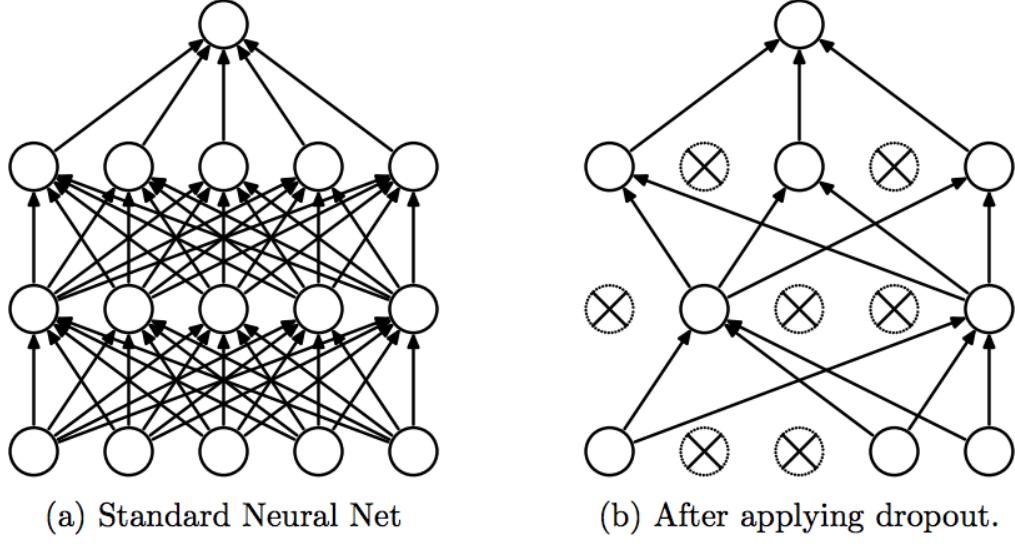


Figure 19: A network with and without dropout. On the left a network without dropout uses all the nodes / weights for each prediction during training. On the right, a network with dropout randomly turns off a selection of nodes during training such that only these nodes are updated when backpropagating the gradients. The effect is that the network is effectively an ensemble model of slightly different networks that share many of the same weights [Srivastava et al. \(2014\)](#).

However, at the time of testing all nodes are used (See Figure 20) and the network effectively regularizes the output by acting as a set of ensemble models (that share some weights).

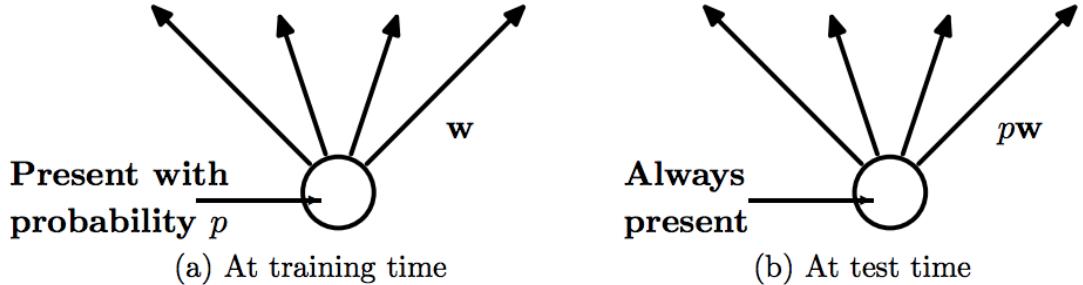


Figure 20: While a node may be randomly turned off with probability p during training, it is active when using the model during testing (or production) [Srivastava et al. \(2014\)](#).

Dropout has been shown to be extremely effective and its use is widespread. In the original paper by [Srivastava et al. \(2014\)](#) a large variety of classification tasks were considerably improved by using the technique - further improving the state-of-the-art of classification. It was used as part

of the chromosomes (designs of neural network architectures introduced in Section 2.5) designed by Dufourq and Bassett (2017) but was not used during this thesis experiment for two reasons:

- It would have added to the already large and complex hyper-parameter search space.
- It has been noted to not be as helpful in many reinforcement learning tasks because of the consistency of the environments - there is little variance between training and test images. However, positive results have been found by Zoph and Le (2016), as well as by Gal et al. (2017), and so future versions of the dissertation experiment may benefit from adding dropout as a part of the tuneable architecture.

Gradient clipping is another technique often used in practice to help remedy the problem of exploding gradients - when the gradient applied to a parameter becomes too large and results in an extremely influential and deterministic outcome on the model (Kanai et al., 2017). While the problem is often solved with the gradient descent optimization techniques discussed in Section 2.3.7, it is sometimes not sufficient in practice. A technique sometimes used to remedy this is gradient clipping - the L^2 norm of the gradients is scaled down to a set number as a way to ensure no update is particularly large. The simplest way to represent this is how it is performed by the Tensorflow function `tf.clip_by_norm()` in Equation 30. The original tensor of gradients, t , is adjusted by normalising relative to a set number, $clip_norm$, and the norm of the tensor itself.

$$t = t \times \frac{clip_norm}{L2_norm(t)} \quad (30)$$

The method proved successful at being able to prevent a complete collapse of the RL agent in initial experiments set out in Appendix A. However, it was not used as part of the final experiment given that Adam was useful enough without needing further regularization of the gradients.

2.3.9 Activation functions

Activation functions create non-linearities in neural networks that ensure that they differ from linear functions. In a typical neural network layer, the inputs x are multiplied by the weights w (and a bias is typically added - b) to form a value z for each node. It is the value z that the activation $g(\cdot)$ is applied to, shown mathematically in Equation 31, and visually with Figure 21.

$$\begin{aligned} z &= xw^T + b \\ y &= g(z) \end{aligned} \tag{31}$$

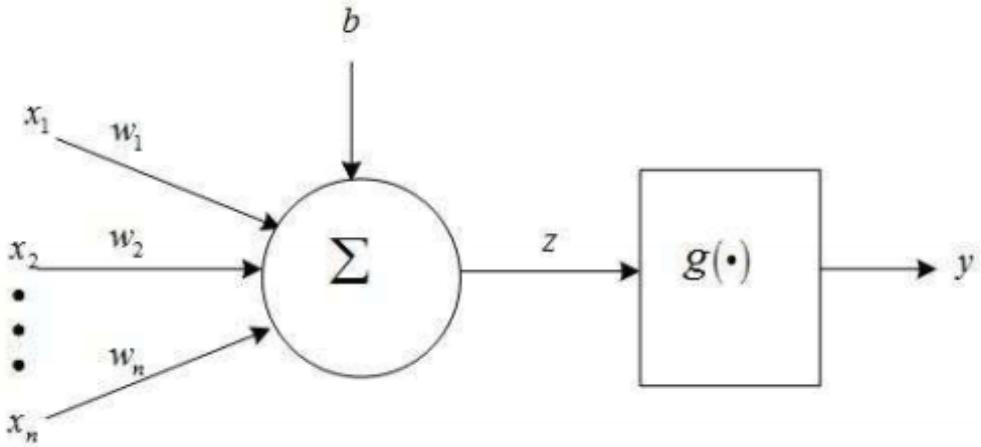


Figure 21: A visual schematic of the activation function. The inputs are multiplied by the weights to form the z value for each node. The z value is then the input for the activation function $g(z)$ (Ding et al., 2018)

Non-linear activation functions ensure that $y \neq z$ for all z , and they are practically differentiable in order to allow for gradient calculation in backpropagation. There are many popular activation functions commonly used in neural networks, but the dissertation only considered the ones described here.

One of the most commonly used functions is the sigmoid, shown in Equation 32 (Ding et al., 2018). While it is easy to differentiate and widely used in early neural networks, it has been known to contribute towards the vanishing gradient problem where many gradients tend towards 0 in many-layered networks.

$$g(x) = \frac{1}{1 + e^{-x}} \tag{32}$$

The softmax function in Equation 33 is similar to the sigmoid in practice when the outcome is

binary, and ensures the sum of all outputs is equal to 1 [Hastie et al. \(2009\)](#). It is commonly used as the last layer in classification networks with multiple classes. Here T_l is the input to the activation function from layer l , but could as easily be replaced by x for simplicity.

$$g_k(T) = \frac{e^{T_k}}{\sum_{l=1}^K e^{T_l}} \quad (33)$$

The Rectified Linear Unit (ReLU) function in Equation 34 (introduced by Geoffrey Hinton) was inspired by neuroscience research that observed that not all neurons are activated simultaneously ([Ding et al., 2018](#)). ReLU ensures that many of the units in a network are effectively “off” half of the time. It is the most popular activation function in deep neural networks.

$$g(x) = \max(0, x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \quad (34)$$

The Leaky ReLU function in Equation 35 is a slight tweak to the classic ReLU function that allows for small non-zero gradients in situations where ReLU does not ([Ding et al., 2018](#)). It has been shown to provide more robust results in many scenarios over standard ReLU.

$$g(x) = \max(0, x) = \begin{cases} x & \text{if } x \geq 0 \\ 0.01x & \text{if } x < 0 \end{cases} \quad (35)$$

The Exponential Linear Unit (ELU) shown in Equation 36 helps solve the vanishing gradient problem of the sigmoid, and also allows small gradients for negative values of z ([Ding et al., 2018](#)). It has been shown to be more robust to noise, and more successful in many tasks, than both ReLU and Leaky ReLU. It was chosen as the standard activation function in the reference model used during the thesis experiment, designed by [Blackwell et al. \(2018\)](#). Here a is a hyper-parameter to be tuned.

$$g(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0 \\ a(e^x - 1) & \text{if } x \leq 0 \end{cases} \quad (36)$$

2.3.10 Convolutional layers

Convolutional neural networks (CNNs) are generally used for processing 2-dimensional data (typically, images). They are a simpler form of matrix multiplication than that previously described in Equation 31. Convolutional layers may include other layers in addition to the actual convolution, such as pooling layers - the term is often used generally for this category of layers in neural networks.

They have been widely and successfully used, with some of the first impressive results with [LeCun et al. \(1989\)](#), and then [Lecun et al. \(1998\)](#) where the technology was used commercially (the network was known as LeNet-5). [Krizhevsky et al. \(2012\)](#) won the famous ImageNet competition with their network that became known as AlexNet. This marked a point of considerable acceleration in the development of high performing CNNs as the improvement and availability of GPU processors allowed considerably more efficient calculations for matrix operations.

In RL, ([Mnih et al., 2015](#)) makes use of a convolutional neural network (CNN) with two convolutional layers, and two fully connected layers to achieve superhuman performance on Atari games. CNNs are also described by [Sutton and Barto \(2018\)](#) as having achieved impressive results for RL in general.

The mathematical form of the convolution operation itself is shown in Equation 37. $S(i, j)$ represents the resultant value in a matrix after applying the convolution operation. I is the input matrix, and K the filter applied to the input. Multiple filters are contained within CNN layers, and they end up representing specific patterns within an image.

$$S(i, j) = (I * K)(i, j) = \sum_m \sum_n I(m, n) K(i - m, j - n) \quad (37)$$

A more tangible example of the operation is shown in Figure 22. Here the summation of each resultant index in the matrix can be seen as the convolution of the input and the filter (kernel).

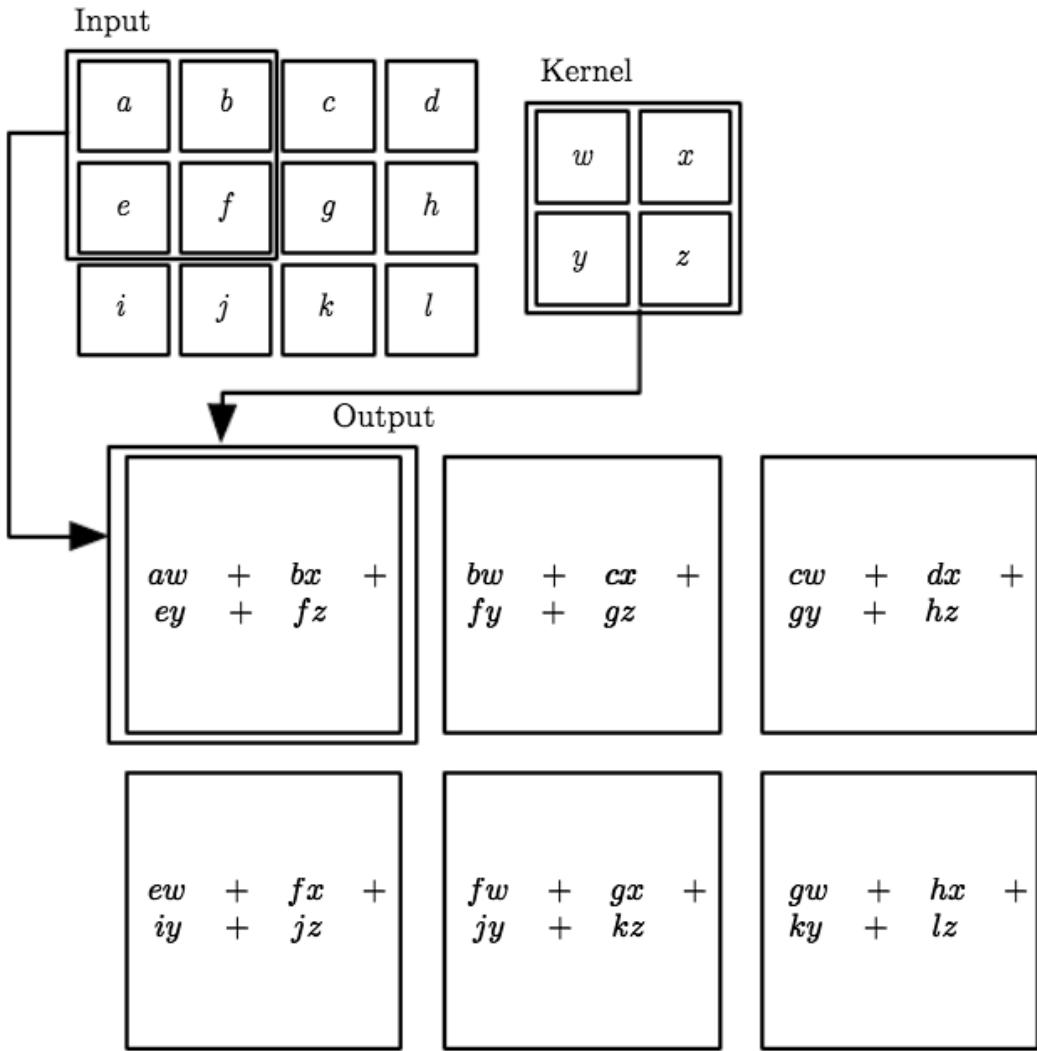


Figure 22: A convolution operation. Each resultant cell in the output matrix is the convolution of an input image, and a filter (kernel) (Goodfellow et al., 2016).

Broadly speaking, CNNs are good at recognising classes of images. Whether the class be “cat” vs. “dog”, or in RL a specific action such as “up” / “down” / “left” / “right” that corresponds to the state of an environment represented by an image of the environment. Figure 23 shows this in high level terms.

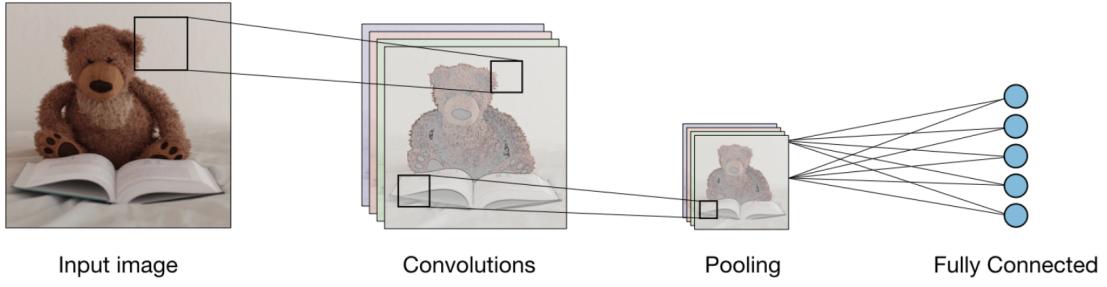


Figure 23: A high level overview of a CNN. An image of a stuffed bear goes through two CNN layers (convolution, and pooling) before passing through a fully connected layer with a softmax activation function. The result is a likelihood of the image belonging to one of the classes associated with the nodes of the fully connected layer ([Amidi and Amidi, 2018](#)).

Figure 24 shows a filter with parameters (blue) being convolved with a same sized part of an input image (red) resulting in the output (as shown in Figure 22) in a cell of the output matrix (purple).

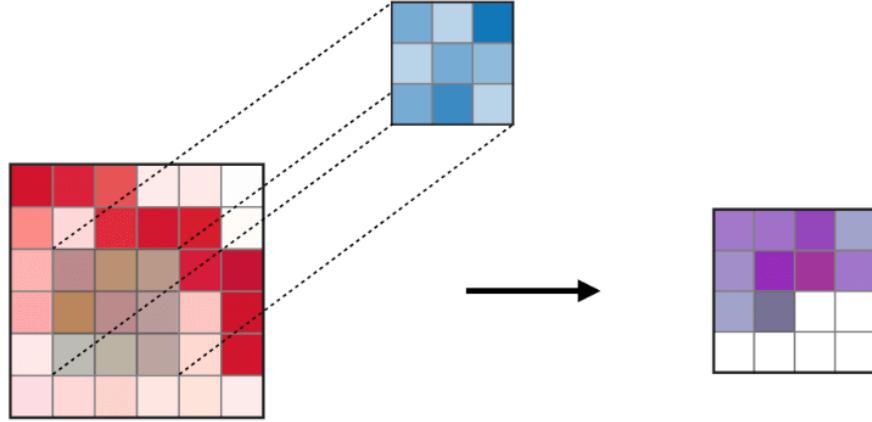


Figure 24: A filter being convolved with an input image, resulting in an output matrix ([Amidi and Amidi, 2018](#))

Another type of CNN layer is the pooling layer. These layers attempt to remove less useful information from an image in order to reduce the number of parameters in a neural network, and reduce the size of later layers in order to allow a fully connected layer (or multiple fully connected layers) to adequately predict classes. Two types of pooling are shown in Figure 25 - max pooling, and average pooling. Max pooling takes the highest valued cell in a specifically sized area of the input and uses that for the output matrix. Average pooling takes the average value of the cells that

are in the input matrix and uses that value. Max pooling was the pooling layer type used in the dissertation experiment, and is the most commonly used in practice.

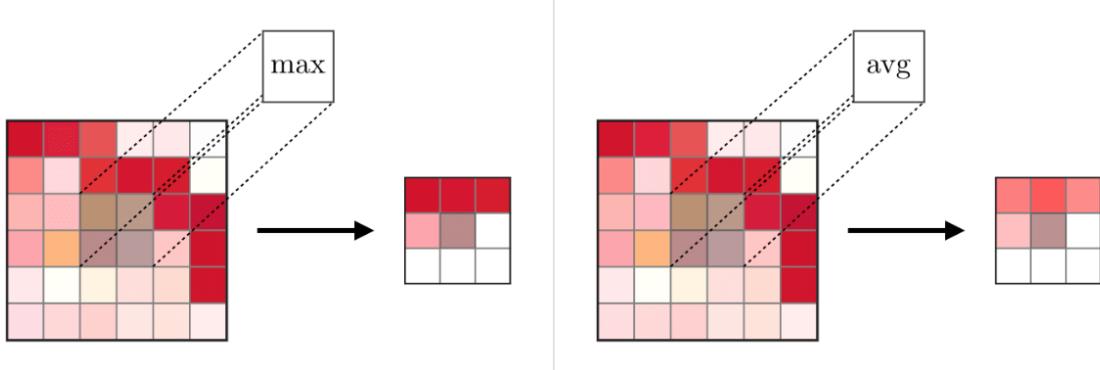


Figure 25: Max and average pooling both reduce the information from a specified part of the input into a single number in the output, relative to the size of the max pooling kernel and the step size (explained below) of the operation ([Amidi and Amidi, 2018](#)).

Figure 26 shows the fully connected layers mentioned previously. This layer connects to the last CNN layer after it has been “flattened” (converted into a 1-dimensional vector). Multiple fully connected layers may be connected in a row, but they ultimately lead to a final layer with some kind of softmax activation (or a linear activation in some cases, such as with RL value functions).

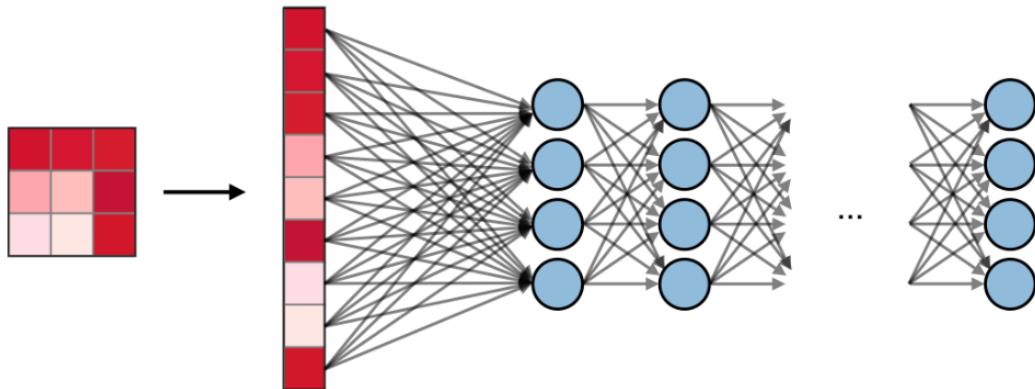


Figure 26: A fully connected layer takes as input the last CNN layer after it has been flattened into a 1-dimensional vector. There may be many fully connected layers in a row, leading to the final output layer ([Amidi and Amidi, 2018](#)).

Filters (sometimes called kernels) are the parts of a CNN that contain the parameters updated by gradient descent. They contain three dimensions: a length, width, and number of channels.

In 2-dimensional cases the number of channels is equal to 1. There may be many filters for each convolutional layer, although earlier CNN layers typically have less filters than later layers due to the more limited numbers of low-level features (i.e. a vertical edge detected in an earlier layer, versus some complex feature in a human face in later layers).

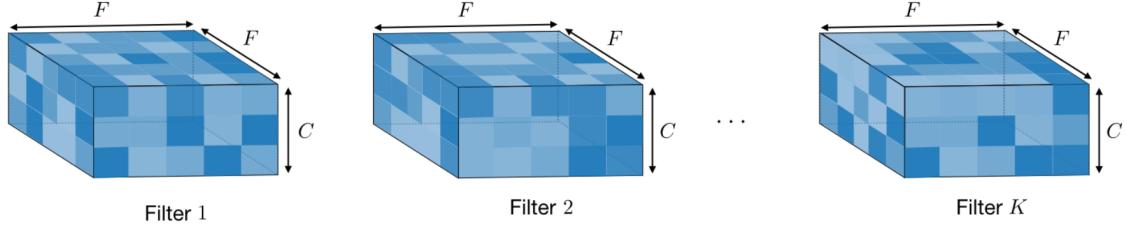


Figure 27: Filters contain the trainable parameters within CNNs. They have a length, width and number of channels. With 2-dimensional images, the number of channels is 1 ([Amidi and Amidi, 2018](#)).

When moving any CNN layer across the input data, the step size (or stride) is an important factor. The larger the step size, the smaller the output of the CNN layer, and hence the less information conveyed at the higher layer. It has been shown that in many cases a larger step size can even substitute for the need to include pooling layers ([Springenber et al., 2014](#)). The concept of step size is shown in Figure 28.



Figure 28: An illustration of step size (stride) on a 1-dimensional vector. Here a filter of size 3 is moved with a stride of 2 across the vector ([Amidi and Amidi, 2018](#)).

One final aspect of CNNs is how to ensure all input data is processed, and how to maintain reasonable output sizes. In Figure 29 three different strategies are shown. The middle approach is called “same” padding and was used in this dissertation - an extra row / column of 0s is added to the input to ensure the filter is able to move across all the input data. In the left image, “valid” padding is shown - with a stride of 2 here the last column of input data would not be part of any convolutions. The right image shows “full” padding where the last rows and columns have “end convolutions” applied to them.

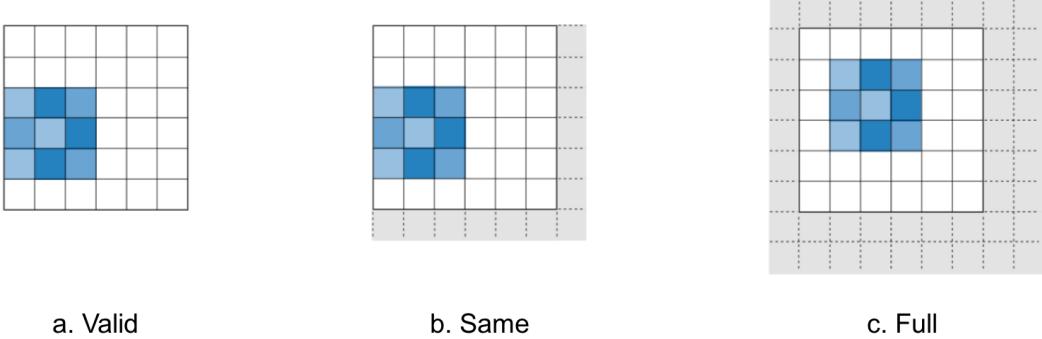


Figure 29: Three different padding strategies, as described above: valid, same, and full. Each strategy is a decision about how much of the input data makes its way into the output - the grey cells represent blank data, usually "0". Full allows the most information through, and valid the least. However, same is very popular ([Amidi and Amidi, 2018](#)).

2.3.11 Recurrent neural networks

Recurrent neural networks (RNNs) are a type of network that use information from previous network outputs as part of their input. These types of neural networks are good at incorporating memory so that information from previous time steps provides context around information at the current time step - therefore, they are good function approximators when a particular task does not have the Markov property (i.e. the current state cannot fully describe all possible future states) ([Hausknecht and Stone, 2015](#)). Long Short-Term Memory (LSTM) architecture is the most prevalent form of recurrent networks as it helps to solve the vanishing gradient problem as partial derivatives back propagate through time in recurrent neural networks (RNNs) ([Hochreiter and Schmidhuber, 1997](#)). [Wierstra et al. \(2009\)](#) found LSTMs to perform well on several benchmark tasks. [Bakker \(2016\)](#) also found LSTMs to perform particularly well on non-Markovian tasks.

The general form of an RNN can be expressed as Equation 38 where the output of a step in the network y_t is a function of the term h_t which in itself is a function of h_{t-1} ([Akandeh and Salem, 2017](#)). The parameter weight matrices are given by W .

$$\begin{aligned} h_t &= \sigma(W_{hx}x_t + W_{hh}h_{t-1} + b_h) \\ y_t &= W_{hy}h_t + b_y \end{aligned} \tag{38}$$

The LSTM network type takes this idea much further and introduces a set of gates within the LSTM node as a way to effectively deal with the vanishing gradient problem described previously. The output layer of the LSTM is expressed as Equation 40, while the inputs to this are shown in Equation 39 - here, as before, h_t is a function of h_{t-1} as well as other inputs from previous steps, and gates of information within the current time step.

$$\begin{aligned} i_t &= \sigma_{in}(W_i x_t + U_i h_{t-1} + b_i) \\ f_t &= \sigma_{in}(W_f x_t + U_f h_{t-1} + b_f) \\ o_t &= \sigma_{in}(W_o x_t + U_o h_{t-1} + b_o) \\ \tilde{c}_t &= \sigma(W_c x_t + U_c h_{t-1} + b_c) \end{aligned} \tag{39}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

$$h_t = o_t \odot \sigma(c_t)$$

$$y_t = W_{hy} h_t + b_y \tag{40}$$

The LSTM network is shown visually in Figure 30. Here the three inputs at each step are shown feeding into the network: c_{t-1} , h_{t-1} , and x_{t-1} from the actual input data. The outputs feed to the next step, but h_t may be used further (as in Equation 40), or as an input to another type of layer or activation function useful as an observed output.

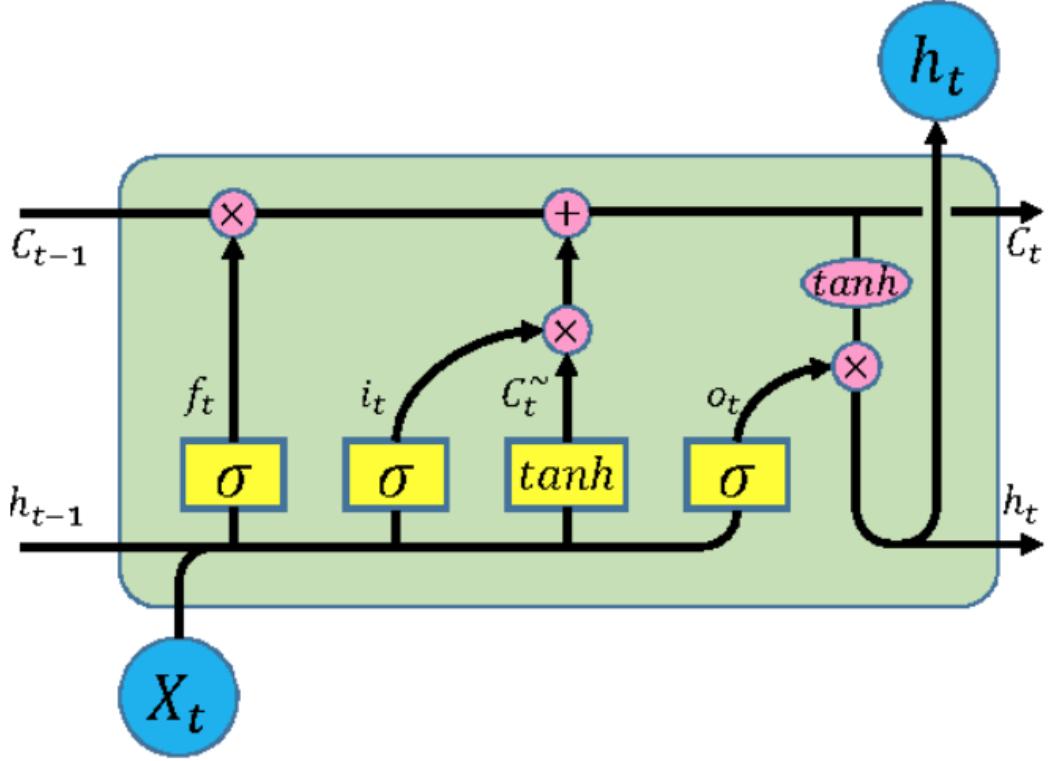


Figure 30: A visual representation of a LSTM cell describe in Equations 39 and 40. The previous cell state c_{t-1} and the previous output h_{t-1} both enter as inputs into the current time step (Sha et al., 2016).

In this dissertation a LSTM network is used as a way to deal with the lack of the Markov property in the Pong task. The image processing network (the neural network architecture being modified in the evolutionary process) inputs a signal into the LSTM layer which then gives outputs for the policy and value functions. The key aspect of LSTMs here is that input is being received from LSTMs at previous time steps which allows the network context (memory) of previous states, enabling it to better deal with the motion within the game environment. This is shown in Figure 31 below.

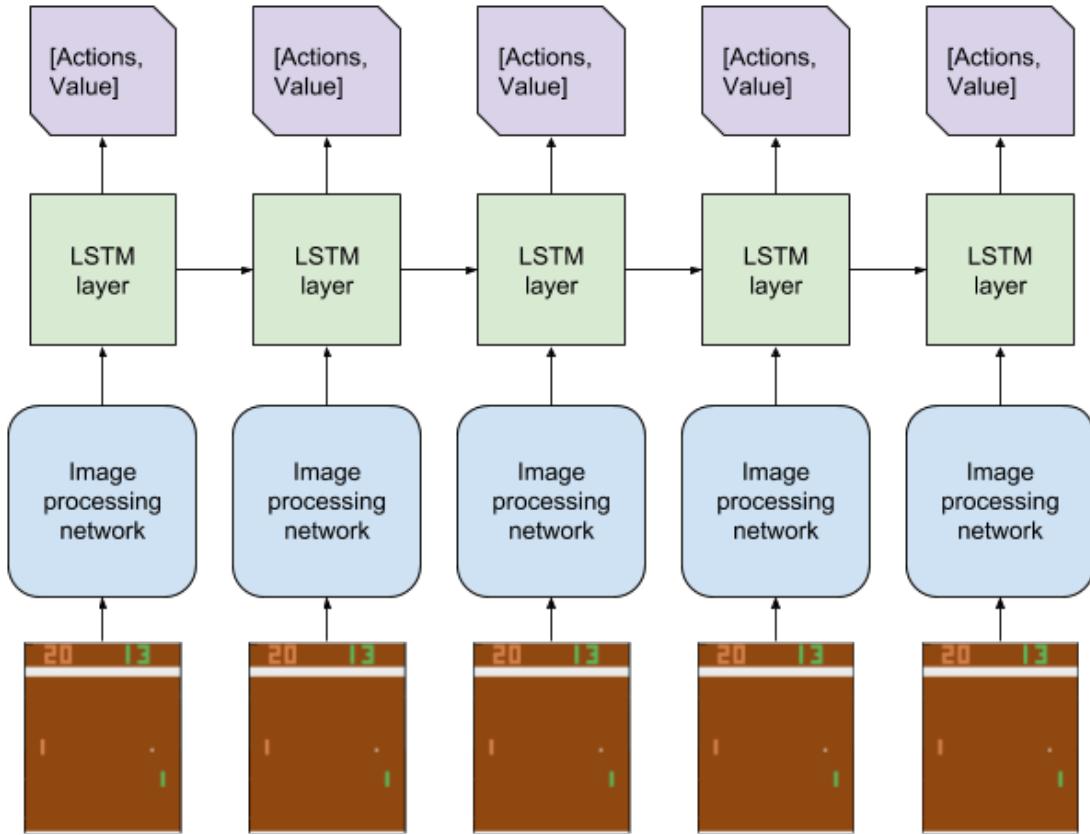


Figure 31: An image processing layer (the chromosome in this dissertation) feeding into LSTM layers before deciding on an action and the values of a vector of actions required for A3C training. Each LSTM layer feeds into the same layer at the next time step as a way to convey context from previous steps, such as the direction of the ball in Pong.

The LSTM layer was effective on this task in multiple cases, such as [Mnih et al. \(2016\)](#). But, more basic approaches have shown some limited success, such as subtracting the previous image from the current one before image processing as a way to capture motion - this was the approach taken by [Karpathy \(2016b\)](#).

2.3.12 Conclusion

Neural networks are composed of multiple and varied parts. This section gave a general overview of neural networks and how they were trained, introducing more complex aspects employed by this dissertation: convolutional networks, recurrent networks, and gradient descent optimisation.

2.4 Parameter searching methods

2.4.1 Introduction

Finding optimal hyper-parameters, and the right architecture, are a crucial aspect of ensuring neural networks work well. Parameter searching methods help find these choices in a systematic and automated way. There are a number of approaches that aim to make specific trade-offs between comprehensively searching through possibilities, versus methods that aim to hone in on a promising direction.

Additionally, there are methods that rely on past experience of similar tasks that aim to define model hyper-parameters that are likely to be good.

[Claesen and De Moor \(2015\)](#) note that the problem of finding good hyper-parameters becomes increasingly difficult when the number of hyper-parameters is large. It follows that the problem is further complicated by the size of the range hyper-parameters can take values from. Automated approaches are thus a necessary method of dealing with the problem.

[Claesen and De Moor \(2015\)](#) show that the problem of hyper-parameter search can be described by Equation 41 where λ^* is the optimal set of hyper-parameters such that the model \mathcal{M}^* minimizes the loss on the test data after training given by $\mathcal{L}(X^{te}; \mathcal{M})$ - this is the same as minimising the loss function $J(\theta)$ in Section 2.3.3. \mathcal{A} is the learning algorithm and \mathcal{L} the loss function. X^{tr} and X^{te} represent the training and test sets, respectively, on a supervised task. The formulation is conceptually applicable to reinforcement learning where we have actions that lead to superior results, or the final scoring of the agent, represented by the test data, and actual exploratory actions during training represented by the training data.

$$\lambda^* = \underset{\lambda}{\operatorname{argmin}} \mathcal{L}(X^{te}; \mathcal{A}(X^{tr}; \lambda)) = \underset{\lambda}{\operatorname{argmin}} \mathcal{F}(\lambda; \mathcal{A}, X^{tr}, X^{te}, \mathcal{L}) \quad (41)$$

Once we have this formulation, [Claesen and De Moor \(2015\)](#) go on to show that there are three challenges faced in searching for the hyper-parameters:

- **Costly objective function evaluation** - The amount of time taken, or the resources required, to train a model enough to evaluate against some test criteria may be high. In this dissertation it was particularly high at 12 hours per model using an expensive AWS EC2

instance.

- **Randomness** - Due to a number of random components involved in the training process, the hyper-parameters λ^* may not be the true optimal set. For example, in this experiment there are a number of random / stochastic components including: Initial weight optimization; the Pong game environment; the order of games.
- **Complex search space** - There can be challenges related to the number of hyper-parameters, the mixture of continuous and integer parameters, and hyper-parameters conditional on others. In this dissertation all three of these search space challenges are present.

The section aims to outline a number of popular search methods, including a broad overview of evolutionary methods. The specific evolutionary method used in this dissertation (genetic algorithm approach) is introduced and described in more detail in Section 2.5.

2.4.2 Grid search

[Bergstra and Bengio \(2012\)](#) note that Grid search is the most widely used strategy for hyper-parameter search. The reason it persists is because it is simple to implement and easy to parallelize given that different hyper-parameters of different models do not depend on each other.

In grid search there is a set Λ that is indexed by K hyper-parameters variables. $(L^{(1)} \dots L^{(K)})$ is a set of values for each variable, so $\Lambda \supseteq (L^{(1)} \dots L^{(K)})$. When performing the grid search, every possible value is trialled - the number of trials is of the size S given by Equation 42:

$$S = \prod_{k=1}^K |L^{(k)}| \quad (42)$$

The resultant number of trials can explode easily when both K and each set of values $L^{(k)}$ is large - known as the *curse of dimensionality*.

2.4.3 Random search

An improved searching method over grid search is random search - shown to be better at finding good models, in less time, for neural network models ([Bergstra and Bengio, 2012](#)).

Random search chooses a number of hyper-parameter sets equal to S trials - each set chosen by a uniform density function over the same configuration space as grid search, resulting in $(\lambda^{(1)} \dots \lambda^{(S)})$.

Random search maintains many of the advantages of grid search, such as an ease of implementation, and the ability to parallelize the search process. However, the advantages are that it is far more efficient in high-dimensional spaces because some dimensions can be considered unimportant in terms of overall performance. Models with high-dimensionality may be considered to have “low effective dimensionality” and therefore don’t require as granular a search across some dimensions. Random search helps to avoid the wasted effort of searching many values in an unimportant subspace. Figure 32 demonstrates this point well - in grid search optimal points on an important dimension are not well covered, but too many points on an unimportant dimension are. The right side of the figure shows how random search covers more points across the important dimension with the same amount of total trials.

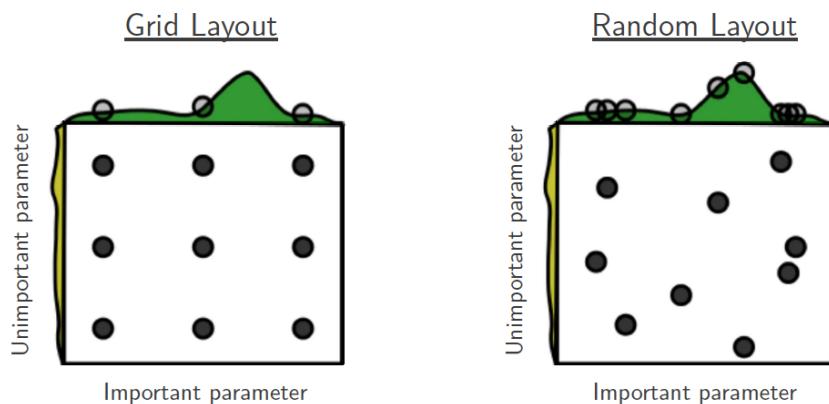


Figure 32: Illustration of random search compared to grid search showing how random search is able to find more optimal solutions with the same number of trials ([Bergstra and Bengio, 2012](#)).

Further practical advantages for random search over grid search noted by [Bergstra and Bengio \(2012\)](#) include that the experiment can be stopped at any point and still be considered complete (due to uniform sampling from the search space); easy to extend parallel experiments without needing to assign computational resources to specific parts of the search area; trials can be asynchronous (i.e. do not need to wait for other trials to end, because of the independence of each experiment); a failure on one trial can be abandoned without affecting the overall experiment.

The experiment in this dissertation makes use of many aspects of random search in the first generation of chromosomes tested.

2.4.4 Other methods

Other popular search methods include Bayesian optimization, and gradient-based optimization. Neither will be explained in detail here, however there are a few observations that are helpful for understanding this dissertation’s methodology.

Bayesian optimization takes random search a step further: using a probabilized view of the search space, then updating it after successive trials. The probability of different parts of the search space are updated in order to encourage more sampling from those areas in new trials. In essence, it tries to hone in on promising areas on more important dimensions. Versions of Bayesian optimization have been shown to be superior to random search, by building on previous trials ([Snoek et al., 2012](#)) - there are similarities here to evolutionary approaches in that more recent trials have learned from the experience of earlier ones.

Gradient-based optimization aims to apply a gradient descent approach to the hyper-parameters themselves (hyper-gradients). [Maclaurin et al. \(2015\)](#) show that this approach is possible by “chaining derivatives back through the entire training procedure” - a very memory-intensive process that poses clear challenges for RL tasks. However, they were able to successfully optimize many traditional hyper-parameters such as step size, weight initialization distributions, regularization methods, and even neural network architectures.

2.4.5 Auto ML

There are a number of other approaches that combine aspects of the previous search strategies, as well as evolutionary algorithms. These approaches are often available as accessible packages in popular programming languages and frameworks for machine learning. Some examples of prominent AutoML:

- **Auto-WEKA** - Using a mostly Bayesian optimization approach is able to optimise several different hyper-parameters depending on the machine learning approach (not just neural networks) ([Kotthoff et al., 2016](#)).
- **Auto-sklearn** - Takes into account past performance on similar datasets, then constructs ensemble models from the various models used in the task evaluation. It has been successful, winning the “ChaLearn AutoML challenge” ([Feurer et al., 2015](#)).

- **Auto-Keras** - [Jin et al. \(2018\)](#) use a combination of Bayesian optimization and other methods to demonstrate state-of-the-art performance for neural network model configuration. Amongst other results, they achieved a 0.55% error rate on the MNIST task ([LeCun and Cortes, 2010](#)), whereas pure Bayesian optimization only achieved a 1.83% classification error.

2.4.6 Conclusions

All of the previously described hyper-parameter optimization methods aim to address the three key problems introduced at the beginning of this section: Costly evaluation; randomness; complex search space. All focus on the cost of evaluation, and how to handle a complex search space. The problem of randomness is largely ignored, but is addressed in this dissertation using a genetic algorithm allowing for models to reappear in multiple generations. The majority of the work is focused on supervised learning tasks with very little literature covering reinforcement learning from a hyper-parameter optimization perspective.

2.5 Genetic algorithms

2.5.1 Introduction

An *evolutionary algorithm* (EA) is an approach to optimization inspired by natural selection. The basic premise is that chromosomes (which in this dissertation are neural networks) are sampled from the search space of possible hyper-parameters and structures, then evaluated by some fitness metric in successive generations, resulting in a population that evolves to perform well on a given task, or set of tasks.

According to [Dufourq \(2018\)](#), *genetic algorithms* (GAs) are a sub-type of EAs that rely on a “survival of the fittest” approach, where each chromosome comprises multiple genes. These genes are edited via mutation in new generations of populations, the size of which is pre-defined. They were first introduced in a machine learning context by [Goldberg \(1989\)](#).

This section will give a high-level overview of GAs, with more detail on concepts with critical implications on the dissertation experiment. The EDEN approach ([Dufourq and Bassett, 2017](#)) was the main influence to the approach taken here and will be explained in more detail. Finally, EAs in a RL context will be reviewed.

2.5.2 Overview

A GA requires two main attributes:

- Some way to represent the genes of a chromosome (*genetic representation*)
- Some way to evaluate the fitness of each chromosome (*fitness function*)

The representation can be in many forms such as some string or vector, or a document structure consisting of items for layers and hyper-parameters (as was the case in this dissertation). The fitness function is elaborated on in Section 2.5.3. The overall evolutionary process is then carried out as follows:

1. **Initialise** an initial population of chromosomes.
2. **Evaluate** the population with some fitness function.
3. **Select** parents based on fitness.
4. **Mutate** the parents in some way and create a new population.
5. Repeat evaluation, selection, and mutation until some **termination** condition is reached.

The initial population should aim to cover a wide representation of the particular search space relevant to the genes in the task. The larger this initial population, the more space is covered so long as the population is generated from some random distribution as described in Section 2.4.3.

2.5.3 Fitness and selection

In order for a genetic algorithm to evolve better performing chromosomes in later generations, some method of selecting high performing chromosomes needs to be in place. There are two components to this: first there must be some measure of fitness; second there must be a way to select genes based on the fitness.

Depending on the task, a number of measures of fitness could be used. For example, in Dufourq and Bassett (2017) the goal is to achieve good performance on several classification tasks, meaning that accuracy was important. However, accuracy was not to be pursued at any expense, and an additional goal of reducing the number of parameters was also a component of fitness. Therefore fitness was improved by higher accuracy, but penalised for the number of parameters as per the minimization of Equation 43 below:

$$\text{fitness}(Net) = \text{val}_{\text{error}} + \alpha \left(1 - \frac{1}{N_p}\right) \quad (43)$$

where Net is the neural network being evaluated, $\text{val}_{\text{error}}$ is the classification error, α is the weight of the penalization, and N_p is the number of model parameters. Similar approaches have been taken by many others, such as [Idrissi et al. \(2016\)](#) looking for the smallest number of hidden layers, and the smallest number of neurons on each layer, but also the minimization of the overall mean square error.

Actually selecting a parent based on the fitness can be done in a number of ways. [Bickle et al. \(1997\)](#) compares 4 approaches: Tournament selection; truncation selection; ranking selection; exponential ranking selection. [Dufourq and Bassett \(2017\)](#) note that [Jinghui Zhong et al. \(2005\)](#) found Tournament Selection to be successful, and used it as part of their EDEN approach. The tournament selection algorithm (specifically, a deterministic version) is described in Algorithm 3. The process is to choose a number of random chromosomes then compare the fitness of each, finally returning the chromosome with the best fitness. The process is repeated with a different random selection of chromosomes to compete the number of times required to produce the desired number of parents. Parents can be selected more than once.

Algorithm 3: A deterministic tournament selection algorithm

```

Input : size: the size of the tournament
Output: The winning chromosome used as a parent
begin
    current_best ← Null
    for  $i \leftarrow 1$  to size do do
        random_chromosome ← randomly select a chromosome from the population
        Evaluate random_chromosome
        if fitness of random_chromosome < fitness of current_best then
            | current_best ← random_chromosome
        end
        return current_best
    end
return
```

The size of each tournament is the parameter that controls the trade-off between exploration and exploitation in tournament selection. Higher selection pressure means that fitter chromosomes are more likely to be selected as parents - the exploitation end of the spectrum. Back (1994) show that selection pressure (probability of chromosome i being selected) is given by equation 44:

$$p_i = \lambda^{-q} \cdot ((\lambda - i + 1)^q - (\lambda - i)^q) \quad (44)$$

where q is the size of the tournament and λ is the population size.

2.5.4 Genetic operations

After a defined number of parent chromosomes are selected via the fitness selection method, the new generation is formed.

In order to form the generation a number of genetic operators are used. These operators are applied to the selected parents, and even to offspring, depending on the specific strategy. The goal of applying these genetic operators is to continue to create enough variation in the chromosomes such that the search space continues to be explored, while selection filters out very poorly performing chromosomes so that they do not form the base of future ones. The operators used in this dissertation are:

- **Reproduction:** The parent chromosome and all its genes are copied exactly to the new generation.
- **Addition:** A random additional gene is added to the chromosome at a random point. In the dissertation context, this would be an additional neural network layer. The operator would have to produce a chromosome that could be effectively evaluated - too many convolutional layers may not be mathematically possible given the way it decreases the size of the output.
- **Deletion:** A random gene is deleted from the chromosome - again, this would be a neural network layer. As before, this would need to be a valid deletion so that the network can be evaluated. A neural network with no layers is no longer a neural network.
- **Change:** This operator is slightly more complex as it may operate on an additional gene - the learning rate. With some probability, the learning rate may be changed in line with the

formula set out in Section 3 (See Table 2). Then, one randomly chosen layer may be changed to a randomly generated layer - again, such that the neural network is valid (for example, it is not possible to have a fully connected layer *before* a convolutional layer).

The above operators, coupled with the fitness based selection are what enable the EA to maintain a trade-off between exploration and exploitation. *Exploration* via the initial randomly created genesis population and subsequent mutations, and *exploitation* via the pressure of the selection mechanism (Dufourq, 2018).

In the dissertation experiment all the parents selected were reproduced in the new generation, then each of the parents had a randomly selected genetic operator applied to it once such that the resulting chromosome was added to the new population. These mutated chromosomes were then mutated once more so that each new generation consisted of: $\frac{1}{3}$ selected parents; $\frac{1}{3}$ once mutated parents; $\frac{1}{3}$ twice mutated parents.

There are other strategies for creating new populations from the previous, such as *crossover*. This strategy attempts to combine genetic material from two selected parents. It may be interesting to compare this strategy in future work.

2.5.5 Termination

The genetic algorithm is terminated when some criteria is satisfied. New generations are created with the N population members reproduced and mutated from the previous generation, with all previous generation members (except for the parents) cleared from memory, but with their results saved for later analysis. Some options for deciding when to stop a GA include:

- When one of the chromosomes reaches a particular level of performance defined in terms of a fitness function.
- After a set amount of time.
- After a set amount of generations.
- Some combination of the above conditions, for example: After a certain level of fitness is reached, or after a set number of generations, whichever is sooner.

For this dissertation, the termination condition was based off a set number of generations, which was chosen as a compromise between computation cost and sufficient exploration.

2.5.6 EDEN

Dufourq and Bassett (2017) introduced their version of a GA applied to a number of supervised machine learning tasks - EDEN (Evolutionary DEep Networks). Their approach formed the basis of the GA approach used in this dissertation. Algorithm 4 details the overall EDEN GA process. Once the number of epochs, the population size, and the maximum number of generations have been set, EDEN begins by setting the starting parameters and creating the initial population of chromosomes. Then, while the number of generations is less than the maximum number specified, parents are selected (via some fitness and selection procedure), and offspring are then created using some genetic operators in order to create a new generation's population of chromosomes. The process is repeated until the maximum number of generations occurs, and finally the best performing chromosome (according to some fitness function) in the history of the experiment may be selected.

Algorithm 4: EDEN genetic algorithm

```

Input: epochs: number of neural network epochs
Input: population_size: population size
Input: generation_max: maximum number of GA generations

begin
    generation  $\leftarrow 0
    epochs  $\leftarrow \text{epochs}$ 
    population_size  $\leftarrow \text{population\_size}$ 
    Create an initial population of chromosomes
    Evaluate the initial population
    while generation  $\leq \text{generation\_max}$  do
        if generation  $\neq 0$  then
            epochs  $\leftarrow (\text{epochs} + 1)$ 
            population_size  $\leftarrow (\text{population\_size} - 10)$ 
        end
        Select the parents
        Create offspring using the genetic operators
        Replace the current offspring with the new offspring created
        Evaluate the current population
        generation  $\leftarrow (\text{generation} + 1)$ 
    end
    return The best chromosome
end$ 
```

In Dufourq and Bassett (2017) the tasks used in the experiment were not RL tasks, but text and image classification tasks. They achieved new state-of-the-art results on three image classification data sets: EMNIST-balanced; EMNIST-digits; Fashion-MNIST. Performance was still good on the other image classification, and text classification tasks. Additionally, due to the parameter penalty in the fitness function (Equation 43), many of the networks created had fewer parameters than some of the existing state-of-the-art networks.

The networks created in Dufourq and Bassett (2017) were able to take on similar layers to that used in this dissertation, with a few notable exceptions:

- Dropout was not used in this dissertation, for reasons described in Section [2.3.8](#).
- 1-dimensional CNNs were not used in this dissertation because of their particular benefit for text classification, which is too dissimilar to the RL Pong task here.
- Related to the above point, there was no need for embedding layers in this dissertation's RL task.
- The ELU activation function (Equation [36](#)) was included in this dissertation considering the good results it has demonstrated for others such as [Blackwell et al. \(2018\)](#).
- Given the relatively lower cost of fitness evaluation in the classification tasks of [Dufourq and Bassett \(2017\)](#) as compared to the RL task here, this dissertation used a more limited search space. For example, where [Dufourq and Bassett \(2017\)](#) allow for choosing the number of filters in a layer between 10 and 100, this dissertation limited the space to 24 to 48.

Finally, a key difference to the EDEN approach shown in Algorithm [4](#) compared to this dissertation, was the removal of the concept of an *epoch*. This is not entirely relevant in a RL context where the learning is more active, and new observations are higher quality data than older observations (generally).

The EDEN approach was used in this dissertation's methodology primarily due to the promising results demonstrated by [Dufourq and Bassett \(2017\)](#), and the ease of implementing the approach in an RL context. A more detailed examination of alternative GA algorithm options may be cause for future research beyond the scope of this work.

2.5.7 Use in reinforcement learning

[Wilson et al. \(2018\)](#) are one of the only existing attempts at using a genetic algorithm approach to evolve effective reinforcement learning agents. They are also able to find a model that effectively plays the game of Pong with a near optimal solution based on episode score, but they make no comment on model robustness, episode length, or time to convergence. Their approach to genetic programming is known as CGP (Cartesian Genetic Programming) that is well described in brief in [Miller \(2014\)](#). In their methodology they explicitly avoid training neural networks, and take an already evolved image processing network that produces the input to the model they're looking to evolve - this allows for exploration of a high number of permutations of last layer-like functions without the

need to actually train a neural network or make any commentary on their design. It makes the assumptions that the image processing network is good enough, which is the opposite approach to that taken in this dissertation. The approach is explicitly instead of A3C or other reinforcement learning approaches, as opposed to in conjunction with them - it was unclear whether any existing reinforcement learning approaches were used when training the image processing network to begin with. Future research could find value in combining the approach in Wilson et al. (2018) and the approach in this dissertation for even more effective full network discovery in reinforcement learning.

2.5.8 Conclusions

This section introduced evolutionary algorithms, with detail on specific elements such as fitness and selection, as well as genetic operators. The EDEN approach was reviewed because it formed the basis of the approach taken in this dissertation. Finally, the limited research in the reinforcement learning domain with EAs was considered, demonstrating the novel nature of the work undertaken in this dissertation.

In terms of the dissertation's design choices, while some of the methods described in Section 2.4 have been shown to successfully address the three primary optimization problems, evolutionary approaches were chosen due to the success demonstrated by Dufourq and Bassett (2017) in image classification and sentiment analysis, and the limited application in the literature within RL.

3 Methodology

3.1 Introduction

This section aims to give an overview of how the experiment was carried out, with details around design, software, and hardware choices. However, the majority of the theory pertaining to the methodology is detailed in Section 2. Each section that follows assumes a working understanding of the key concepts (reinforcement learning; neural networks; hyper-parameter optimisation techniques; evolutionary algorithms), and familiarity with much of the terminology.

Once the general methodology is understood, the summary tables in Section 3.5 should provide a concise view of the design choices made. The full code of the experiment is available for review at <https://github.com/blakecc/evorl> (The author's Github repository).

3.2 High level overview

Figure 33 shows a very high level overview of the experiment setup, but the process is a small variation on the EDEN GA in Algorithm 4 where *epochs* are not relevant (as explained in Section 2.5.6). An initial population of size 12 is generated in line with the parameters set out in Table 1, then the experiment carried out and the results evaluated, after which 4 parents are selected via tournament selection (Section 2.5.3). These parents are then reproduced exactly in the new generation, then mutated once and their mutations added to the new population, then the mutations mutated once so that 4 twice mutated parents are added - this results in 12 chromosomes added to each new generation. The process repeats for 10 new generations.

The size of the populations, and the number of generations were chosen primarily based on the cost of running the experiment. Larger populations, and more generations, would allow for a more widely explored architecture design space which is preferable, and to be considered if the researcher has more resources to do so.

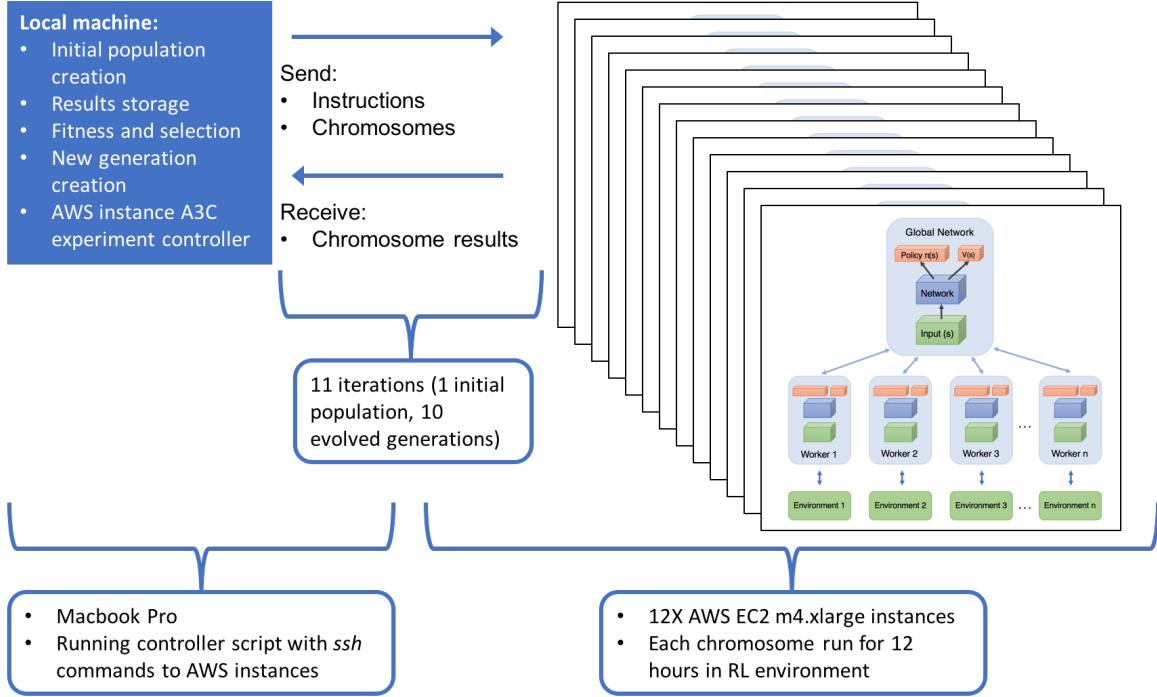


Figure 33: A high level view of the overall experiment. A script running on a local machine controls parallel A3C experiments for each chromosome in a generation. The script ensures the right models are sent to the AWS instances, and that the experiments are started and ended effectively, with their result files (the full episode histories and other information) sent back to the local machine. The local machine then uses these results in order to calculate the fitness of each chromosome, select the parents via tournament selection, and create a new generation with the genetic operators. The new generation chromosomes are then sent to the instances and the cycle repeats until the full genetic algorithm terminates.

Based on some initial experimentation, the time limit of 12 hours was set for each individual chromosome A3C experiment on an AWS instance. The time was chosen after experimenting with the known high performing chromosome from [Blackwell et al. \(2018\)](#) on a m4.xlarge instance - it was long enough to ensure convergence, but short enough to keep AWS costs manageable.

3.3 Task

The task used in this experiment is the Atari game Pong, played within the OpenAI Gym environment - [Brockman et al. \(2016\)](#). Pong is a simple game similar to tennis, or ping pong, where there are two players (each with a paddle) and one ball. The goal of each player is to hit the ball with the paddle so that it cannot be returned by the other player (the ball goes past the other player) - when this is achieved the player scores a single point. The two players play until one achieves a score of

21. In the context of evaluating a player in this environment, it is possible that they can have a net score as high as +21 points (when the other player scores 0 points), or as low as -21 points when the other player scores 21 points (and they score 0). Similarly it is possible to have a net score of 1, or -1, when one player scores 21, and the other 20.

A screen shot of the game is shown in Figure 34. Here the left player is the in-built environment agent, and the right player is the agent we are training. In the current state the in-built environment agent is leading with a net score of +7.

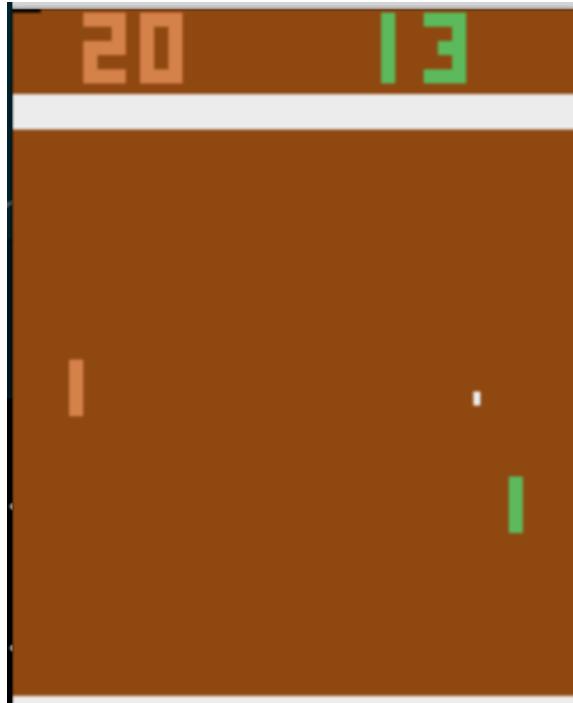


Figure 34: A frame from the Pong environment used in the experiment. The built-in environment AI has 20 points and is 1 point away from winning, while the RL agent (in green on the right) has only 13 points. If the built-in AI scores the next point, the net score for the RL agent will be -8.

Each player has the ability to make three possible moves after each state (frames of the environment): move up; move down; do nothing. For the experiment, the RL agent was not able to choose to do nothing so it only had two choices.

In the image it can be seen that the top part does not constitute part of the game area, or any part of the state (the score is tracked with a variable within the experiment code). This means that it should be possible to remove it from the state with little consequence to the state information. A 2-dimensional monotone version of the original frame has height of 194 pixels, and a width of 160

pixels. By removing the top 34 pixel rows we are left with a 160x160 pixel image. However, this is still 25600 pixels, and it would make the input data simpler if we could scale it down without losing any information. Therefore a down-sampling process is carried out to achieve a 42x42 pixel image which is just 1764 total pixels, with all key information retained - less than 10% of the cropped original image. This was preferred to using additional convolutional layers in order to improve efficiency of the experiment.

A final note regarding the choice of just one task versus experimenting with many: GAs are heuristic optimisation algorithms that are general and not particular to any task. Therefore the expectation is that if it can be shown that a GA is able to find better networks versus a hand designed network on just one task (in this case, Pong), then it indicates the algorithm probably would do so on others.

3.4 Tournament score

The tournament score is used by the tournament selection algorithm to evaluate the fitness of each chromosome. It is the measure of how well the chromosome performed as a policy approximation model on the task. The score includes three aspects of performance:

- **Skill:** The ability of the agent to beat the opposing player as measured by winning margin. However, due to the variance in the model this needs to be measured by some moving average to ensure that the model has a consistent ability to achieve a high winning margin. Hence the EMA (exponential moving average) is calculated for the entire game history. An α of 0.9 was chosen for the EMA (Equation 45), which allowed for the result to sufficiently capture a consistent performance, but also reflect more recent games. This is a bit more aggressive than the α of 0.99 used by [Karpathy \(2016b\)](#) or [Blackwell et al. \(2018\)](#), but allows quicker recognition of convergence.
- **Robustness:** In many cases a model may collapse at various points in training, leading to very poor performance and an inability to recover - such as in Appendix A or Figure 36. For this reason a *noise* metric is introduced which measures the average error between the EMA and the actual game net scores.
- **Efficiency:** While the experiment sought to find networks that performed well on the task, an alternate goal was to find networks that may perform well but also do so with smaller, and

more efficient to train, networks. The way this was measured was with an output metric - the average episode time. The number of parameters was not chosen because of the complex relationship between number of image processing network parameters and input to the LSTM model. A far simpler, and more realistic measure, was simply the average episode time across the entire history.

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot S_{t-1}, & t > 1 \end{cases} \quad (45)$$

These three aspects are then quantified in Equation 46, which uses a similar approach to the Dufourq and Bassett (2017) Equation 43 which includes the penalty terms as denominators below 1. These penalty terms were assigned coefficients of 0.5 (for both β_1 and β_2), although this can be adjusted in future experiments in order to assign more weight to different aspects. Ultimately, the goal was to minimise this tournament score such that some reasonable balance between skill, robustness, and efficiency was achieved.

$$\begin{aligned} \text{Score} = & \\ & 1 - \frac{\max(\text{EMA}_{\alpha=0.9}) + 21}{42} + \\ & \beta_1 \times \left(1 - \frac{1}{1 + \text{Noise}_{\text{avg}}} \right) + \\ & \beta_2 \times \left(1 - \frac{1}{1 + \text{EpisodeTime}_{\text{avg}}} \right) \end{aligned} \quad (46)$$

3.5 Summary of experiment parameters

The following three tables show a condensed summary of the key design parameters involved in the experiment. Each of the parameters affect the results in their own way, and can be altered in future experiments.

Table 1 shows the parameters (selection methods and ranges) for creating each of the members of the first population. *Random range* when there is a probability distribution refers to sampling numbers from the specified probability distribution. *Random range* when there is a specific range

specified refers to sampling from the range of integers with uniform probability. *Random choice* refers to randomly sampling with equal probability from the list of options. *Conditional* refers to when various sampling types and parameters are used depending on some condition of the previous layer.

Table 1: Summary of hyper-parameters used when creating 11 of the first population of chromosomes - the 12th chromosome was a clone of the OpenAI model.

Parameter	Selection method	Range
Learning rate	Random distribution	$\text{Lognormal}(0, 1) \div 10^4$
Optimiser	Fixed	Adam
Number of layers	Random range	[3, 6]
Layer type	Conditional	If previous fully connected, then must be fully connected. If previous convolutional-type and this layer 4th or higher, then 50% probability to add a fully connected layer. Else, select from convolutional layer types.
Convolutional layer types	Random choice	[conv_2d, conv_2d, conv_2d, conv_2d, max_pool_2d, max_pool_2d, fully_conn]
Full connected layer types	Random choice	[fully_conn]
Number of filters	Random range	[24, 48]
Filter size	Fixed	(3, 3)
Max pooling size	Fixed	(2, 2)
Activation of convolutional layers	Random choice	[ELU, ReLU, LeakyReLU]
Number of fully connected units	Random choice	[24, 48]
Activation of fully connected layers	Random choice	[ELU, Sigmoid, Softmax, ReLU]
LSTM layer	Fixed	Yes
LSTM units	Fixed	256
Final layer activation	Fixed	Value function = Linear, Action space = Logit weighted categorical sample of potential actions

Table 2 shows the additional and differing parameters from Table 1 used in mutation. The main difference is the learning rate, which samples from a normal distribution and incorporates the learning rate of the direct ancestor. The *Add* and *Change* mutation operators use the same generating parameters as described in Table 1.

Table 2: Summary of hyper-parameters used when mutating chromosomes after the first generation.

Parameter	Selection method	Range
Learning rate	Random distribution	50% probability of change for any mutation type. When changing $\max(LR_{-1} + \mathcal{N}(0, 1) \times LR_{-1}, 10^{-6})$ where LR_{-1} is the direct ancestor's learning rate
Mutation type	Conditional random choice	If the current number of layers is just 1, then [add, change], else if current number of layers is $>= 6$ then [delete, change], else [add, delete, change]
Delete mutation	Random choice	Delete 1 of any of the layers
Add mutation	Conditional random choice	Add a new layer after any random layer in the chromosome including after an additional “0” layer to allow for adding a new first layer - follow the same rules for adding layers as when creating the first population in Table 1. Possible choices are therefore 1 + Number of layers. Resultant chromosome <i>must</i> be valid - repeat layer generation at the selected layer until a valid chromosome is generated.
Change mutation	Conditional random choice	Change 1 of any of the layers. Follow the same rules for adding layers as when creating the first population in Table 1. Resultant chromosome <i>must</i> be valid - repeat layer generation at the selected layer until a valid chromosome is generated.

Finally, Table 3 shows the overall parameters involved in running the experiment - from the number of workers, to the selection method, to the specific software and hardware choices involved. The table includes the cost of running the experiment, which is an important metric considering one of the main goals of finding efficient ways of creating better architectures is to reduce the amount of resources required. These experiment run parameters likely have less influence on the specific architectures that evolve (with the exception of the selection method and attributes) and pertain mostly to the efficiency, length of time, and cost of the experiment.

Table 3: Summary of experiment run parameters.

Parameter	Description
Number of workers per chromosome	2
Maximum run time per chromosome	12 hours
Maximum number of steps allowed	16 million
Order of experiments	Parallel for each generation
Number of generations	Initial population +10
Generation size	12
Parent selection method	Tournament selection
Parents selected	4
Tournament size	4
Selection criteria	Tournament score - see Equation 46
AWS EC2 Instance Type	M4.xlarge for each chromosome. 4 CPUs each a 2.4GHz Intel Xeon E5-2677 v3, 16 GiB Memory, EBS storage.
Base image used	Deep Learning AMI (Ubuntu) v20.0
Instance launch type	Spot
Compute cost per hour	\$0.0689
Storage cost per GB per month	\$0.11
Total experiment run cost (actual)	\$152.72

3.6 Conclusions

This section gave a high level overview, and then specific parameter details of the dissertation experiment. The theory and logic behind concepts such as genetic algorithms, tournament selection, and reinforcement learning, were detailed previously in Section 2. An even more detailed understanding may be gained by reviewing the Python code developed to run the experiment at <https://github.com/blakecc/evorl>.

4 Results and discussion

4.1 Introduction

The results of the experiment demonstrate that using a genetic algorithm was a reasonable approach to designing policy approximation functions in reinforcement learning. With a limited number of generations, small initial population and subsequent generation sizes, the results are promising considering the possibility of significantly scaling all these aspects.

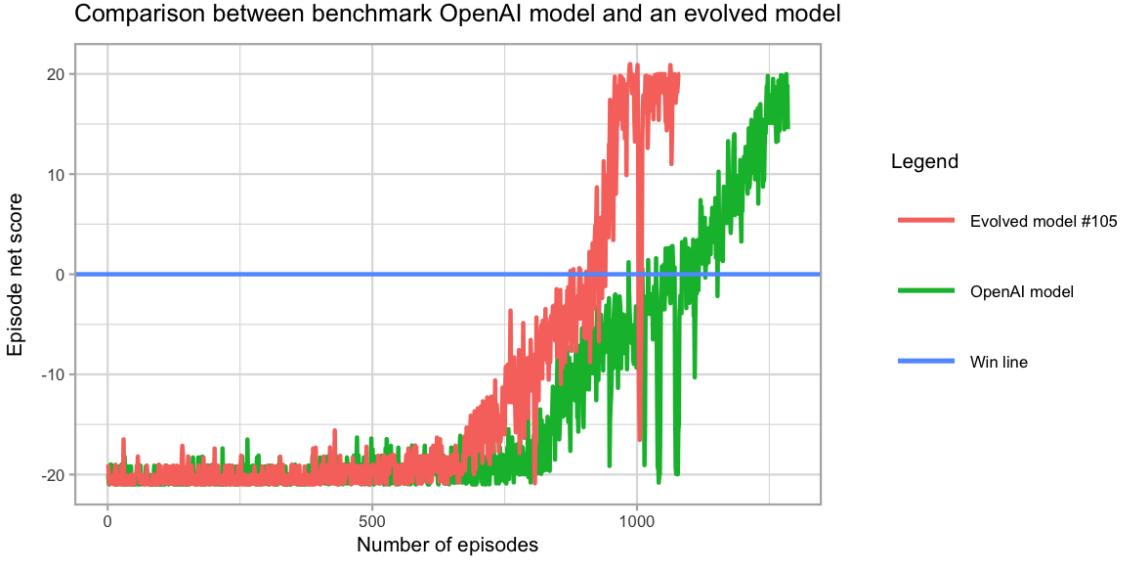


Figure 35: The EMA of net episode score history of two different models - the OpenAI benchmark model from Blackwell et al. (2018), and an evolved model from the experiment (#105 - see Appendix B). The evolved model is seen to be able to achieve a higher EMA than the OpenAI model, as well as being able to converge in significantly fewer episodes. However, as measured by the Tournament Score metric (Equation 46) the two models performed similarly, as the evolved model had a longer average episode time - future work may choose to change the Tournament Score metric in such a way as to select this type of performance more frequently. Note that these are the episode score histories of two individual RL agents training on the task, and it is possible that they are not representative of the average path. However, a number of different runs of these exact architectures occurred during the experiment, and the results of the averages are shown in Figure 37 - it can be seen that architecture of the evolved model performed better on average too, and not only this single comparison.

A sub goal of the research was to find a structure that could outperform the one used by Blackwell et al. (2018). The objective was satisfied in some respects: using the tournament score metric described in Equation 46 individual models were able to perform on par with the best iteration of OpenAI’s model, although with fewer parameters and less time taken to reach perfect scores on the game of Pong.

In many cases, evolved models were also able to show very noticeable improvements over the OpenAI benchmark model on individual performance metrics such as achieving higher average running scores (EMA) and time to convergence (which was not directly measured in the tournament score metric) - Figure 35 demonstrates this with the EMA history of an evolved model compared against the OpenAI model.

The full results will be described from three perspectives:

- *Reinforcement learning*: The learning progress of individual experiments and how training progresses.
- *Evolution*: The progression of genes over generations, and the overall change in population fitness.
- *Neural network design and parameter selection*: An in-depth analysis of all relevant parameters, such as the number of layers, layer types, activation functions and learning rates. General observations of which combinations of parameters achieve superior results are made based on the fitness of individual chromosomes.

4.2 Reinforcement learning

Individual chromosomes achieved various levels of performance during training, with unique training paths on the way to mastery (convergence). Figure 36 illustrates some of the training paths taken. The OpenAI designed chromosome achieves mastery for an individual worker after about 1100 episodes (a set of games where one player wins by scoring 21 points) - seen in figure 36a. A descendent of the original OpenAI chromosome, in figure 36b, achieves mastery in approximately half the number of episodes, but with a much less stable trajectory. Another chromosome, figure 36c, shows signs of improvement but keeps collapsing and never quite achieves full mastery, despite several individual episodes showing convincing wins over the in-built Pong AI. Finally, in figure 36d, a chromosome that shows no discernible progress is shown. The variation in the learning paths further demonstrates the importance of finding consistently good policy approximation models - a poor design choice can hide effective general RL performance for a task, or a RL approach in general, because good performance was not achieved on a particular run.

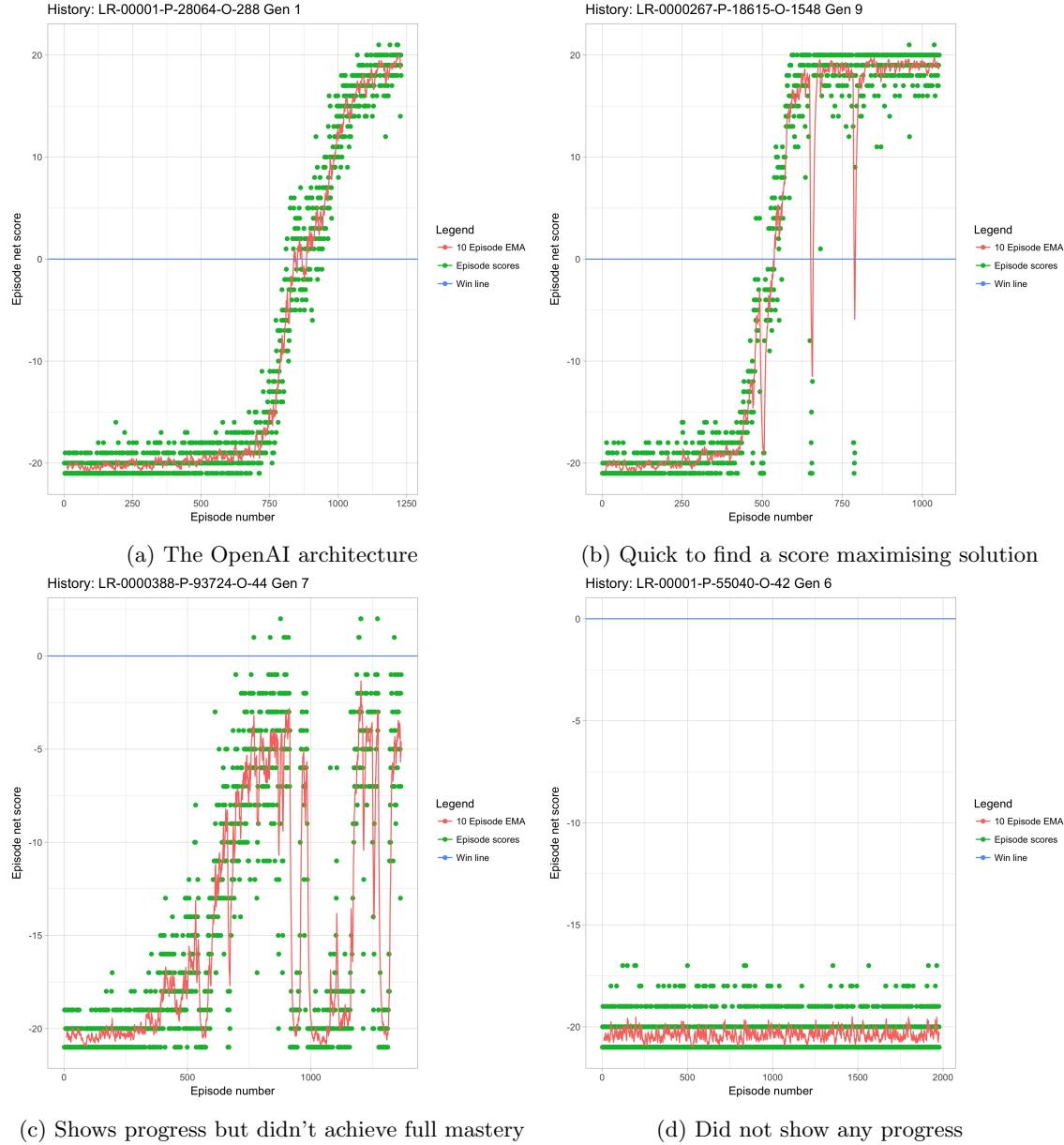


Figure 36: A selection of training histories demonstrating variation in performance of architecture design. Some models showed steady convergence (a), some showed quick but unstable convergence (b), some showed progress but an inability to converge (c), and others no progress at all (d).

4.3 Evolution

Figure 37 shows the full evolutionary process during the experiment. How to read the table:

- Each row represents a unique model architecture, that could have appeared in multiple generations, or even multiple times in the same generation.

- *Count* represents the number of generations the model appeared in, and *Sum* represents the total number of times the model appeared in the experiment.
- In the columns marked *Generations* each of the total 11 generations are represented, where a number in that column for a model indicates how many times that model was present in that generation - the vast majority of the time a model was only present once, but due to the random mutation effects and the chance of the same model being selected twice via tournament selection, there are several instances of a model being present twice in a generation. This increases the survival chances of a model, as well as the survival chances of its offspring.
- The models are ordered in such a way as to make it easy to understand their genealogy. The ordering and shading / numbering in the *Generations* cells is similar to that of a nested list of bullet points. The direct ancestor of each model can be found by finding the first coloured cell one column to the left and upwards. This method corresponds to using the *predecessor* column.
- The training results columns (first 4 columns) are all based off averages for that model architecture across each time they were trained. This means the best figures are often slightly higher than what is displayed here for models that survived for several generations. Bracketed figures represent negative values.
- As a reminder for how each generation is composed: The first 4 models are the winners from the previous generation; the next 4 are one mutation away from each of these; the last 4 are two mutations away from the winners.
- The names of the models were designed to help identify unique structures easily - each comprises information on the learning rate (LR), the overall number of image processing layer parameters (P), and the size of the output (O) of the image processing layer. The full detail on each chromosome can be found in Appendix B.

The first generation (0) population was created by a genesis function using the parameters described in Table 1. The tournament selection method chose just four chromosomes from this generation to be carried over to the next. These four chromosomes each created two children, and therefore their genes are present in three chromosomes each in generation 1. It is these four initially selected chromosomes that are the oldest ancestors of all subsequent chromosomes.

4 RESULTS AND DISCUSSION

4.3 Evolution

Best EMA Avg.	Noise around EMA Avg.	Avg. episode time Avg.	Tournament score Avg.	Mutations	Oldest Ancestor	Predecessor	Model #	Generation	Model	Generation:										Count	Sum
										0	1	2	3	4	5	6	7	8	9	10	
19.50	0.28	34.00	0.63	1	000		000	0	LR-0.0001-P-28064-O-288	1	1	2	1	1	1					6	7
(6.99)	0.16	25.25	1.22	2	000	012	016	1	LR-4.6e-05-P-44236-O-128	1										1	1
(12.35)	0.16	25.81	1.34	3	000	016	020	1	LR-4.6e-05-P-34988-O-288	1										1	1
(10.56)	0.15	26.27	1.30	2	000	024	028	2	LR-4.6e-05-P-29220-O-324	1										1	1
16.95	0.28	30.08	0.69	3	000	028	032	2	LR-4.6e-05-P-18816-O-1152	1										1	1
0.37	0.18	26.58	1.05	2	000	038	042	3	LR-4.6e-05-P-31938-O-288	1										1	1
0.57	0.22	29.17	1.06	3	000	042	046	3	LR-8.4e-05-P-27774-O-288	1										1	1
20.83	0.38	36.19	0.63	2	000	051	055	4	LR-0.0001-P-18816-O-1152	1	2	1	1	1						5	6
20.49	0.29	34.82	0.61	3	000	055	059	4	LR-0.000133-P-9568-O-1152	1										1	1
20.90	0.39	36.44	0.63	3	000	060	064	5	LR-0.0001-P-17660-O-1008	1										1	1
(14.61)	0.14	38.23	1.40	4	000	064	068	5	LR-4.5e-05-P-8412-O-3388	1										1	1
8.14	0.28	23.76	0.89	3	000	075	079	6	LR-0.000119-P-70701-O-45	1	1	1								3	3
20.00	0.34	30.92	0.63	4	000	079	083	6	LR-0.000119-P-91700-O-45	1	1									2	2
10.87	0.36	33.40	0.86	5	000	085	089	7	LR-0.000388-P-93724-O-44	1	1									2	2
20.91	0.52	35.77	0.66	6	000	089	093	7	LR-0.000332-P-79387-O-44	1										1	1
(15.60)	0.10	17.67	1.39	6	000	099	103	8	LR-0.00986-P-84476-O-44	1										1	1
(16.40)	0.10	18.64	1.41	7	000	103	107	8	LR-0.00986-P-91020-O-44	1										1	1
(14.70)	0.12	19.12	1.38	4	000	084	088	7	LR-1e-06-P-183853-O-45	1										1	1
(16.32)	0.11	18.74	1.41	5	000	088	092	7	LR-1e-06-P-185003-O-25	1										1	1
0.53	0.19	26.68	1.05	4	000	097	101	8	LR-0.000105-P-87882-O-45	1										1	1
20.94	0.35	38.85	0.62	5	000	101	105	8	LR-0.000105-P-21417-O-1476	1	1									2	2
20.59	0.49	42.49	0.66	6	000	111	115	9	LR-0.000105-P-21995-O-1548	1	2									2	3
20.90	0.34	41.08	0.62	7	000	115	119	9	LR-0.000267-P-18615-O-1548	1										1	1
8.75	0.24	54.45	0.88	7	000	122	126	10	LR-0.000162-P-9568-O-3872	1										1	1
20.99	0.39	38.61	0.63	8	000	126	130	10	LR-0.00025-P-16508-O-1152	1										1	1
(1.60)	0.19	50.88	1.11	7	000	123	127	10	LR-0.000105-P-9568-O-3872	1										1	1
(16.40)	0.11	41.62	1.43	8	000	127	131	10	LR-1e-06-P-11591-O-4719	1										1	1
(16.40)	0.11	21.35	1.42	3	000	086	090	7	LR-4e-06-P-15354-O-1152	1										1	1
(15.59)	0.11	22.70	1.40	4	000	090	094	7	LR-4e-06-P-14378-O-1152	1										1	1
(11.79)	0.16	50.41	1.34	3	000	096	100	8	LR-0.0001-P-9568-O-3872	1										1	1
15.83	0.34	34.40	0.74	3	000	061	065	5	LR-0.0001-P-9568-O-1152	1										1	1
(14.49)	0.14	41.05	1.39	4	000	065	069	5	LR-0.0001-P-320-O-3872	1	1									2	2
18.90	0.26	28.29	0.64	2	000	063	067	5	LR-0.0001-P-40202-O-168	1	2									2	3
(14.61)	0.12	19.26	1.38	3	000	067	071	5	LR-1e-06-P-30954-O-168	1										1	1
(16.23)	0.11	21.62	1.41	3	000	072	076	6	LR-0.0001-P-48116-O-42	1										1	1
(17.21)	0.11	21.86	1.44	4	000	076	080	6	LR-0.0001-P-55040-O-42	1										1	1
(15.31)	0.12	22.30	1.40	3	000	073	077	6	LR-0.0001-P-36445-O-29	1										1	1
(4.37)	0.16	23.07	1.15	4	000	077	081	6	LR-0.0001-P-43045-O-29	1										1	1
(17.13)	0.11	20.03	1.43	2	000	026	030	2	LR-1e-06-P-31044-O-288	1										1	1
(15.60)	0.12	21.01	1.40	3	000	030	034	2	LR-2e-06-P-37407-O-128	1										1	1
(4.74)	0.17	24.97	1.17	1	001		001	0	LR-0.000156-P-28870-O-44	1	1									2	2
3.43	0.19	26.85	0.98	2	001	015	019	1	LR-0.000232-P-30790-O-44	1										1	1
(11.42)	0.13	23.00	1.31	3	001	019	023	1	LR-9.4e-05-P-22822-O-28	1										1	1
2.09	0.21	31.31	1.02	1	002		002	0	LR-8e-05-P-38170-O-360	1	1									2	2
20.81	0.33	36.03	0.62	2	002	014	018	1	LR-0.000109-P-23730-O-1440	1										1	1
20.83	0.44	38.77	0.64	3	002	018	022	1	LR-0.000109-P-20800-O-1440	1	1	1							3	3	
3.15	0.24	55.21	1.01	4	002	027	031	2	LR-0.000242-P-8160-O-4235	1										1	1
20.95	0.33	35.82	0.61	5	002	031	035	2	LR-0.000242-P-8160-O-1260	1	1									2	2
20.86	0.42	28.92	0.63	6	002	037	041	3	LR-0.000691-P-8160-O-315	1	1	1	1	1	1	1	1		7	7	
17.47	0.42	31.67	0.72	7	002	041	045	3	LR-0.000691-P-8160-O-1260	1	1									3	3
(17.30)	0.08	36.02	1.44	8	002	048	052	4	LR-0.000691-P-8160-O-4235	1										1	1
(15.51)	0.11	18.84	1.39	9	002	052	056	4	LR-1e-06-P-109824-O-24	1										1	1
20.90	0.37	31.84	0.62	7	002	050	054	4	LR-0.000691-P-10760-O-315	1										1	1
20.79	0.39	30.88	0.63	8	002	054	058	4	LR-0.000691-P-22129-O-140	1										1	1
20.80	0.44	29.94	0.64	7	002	062	066	5	LR-0.000691-P-11085-O-315	1										1	1
2.85	0.28	28.02	1.02	8	002	066	070	5	LR-0.000691-P-11085-O-140	1										1	1
(15.60)	0.12	19.41	1.40	7	002	074	078	6	LR-1e-06-P-15220-O-140	1										1	1
(17.21)	0.11	20.51	1.44	8	002	078	082	6	LR-2e-06-P-25415-O-35	1										1	1
(15.85)	0.12	17.53	1.40	7	002	087	091	7	LR-1e-06-P-350-O-315	2										1	2
8.12	0.30	25.38	0.90	7	002	098	102	8	LR-0.000691-P-23012-O-188	1	2									2	3
(16.21)	0.10	18.22	1.41	8	002	102	106	8	LR-0.0001635-P-37852-O-35	1										1	1
(12.66)	0.13	19.51	1.33	8	002	109	113	9	LR-0.0001602-P-10872-O-188	2										1	2
8.46	0.32	26.63	0.90	8	002	110	114	9	LR-0.000471-P-28427-O-188	1	2									2	3
(15.51)	0.11	21.02	1.40	9	002	114	118	9	LR-1e-06-P-42419-O-132	1										1	1
(17.22)	0.11	19.82	1.44	9	002</																

The OpenAI designed chromosome, named “*LR-00001-P-28064-O-288*” (model # 000) and viewable in Appendix B, is seen to have the most genetic successors, while the chromosome with name “*LR-8e-05-P-38170-O-360*” (model # 002) has the next most successors. Both still have successors in the final generation - see Table 6.

The OpenAI chromosome itself has an interesting journey - it remains for five successive generations and is present twice in generation 2, but does not make it to generation 6 (see Table 4). This result is due to the inherent randomness in tournament selection - sometimes a relatively fit chromosome is not picked for a tournament. However, its genes propagate over time and form the basis of even fitter chromosomes. Similarly the chromosome “*LR-0.000691-P-8160-O-315*” (model number 041) lasts for seven generations but does not make it to the last. However, a long set of branches from this chromosome can be seen in Figure 37.

Another interesting aspect from the evolutionary process was the reappearance of dead chromosomes. They are recreated from mutations of successors to themselves - it is not inaccurate to say that these chromosomes may still survive even after dying as they have a probabilistic chance of being included in future generations. It is unknown whether the generation 0 chromosomes that were not selected could have had successors with fitness scores superior to that of the other chromosomes - the tournament selection process may not have been random enough to allow genes with initially poor fitness to propagate long enough to demonstrate this possibility. Future work on this topic could try a combination of several changed hyper-parameters in an attempt to test whether this would happen:

- Larger initial population;
- Larger generation sizes;
- Less chromosomes selected for tournaments (effect of increasing tournament selection randomness);
- Probabilistic tournament wins (a deterministic tournament selection method was used here);
- Other variations in training conditions and environments.

The below tables provide more detail on the evolutionary process.

As previously noted, many of the initial population models did not survive long except for the OpenAI model (000) seen in Table 4. This result makes sense when considering Table 5 as the

OpenAI model had far better performance than the others - their performance was not strong enough to allow subsequent selection.

Table 4: Genesis chromosome survival: The number of times each chromosome from the genesis population occurred in each generation. The eight chromosomes that did not occur after generation 0 were excluded from this table.

Chromosome	Model #	0	1	2	3	4	5	6	7	8	9	10
LR-0.0001-P-28064-O-288	000	1	1	2	1	1	1					
LR-0.000156-P-28870-O-44	001	1	1									
LR-8e-05-P-38170-O-360	002	1	1									
LR-0.000111-P-9516-O-369	011	1	2									

Table 5: Genesis chromosome survival average tournament score: For each of the genesis chromosomes that survived beyond generation 0, the table shows the average score of that chromosome (averaged over multiple runs when relevant). #000 performs consistently well, and therefore survives longer.

Model #	0	1	2	3	4	5	6	7	8	9	10
000	0.63	0.63	0.63	0.63	0.63	0.63	0.63				
001	1.17	1.17									
002	1.02	1.02									
011	1.14	1.14									

However, what is more interesting is the survival of successors to the initial population in Table 6. As previously noted, there were two initial population models that had genes in the final generation. The proportion of these two sets of ancestor successors changes significantly over time, with the OpenAI genes close to not being selected for the 9th generation. Future experiments with larger population sizes could provide more robust insight here by reducing the probability of some of these results occurring due to chance.

Table 6: Successors to oldest ancestors survival over time: The 4 chromosomes in the previous 2 tables are the oldest ancestors of successor chromosomes. This table shows how many successors of each occurred in subsequent generations. #000 and #002 have successors until the final generation, even though #002 itself did not perform nearly as well as #000 as an individual chromosome, but it passed on genes that were successful in survival in later generations.

Oldest ancestor	Model #	0	1	2	3	4	5	6	7	8	9	10
LR-0.0001-P-28064-O-288	000	1	3	6	3	3	9	9	9	9	3	6
LR-0.000156-P-28870-O-44	001	1	3	0	0	0	0	0	0	0	0	0
LR-8e-05-P-38170-O-360	002	1	3	3	6	6	3	3	3	3	9	6
LR-0.000111-P-9516-O-369	011	1	3	3	3	3	0	0	0	0	0	0

An unexpected, but interesting, result regarding the progression of chromosomes can be seen in

Table 7. The average scores do not improve monotonically at each generation. In fact, the best performance for the ancestors of *002* occurred in Generation 1 and 2, while the best performance for the OpenAI ancestors occurred in Generation 4. This is likely explained by the mutated chromosomes in each generation - specific individual mutations of fit chromosomes could drastically deteriorate fitness. It could therefore be hypothesised that a trait of strong genes that survive over time is the robustness to mutations.

Table 7: Successors to oldest ancestors survival over time average tournament score: Similar to Table 5, here we show the average scores of successor chromosomes in each generation. While the successors of these ancestors survive, the average scores do not monotonically improve indicating that genetic survival is about ensuring robustness to mutation such that there is a high chance some successors perform well.

Model #	0	1	2	3	4	5	6	7	8	9	10
000	0.63	1.06	1.01	0.91	0.62	0.89	0.98	1.03	1.06	0.63	0.89
001	1.17	1.15									
002	1.02	0.76	0.76	0.9	0.91	0.77	1.16	1.15	0.98	1.06	0.86
011	1.14	1.04	0.98	1.03	1.22						

In terms of mutations, Table 8 shows the average number of mutations for chromosomes in each generation. There are several interesting observations:

- The OpenAI ancestors do not mutate that frequently at first, probably because early offspring performed well and were selected again.
- The successors to *002* gathered mutations quickly, with the original ancestor not lasting long. The average chromosome in the last generation has undergone 9 mutations - this a high number considering that the maximum number of mutations for any chromosome in the experiment was 10. This resulted, among others, in model “LR-0.000471-P-7835-O-315” (Model # 128) with a tournament score of 0.64 which is among the fittest chromosomes in all generations and required 10 mutations from *002*. Compare the original oldest ancestor in Table 10 to this young model in Table 9. Every layer is different to the original model, and so is the learning rate.
- The average number of mutations actually decreased in successive generations for successors to Model # *002* for Generation 4 to 5, and 6 to 7. This could be influenced by the decrease in the successors to this chromosome in Generation 5, but shows that it is necessary to allow older chromosomes to persist lest their successors not be as fit.

Table 8: Successors to oldest ancestors average number of mutations

Model #	0	1	2	3	4	5	6	7	8	9	10
000	1.00	2.00	2.00	2.00	2.00	2.67	3.00	4.00	4.33	6.00	7.00
001	1.00	2.00									
002	1.00	2.00	4.00	5.00	7.50	7.00	7.00	6.67	7.00	7.56	9.00
011	1.00	1.33	3.00	3.00	3.00						

Table 9: Architecture of model # 128, or “LR-0000471-P-7835-O-315”. Learning rate is 0.000471

	Type	Size	Number	Activation
0	conv_2d	3	24	leaky_relu
1	conv_2d	3	35	elu
2	max_pool_2d	2		
3	max_pool_2d	2		

Table 10: Architecture of model # 002, or “LR-8e-05-P-38170-O-360”. Learning rate is 0.00008

	Type	Size	Number	Activation
0	conv_2d	3	25	leaky_relu
1	conv_2d	3	40	leaky_relu
2	conv_2d	3	40	relu
3	conv_2d	3	40	leaky_relu

Finally, it is worth highlighting the variation in the performance of the same model over time. In Figure 38 the same model’s (OpenAI’s) EMA (exponential moving average) is plotted for all 7 of its runs. Only 6 out of 7 eventually converge on the optimal net score in terms of the game of Pong (Note: not tournament score, which takes net score as an input), and all show different paths to convergence. This shows that it is not good enough to demonstrate fitness once, as it is needed consistently. Since all models are randomly initialized and have different game histories, even those with the same architecture will encounter different conditions on each run - a gene will need to demonstrate fitness over time on different runs in order to persist.

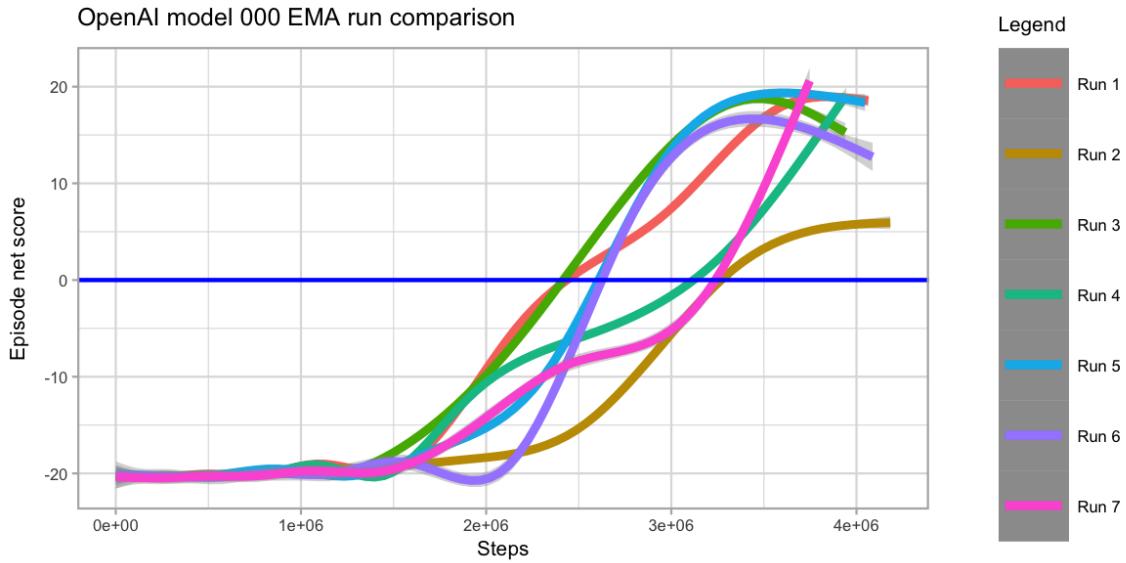


Figure 38: Plot showing comparison of the EMA of 7 different runs of the OpenAI chromosome - model # 000

4.4 Neural network design and parameter selection

A key contribution of the research is to be able to make specific recommendations about what structures to choose for similar kinds of reinforcement learning tasks. The models generated and tested in the experiment are reviewed at a summary level and at a more detailed level in order to give insight about the specific structures.

4.4.1 Summary parameters

At the summary level, there are 4 key metrics:

- “midparams” - The total number of parameters for the chromosome itself. There are more parameters overall when considering the LSTM model that each chromosome feeds into, but since this is the same for all it is not counted.
- “outnum” - The size of the output of the chromosome model architecture. It is the size of the input to the LSTM layer.
- “learnrate” - The learning rate of the model overall, including the LSTM layer.

- “layers” - The number of layers in the chromosome (excluding the LSTM).

Several permutations of multiple linear regression models were tried in order to see if these summary features could be used to predict good model performance. The best of these regressions, as measured by the Akaike Information Criterion (AIC), was that shown in Table 11. The number of layers added no predictive value, but the combination of the log of the remaining 3 metrics, and the square of the OutputNumber had significant predictive value (i.e. less than 2.2^{-16} chance of being random). The regression model structure is described in Equation 47.

Table 11: Multiple linear regression of model feature statistics to Tournament Score

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	1.81	0.34	5.32	0.00
log(midparams)	-0.06	0.02	-2.62	0.01
log(outnum)	-0.17	0.02	-7.63	0.00
I(outnum^2)	0.00	0.00	6.06	0.00
log(learnrate)	-0.08	0.01	-6.34	0.00

$$\text{Score} =$$

$$\begin{aligned}
 & \beta_0 + \\
 & \beta_1 \times \log(\text{ParameterCount}) + \\
 & \beta_2 \times \log(\text{OutputNumber}) + \\
 & \beta_3 \times \text{OutputNumber}^2 + \\
 & \beta_4 \times \log(\text{LearningRate})
 \end{aligned} \tag{47}$$

As a way to visualise the predictive power of the regression, the fitted vs. actual values are plotted in Figure 39. The main observation is that the regression has many false positives (predicting good performance when the actual performance was poor), but almost no false negatives (predicting bad performance when actual performance was good). This is useful for helping to exclude certain architectures, but would be less helpful for guaranteeing good performance. In general, the regression tells us the following:

- Too many parameters decreases performance - likely due to it slowing down the model and increasing episode time (a component of tournament score).

- Learning rates that are too low decrease performance - the coefficient is negative, but this is because learning rates are always less than 0, and the log of a number between 0 and 1 is always negative. The result is likely due to low learning rates taking too long to converge on maxima, or never adequately learning from the very weak signal typical in reinforcement learning.
- Too many, and too few outputs to the LSTM decreases performance. Too few likely means that the LSTM has too little information to learn from, while too many tends to drastically increase episode time. Hence, the regression found predictive value in both a negative coefficient of the log, and a positive coefficient of the square.

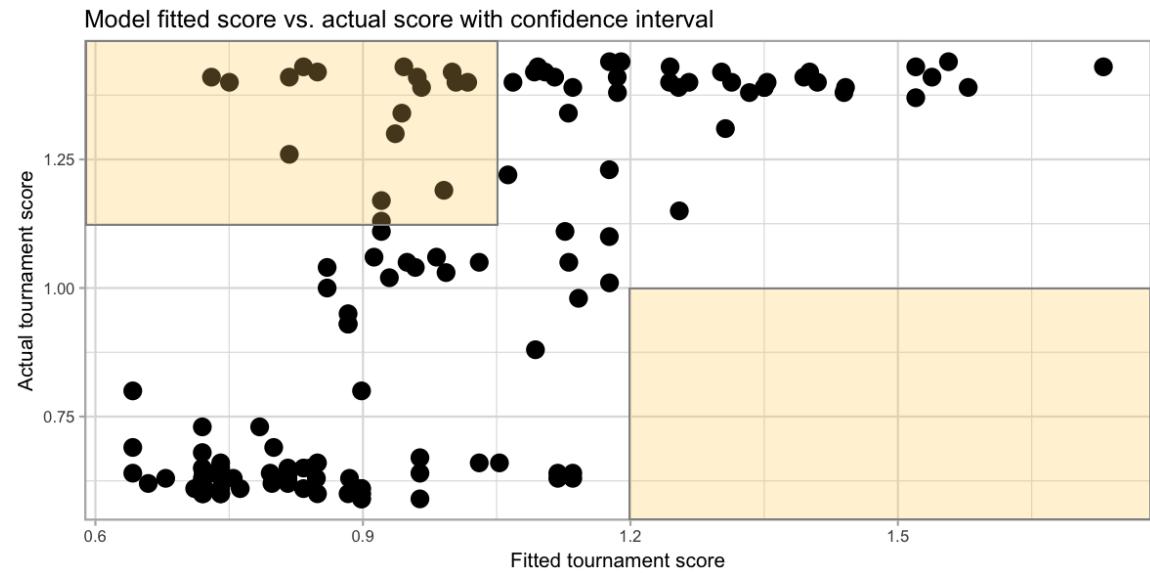


Figure 39: Plot of fitted vs. actual Tournament Score as per regression Equation 47. The R^2 of the model is ≈ 0.46 meaning there is still a significant amount not explained by the regression variables. The lower right quadrant of the chart is empty, indicating that when the model predicts a poor Tournament Score it is unlikely to be wrong - a low rate of false negatives. However, in the top left quadrant there are many examples of the regression predicting a good performing model that actually does not perform well - a high rate of false positives.

4.4.2 Layers

Digging deeper into the architecture and hyper-parameters, we look at the types of layers. As previously noted, the number of layers was not predictive in the context of the other summary metrics, likely because it is correlated with the number of parameters and / or it does not tell us enough about the architecture given the variety of layers. However, looking at Table 12 a fairly equal

distribution across 2 to 5 layers exists in the population over time. In many generations 2 layers did not exist, nor did 6 layers (and did not exist for the final generation). More than 6 layers tended to produce invalid models given the reduction in dimensions from the original input, and this may explain the general limit here - a task with higher input dimensions could have different results.

Table 12: Layer count distribution per generation

Count of layers	0	1	2	3	4	5	6	7	8	9	10
2.00			1	1	1	2			2		3
3.00	2	2	3	4	5	4	1	4	2	4	3
4.00	4	6	6	5	5	3	3	6	3	2	2
5.00	3	1	2	2	1	3	5	2	5	6	4
6.00	3	3				3					

Looking at the type of layers gives more insight. First, the distribution of fully connected layers is shown in Table 13. While the number of these layers was as high as 5 in the initial population, by the last generation (and some in between) there are no fully connected layers in any of the chromosomes. Models with fully connected layers often had far more parameters than those with only convolutional layers, and would have much smaller outputs to the LSTM layer. This may have been a misbalance between overly complex image processing parameters, and not complex enough input to the LSTM layer.

The distribution of convolutional 2D (Conv2D) layers is shown in Table 14. For all chromosomes since the initial population, at least one Conv2D layer was present demonstrating its importance. By far the most common number of this layer type was 3. Mathematically, this would correspond well with reducing the number of parameters, but also retaining a reasonably sized input to the LSTM - previously observed to be predictive of good models.

Lastly, the distribution of max pooling 2D (Pool2D) layers is shown in Table 15. There are many instances of there being no max pooling layers at all, with this being the mode for all but one generation. This is not to say that max pooling was ineffective, as we will see in some specific model examples. Pool2D layers persisted across all generations, unlike the fully connected layers. However, as [Springenberg et al. \(2014\)](#) noted, it is possible to do away with max pooling layers by increasing the stride of convolutional layers or even adding more convolutional layers - it results in similar dimensionality reduction with no loss of accuracy. In this experiment the stride length was fixed at 2 for convolutional layers, but future work could simply vary the stride length and do away with pooling layers.

Table 13: Fully connected layer count distribution per generation

Count of Fully Connected layers	0	1	2	3	4	5	6	7	8	9	10
0.00	4	11	12	12	11	12	8	6	6	12	12
1.00	2	1			1		4	3	3		
2.00	3							3	3		
3.00	1										
4.00	1										
5.00	1										

Table 14: Convolutional 2D layer count distribution per generation

count_conv2d	0	1	2	3	4	5	6	7	8	9	10
0.00	3										
1.00	3			1		1		2	1		
2.00	1	2	2	4	6	5	1	3	3	4	4
3.00	1	4	4	2	5	3	4	7	8	6	5
4.00	4	5	5	5	1	2	3			1	3
5.00		1	1			1	2			1	
6.00							2				

Table 15: Max pooling layer count distribution per generation

Count of Max Pool layers	0	1	2	3	4	5	6	7	8	9	10
0.00	5	6	8	4	5	6	9	9	7	4	6
1.00	3	1	4	7	4	3			3	3	4
2.00	4	5		1	3	2	3	1	2	3	2
3.00						1		2		2	

4.4.3 Learning rate

Each chromosome is composed of a learning rate in addition to the neural network architecture. It is a critical component of each model. Figure 40 shows the progression of the learning rate over time. The figure is a standard box plot with the individual learning rates jittered as an overlay. An initially tight distribution of learning rate widens over time. The median of the initial learning rate generating function was designed to be 0.0001 - the same as the original OpenAI model. However, this changed considerably and was approximately 4 times higher in the final generation meaning that a smaller learning rate was not helpful in survival. Additionally, the very high learning rates from generation 8 and 9 did not survive long. Higher learning rates allow for larger parameter adjustments during training, but when they are too high they may not allow the model to converge on a maxima - this is borne out in Figure 40. For this task a learning rate > 0.0001 and < 0.0005 appears to be the most optimal range.

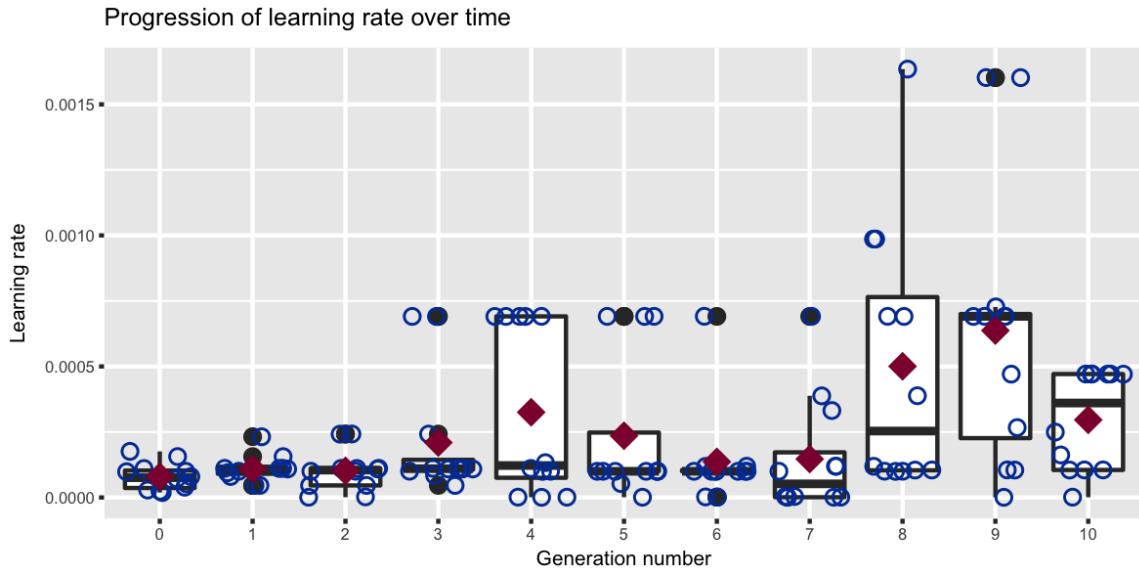


Figure 40: The progression of learning rate over 10 generations

4.4.4 Activation functions

Activation functions are what differentiates deep neural networks from ordinary linear functions. Because the convolutional 2D layers were shown to be the most effective at creating fit and surviving chromosomes, this section will focus on their activations (all described in section 2.3.9):

- ELU - Exponential linear unit
- ReLU - Rectified linear unit
- Leaky ReLU - Leaky rectified linear unit

The most successful activation function was the ELU function, which allows for small negative output values. Distribution of ELU survival is shown in Table 16. For most generations, including the last, all models had at least one ELU activated layer, with half having at least 2 ELU activated layers in the final generation.

The persistence of ELU layers over time is likely not random, despite it being the sole activation of the original OpenAI chromosome - it wasn't present at all in the Model # 002 ancestor, and had an equal chance of being added during layer mutation (addition and change mutation types).

The ReLU distribution is shown in Table 17, and shows the worst performance with most chromosomes having no ReLU layers in all but two generations. In generation 9, 75% of chromosomes

have at least one layer that is ReLU activated, but this reduces to just 33% by generation 10 demonstrating that it was not a sustainable choice for survival. The result is surprising given the popularity of ReLU in the literature.

The Leaky ReLU distribution is shown in Table 18. The results are similar to that of ReLU although slightly better - in the last two generations the majority of chromosomes have at least one Leaky ReLU activated layer.

Table 16: ELU activation count distribution per generation

Count of ELU activations	0	1	2	3	4	5	6	7	8	9	10
0.00	8	4								2	
1.00	3	5	4	7	9	5	3	3	4	7	6
2.00			2	2	1	2		3	5	2	4
3.00			1	4	2	1	3	4	6	3	1
4.00	1	2	2	1	1	2	5				

Table 17: ReLU activation count distribution per generation

Count of ReLU activations	0	1	2	3	4	5	6	7	8	9	10
0.00	8	4	7	7	7	10	8	11	7	3	8
1.00	2	5	3	3	1	2	2	1	5	8	3
2.00	1	3	2	2	4		1		1	1	
3.00	1						1				

Table 18: Leaky ReLU activation count distribution per generation

Count of Leaky ReLU activations	0	1	2	3	4	5	6	7	8	9	10
0.00	6	6	7	6	8	7	9	11	9	3	5
1.00	3		4	5	4	5	1	1	3	7	4
2.00	2	4	1	1			1			2	3
3.00	1	2					1				

4.4.5 Convolutional 2D filters

Given that convolutional 2D layers were shown to lead to fit chromosomes, further analysis on the specific structure of the layers is warranted. Two simple metrics are proposed.

In Equation 48 a ratio called the Filter Ratio is devised. The metric quantifies the relative number of earlier convolutional layer filters to later ones - for models where there are at least two convolutional layers. A ratio greater than 1 indicated that earlier layers have more filters than later ones, while less than 1 indicates the more typical case where earlier layers are less numerous (i.e. indicative of fewer low level features). Figure 41 shows how this ratio changes over the course

of evolution. There is a wide distribution in early generations, but it narrows steadily with all chromosomes exhibiting a ratio less than or equal to 1 from generation 6 onwards. The practice of having fewer filters in earlier layers is widespread in the literature - for example in the classic paper by Krizhevsky et al. (2012) that introduced what became known as AlexNet.

$$\text{FilterRatio} = \prod_{i=1}^N \frac{X_i}{X_{i+1}}$$

where, (48)

N = Number of convolutional layers, and,

$$N \geq 2$$

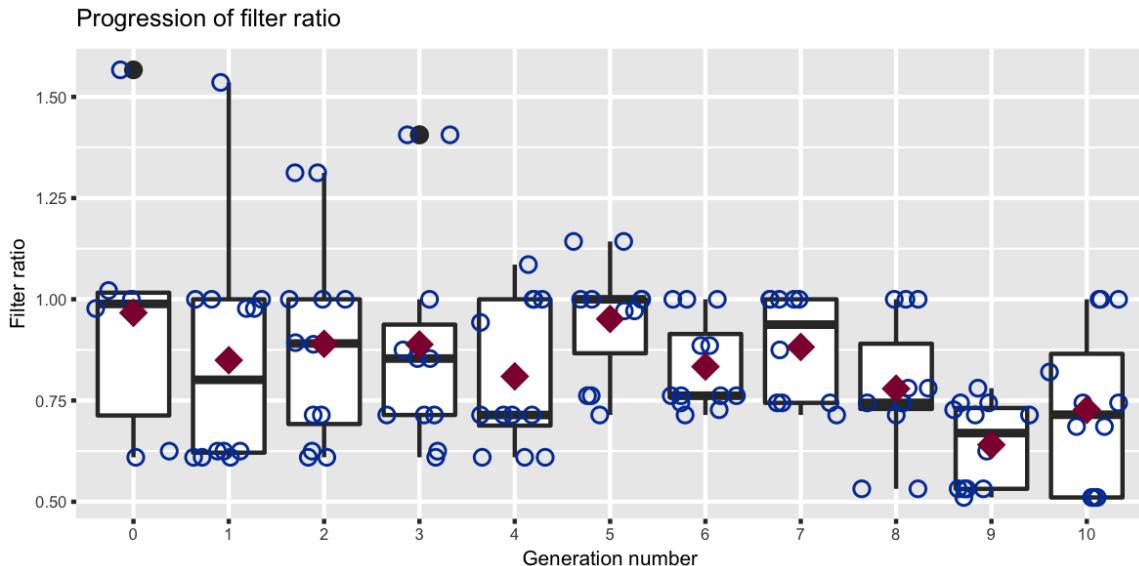


Figure 41: The progression of the filter ratio - a measure of size of earlier layers to later layers when there are 2 or more convolutional layers.

The average number of filters per layer was analysed, with results in Figure 42. No discernible pattern seems to be present, and is likely prevented from occurring - the number of filters per layer was set to occur within a fairly narrow range between 24 and 48. Future work may find more interesting results with a wider range of possibilities.

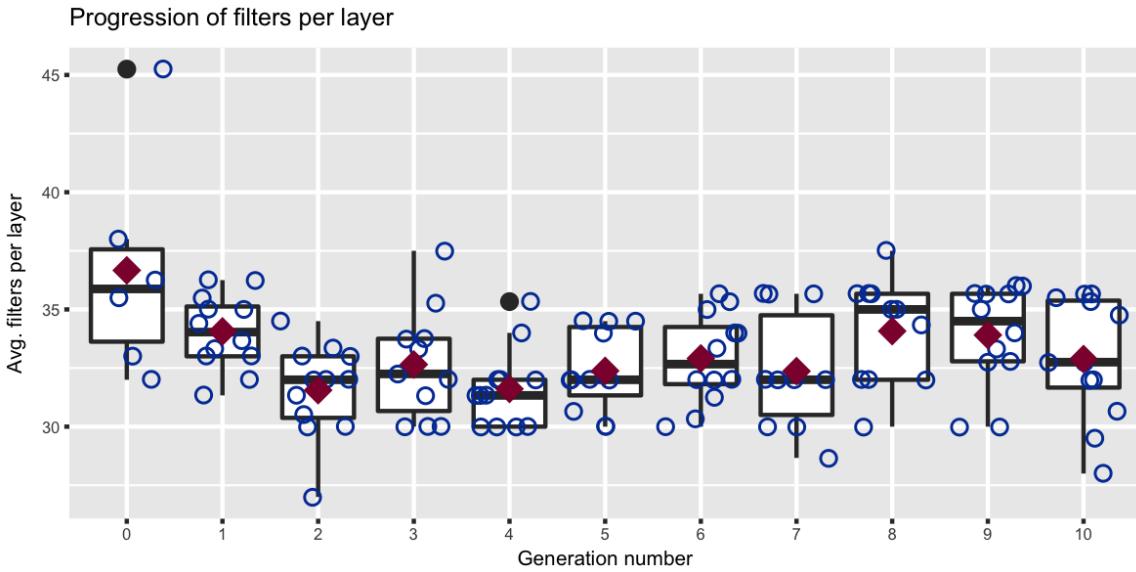


Figure 42: The progression of average number of filters per convolutional 2D layer, when there are 2 or more convolutional layers.

4.4.6 Model examples

The final analysis of neural network architecture takes a more qualitative approach. High level details of samples of the best performing chromosomes and the worst performing are shown. The idea is to demonstrate the variety of architectures that may lead to good and bad results - while the previous observations are indicative of generally good design in terms of specific hyper-parameters, these examples show that the whole chromosome must work as a combination of alleles (genes at a specific point in the structure) that express as a phenotype (the physical manifestation of a gene) that is fit for survival.

Table 19 shows the best 10 models measured by the Tournament Score metric described in Equation 46. All these models performed similarly - there is little difference between 0.59 and 0.61. However, the variety of form is interesting:

- The number of parameters ranges from ~ 8000 to ~ 40000 .
- Output to the LSTM ranges from 168 to ~ 1500 - however it is never as small as a fully connected layer given that none of the top models had one.
- Some models had as many as 5 convolutional layers while others just 2.
- The filter ratio is below 1 for most, and as low as ~ 0.5 for some.

- Commonality can be seen amongst learning rates - most are within the order of 10^{-4} but there are two models where the rate is 2-4X higher.

Table 19: The best performing 10 models and their attributes

T Score	LR	Params	Output	Layers	Conv2D	Max Pool	Full Conn	Filter ratio
0.59	0.0001	28 064	288	4	4	0	0	1.00
0.59	0.0001	40 202	168	5	5	0	0	0.76
0.60	0.000109	20 800	1 440	3	3	0	0	0.62
0.60	0.000111	16 951	369	4	3	1	0	0.61
0.60	0.0001	18 816	1 152	3	3	0	0	1.00
0.60	0.000105	21 417	1 476	3	3	0	0	0.78
0.60	0.000471	28 427	188	5	4	1	0	0.51
0.61	0.000109	23 730	1 440	3	3	0	0	0.62
0.61	0.000242	8 160	1 260	3	2	1	0	0.71
0.61	0.000133	9 568	1 152	3	2	1	0	1.00

The same format shows the worst performing models in Table 20. Again, there is no one specific pattern, but many models exhibit outlier parameters:

- The upper range of parameters is ~ 70000 , which is almost 2X the best models.
- 4 models have very small outputs to the LSTM at less than 50, while two were higher than 4000.
- Two models had fully connected layers, although the distribution of number of convolutional layers was similar to the best models.
- Two models had filter ratios higher than 1, and the average filter ratio is higher than the best models.
- The most notable difference to the best models is the learning rates. 3 were significantly higher than the 10^{-4} mark, while 5 were orders of magnitude lower.

Table 20: The worst performing 10 models and their attributes

T Score	LR	Params	Output	Layers	Conv2D	Max Pool	Full Conn	Filter ratio
1.42	0.000176	72 051	45	6	1	2	3	1.00
1.42	0.000001	16 951	1 476	3	3	0	0	0.61
1.42	0.000004	15 354	1 152	3	3	0	0	1.00
1.43	0.000001	31 044	288	4	4	0	0	1.31
1.43	0.000691	8 160	4 235	2	2	0	0	0.71
1.43	0.000471	13 575	140	5	3	2	0	0.69
1.43	0.000001	11 591	4 719	2	2	0	0	0.82
1.44	0.000071	71 871	41	5	4	0	1	1.02
1.44	0.0001	55 040	42	6	6	0	0	0.76
1.44	0.000002	25 415	35	6	4	2	0	0.89

4.5 Conclusions

As noted in the previous section, models need to be considered in their entirety as no specific hyper-parameter allows for good performance by itself. However, some general heuristics emerged from reviewing the change of parameters over the course of evolution, and comparing the fitness of models:

- A learning rate of approximately 10^{-4} or up to 4X higher works well - rates that are orders of magnitude lower do not result in good performance.
- At least two convolutional layers are required, although more than that and less than five seem optimal. For lower numbers of convolutional layers, a max pooling layer can be helpful as a way to reduce the size of the output to the LSTM layer.
- The ELU activation is the most helpful, and ReLU the least helpful. Leaky ReLU did not perform badly, but not nearly as well as ELU. Other activation types such as Sigmoid were not tried for convolutional layers so no comment is made on them, except to note that they are not typically used in practice.
- The number of filters in earlier convolutional layers should be less than that of later layers - using the filter ratio described in Equation 48 the result should be less than 1.

5 Conclusion

Due to the inefficiency of grid and random search approaches to hyper-parameter optimisation, an evolutionary method based on [Dufourq and Bassett \(2017\)](#) was taken to find more optimized neural

networks for policy function approximation in A3C reinforcement learning. This is a novel approach to model discovery in RL, where models are typically hand-crafted, or optimised with some basic grid-like approach as with Mnih et al. (2015).

The results of the experiment showed that an evolutionary approach could be used successfully to find models that were: skilful; robust; efficient. Several models found performed better, or as well, as state-of-the-art approaches with improvements on the number of model parameters and average episode time (efficiency). Additionally, more general observations about neural network structures were made, that could even benefit a hand-crafted heuristic approach to model design.

There were several limitations with the experiment, primarily due to the computational cost of the RL training, as well as the complex search space of the hyper-parameters. In particular, the limited size of the initial population, the number of total generations and size of new generations, and the bounds of the individual hyper-parameter search areas, were all restrictions on the scope of the results. These issues may be addressed in the future with increased computational resources (including GPU hardware), more efficient approaches to RL than A3C, better chosen hyper-parameter bounds, and different RL tasks.

Future research into the problem would do well to address these limitations, as well as extend the research in a number of ways: Different fitness functions, or coefficients in the current function, to optimise for other aspects of performance; comparison of RL on pixel input versus physics-based input such as coordinates of object location and direction; easier and more difficult tasks comparing the generalizability of found models; incorporation of more hyper-parameters such as the size of the LSTM layer; different parameters for tournament selection, or alternative parent selection methods; different genetic operators such as *crossover*; more types of neural network layers or optimisers such as *residual neural networks* or *dropout* or norm *regularization*; longer training times, or shorter training times (to optimise for speed of convergence). Each variation may solve a particular challenge for researchers and practitioners alike. Additional research may further extend to more practical RL applications outside of simulated game environments to tasks such as training RL agents how to walk, or even telescope array optimization on projects such as the Square Kilometre Array (SKA).

References

- Akandeh, A. and Salem, F. M. (2017), ‘Simplified Long Short-term Memory Recurrent Neural Networks: part I’.
- URL:** <http://arxiv.org/abs/1707.04619>
- Amidi, A. and Amidi, S. (2018), ‘CS 230 - Convolutional Neural Networks Cheatsheet’.
- URL:** <https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks>
- Andersch, M. (2015), ‘Inference: The Next Step in GPU-Accelerated Deep Learning’.
- URL:** <https://devblogs.nvidia.com/inference-next-step-gpu-accelerated-deep-learning/>
- Back, T. (1994), Selective pressure in evolutionary algorithms: a characterization of selection mechanisms, in ‘Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence’, pp. 57–62.
- URL:** <https://pdfs.semanticscholar.org/ea3c/6ef1b13eb007a5a633a71c011fb0f9843218.pdf>
<http://ieeexplore.ieee.org/document/350042/>
- Bakker, B. (2016), Reinforcement learning memory, in ‘Advances in Neural Information Processing Systems’, Curran Associates.
- URL:** <https://papers.nips.cc/paper/1953-reinforcement-learning-with-long-short-term-memory.pdf>
- Bergstra, J. and Bengio, Y. (2012), Random Search for Hyper-Parameter Optimization, Technical report, Université de Montréal.
- URL:** <http://scikit-learn.sourceforge.net>.
- Blackwell, T., Gray, J., Oliver, A. and Brockman, G. (2018), ‘universe-starter-agent’.
- URL:** <https://github.com/openai/universe-starter-agent>
- Blickle, T., Blickle, T. and Thiele, L. (1997), ‘A Comparison of Selection Schemes used in Evolutionary Algorithms’, *EVOLUTIONARY COMPUTATION* 4, 361—394.
- URL:** <http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.9584>
- Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W. (2016), ‘OpenAI Gym’.
- URL:** <http://arxiv.org/abs/1606.01540>

- Claesen, M. and De Moor, B. (2015), Hyperparameter Search in Machine Learning, *in* ‘MIC 2015: The XI Metaheuristics International Conference’, Agadir, Morocco.
- URL:** <https://www.codalab.org/competitions/2321>.
- Ding, B., Qian, H. and Zhou, J. (2018), Activation functions and their characteristics in deep neural networks, *in* ‘2018 Chinese Control And Decision Conference (CCDC)’, IEEE, pp. 1836–1841.
- URL:** <https://ieeexplore.ieee.org/document/8407425/>
- Dufourq, E. (2018), Evolutionary Machine Learning, Phd, University of Cape Town.
- Dufourq, E. and Bassett, B. A. (2017), ‘EDEN: Evolutionary Deep Networks for Efficient Machine Learning’.
- URL:** <http://arxiv.org/abs/1709.09161>
- Feurer, M., Springenberg, J. T., Klein, A., Blum, M., Eggensperger, K. and Hutter, F. (2015), Efficient and Robust Automated Machine Learning, *in* ‘Proceedings of the 28th International Conference on Neural Information Processing Systems’, pp. 2755–2763.
- URL:** <https://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning>
- Gal, Y., Hron, J. and Kendall, A. (2017), Concrete Dropout, *in* I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan and R. Garnett, eds, ‘Advances in Neural Information Processing Systems 30’, Curran Associates, Inc., pp. 3581–3590.
- URL:** <http://papers.nips.cc/paper/6949-concrete-dropout.pdf>
- Gašić, M. G. (2017), Actor-critic methods, Technical report, Cambridge University.
- URL:** <http://mi.eng.cam.ac.uk/~mg436/LectureSlides/MLSALT7/L5.pdf>
- Gibbs, S. (2017), ‘AlphaZero AI beats champion chess program after teaching itself in four hours — Technology — The Guardian’.
- URL:** <https://www.theguardian.com/technology/2017/dec/07/alphazero-google-deepmind-ai-beats-champion-program-teaching-itself-to-play-four-hours>
- Goldberg, D. E. D. E. (1989), *Genetic algorithms in search, optimization, and machine learning*, Addison Wesley.
- Goodfellow, I., Bengio, Y. and Courville, A. (2016), *Deep Learning*, MIT Press.

- Grubb, J. (2019), ‘DeepMind’s AlphaStar AI sweeps StarCraft II pros in head-to-head match — VentureBeat’.
- URL:** <https://venturebeat.com/2019/01/24/alphastar-deepmind-beats-starcraft-pros/>
- Hastie, T., Tibshirani, R. and Friedman, J. (2009), *The Elements of Statistical Learning*, Springer Series in Statistics, Springer New York, New York, NY.
- URL:** <http://link.springer.com/10.1007/978-0-387-84858-7>
- Hausknecht, M. and Stone, P. (2015), ‘Deep Recurrent Q-Learning for Partially Observable MDPs’.
- URL:** <http://arxiv.org/abs/1507.06527>
- Hochreiter, S. and Schmidhuber, J. (1997), ‘Long Short-Term Memory’, *Neural Computation* **9**(8), 1735–1780.
- URL:** <http://www.mitpressjournals.org/doi/10.1162/neco.1997.9.8.1735>
- Idrissi, M. A. J., Ramchoun, H., Ghanou, Y. and Ettaouil, M. (2016), Genetic algorithm for neural network architecture optimization, in ‘2016 3rd International Conference on Logistics Operations Management (GOL)’, IEEE, pp. 1–4.
- URL:** <http://ieeexplore.ieee.org/document/7731699/>
- Jin, H., Song, Q. and Hu, X. (2018), ‘Auto-Keras: Efficient Neural Architecture Search with Network Morphism’.
- URL:** <http://arxiv.org/abs/1806.10282>
- Jinghui Zhong, Xiaomin Hu, Jun Zhang and Min Gu (2005), Comparison of Performance between Different Selection Strategies on Simple Genetic Algorithms, in ‘International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC’06)’, Vol. 2, IEEE, pp. 1115–1121.
- URL:** <http://ieeexplore.ieee.org/document/1631619/>
- Jordan, J. (2018), ‘Setting the learning rate of your neural network.’.
- URL:** <https://www.jeremyjordan.me/nn-learning-rate/>
- Juliani, A. (2016), ‘Simple Reinforcement Learning with Tensorflow Part 8: Asynchronous Actor-Critic Agents (A3C)’.

URL: <https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2>

Kanai, S., Fujiwara, Y. and Iwamura, S. (2017), ‘Preventing Gradient Explosions in Gated Recurrent Units’.

URL: <https://papers.nips.cc/paper/6647-preventing-gradient-explosions-in-gated-recurrent-units>

Karpathy, A. (2016a), ‘Deep Reinforcement Learning: Pong from Pixels’.

URL: <http://karpathy.github.io/2016/05/31/r/>

Karpathy, A. (2016b), ‘Training a Neural Network ATARI Pong agent with Policy Gradients from raw pixels’.

URL: <https://gist.github.com/karpathy/a4166c7fe253700972fcbe77e4ea32c5>

Kotthoff, L., Thornton, C., Hoos, H. H., Hutter, F. and Leyton-Brown, K. (2016), Auto-WEKA 2.0: Automatic model selection and hyperparameter optimization in WEKA, Technical report, University of British Columbia.

URL: <http://automl.org/autoweka>

Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2012), ‘ImageNet Classification with Deep Convolutional Neural Networks’.

URL: <https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks>

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. and Jackel, L. D. (1989), ‘Backpropagation Applied to Handwritten Zip Code Recognition’, *Neural Computation* 1(4), 541–551.

URL: <http://www.mitpressjournals.org/doi/10.1162/neco.1989.1.4.541>

Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998), ‘Gradient-based learning applied to document recognition’, *Proceedings of the IEEE* 86(11), 2278–2324.

URL: <http://ieeexplore.ieee.org/document/726791/>

LeCun, Y. and Cortes, C. (2010), ‘MNIST handwritten digit database’, *AT&T Labs [Online]. Available: <http://yann.lecun.com/exdb/mnist>*.

URL: <http://yann.lecun.com/exdb/mnist/>

Levine, S. (2018a), Actor-Critic Algorithms CS 294-112: Deep Reinforcement Learning, Technical report, University of California, Berkeley.

URL: <http://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-6.pdf>

Levine, S. (2018b), Introduction to Reinforcement Learning CS 294-112: Deep Reinforcement Learning, Technical report, University of California, Berkeley.

URL: http://rll.berkeley.edu/deeprlcourse/f17docs/lecture_3_rl_intro.pdf

Maclaurin, D., Duvenaud, D. and Adams, R. P. (2015), ‘Gradient-based Hyperparameter Optimization through Reversible Learning’.

URL: <http://arxiv.org/abs/1502.03492>

Miller, J. F. (2014), ‘Cartesian Genetic Programming in a nutshell’.

URL: <http://www.cartesiangp.com> <http://www.cartesiangp.co.uk/cgp-in-nutshell.pdf>

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D. and Kavukcuoglu, K. (2016), ‘Asynchronous Methods for Deep Reinforcement Learning’.

URL: <http://arxiv.org/abs/1602.01783>

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S. and Hassabis, D. (2015), ‘Human-level control through deep reinforcement learning’, *Nature* **518**.

URL: <https://storage.googleapis.com/deepmind-media/dqn/DQNNaturePaper.pdf>

Ng, A. Y., Ng, A. Y., Harada, D. and Russell, S. (1999), ‘Policy invariance under reward transformations: Theory and application to reward shaping’, *IN PROCEEDINGS OF THE SIXTEENTH INTERNATIONAL CONFERENCE ON MACHINE LEARNING* pp. 278—287.

URL: <http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.48.345>

OpenAI (2018), ‘OpenAI Five’.

URL: <https://blog.openai.com/openai-five/>

Ruder, S. (2016), ‘An overview of gradient descent optimization algorithms’.

URL: <http://arxiv.org/abs/1609.04747>

Schulman, J., Levine, S., Moritz, P., Jordan, M. I. and Abbeel, P. (2015), ‘Trust Region Policy Optimization’.

URL: <https://arxiv.org/pdf/1502.05477.pdf> <http://arxiv.org/abs/1502.05477>

Sha, L., Chang, B., Sui, Z. and Li, S. (2016), ‘Reading and Thinking: Re-read LSTM Unit for Textual Entailment Recognition’, *undefined*.

URL: <https://www.semanticscholar.org/paper/Reading-and-Thinking%3A-Re-read-LSTM-Unit-for-Textual-Sha-Chang/4c7e85ff37dd8b99d8f443eabd3b163ff8b71538>

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. and Hassabis, D. (2016), ‘Mastering the game of Go with deep neural networks and tree search’, *Nature* **529**.

URL: <https://storage.googleapis.com/deepmind-media/alphago/AlphaGoNaturePaper.pdf>

Snoek, J., Larochelle, H. and Adams, R. P. (2012), ‘Practical Bayesian Optimization of Machine Learning Algorithms’, *eprint arXiv:1206.2944*.

URL: <http://arxiv.org/abs/1206.2944>

Springenberg, J. T., Dosovitskiy, A., Brox, T. and Riedmiller, M. (2014), ‘Striving for Simplicity: The All Convolutional Net’.

URL: <https://arxiv.org/pdf/1412.6806.pdf> <http://arxiv.org/abs/1412.6806>

Srivastava, N., Hinton, G., Krizhevsky, A. and Salakhutdinov, R. (2014), Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Technical report, University of Toronto.

URL: <http://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf>

Sutton, R. S. and Barto, A. G. (2018), *Reinforcement Learning: An Introduction*, second edn, The MIT Press, Cambridge, Massachusetts.

Wierstra, D., Förster, A., Peters, J. and Schmidhuber, J. (2009), ‘Recurrent policy gradients’, *Logic Journal of the IGPL* **18**(5), 620–634.

URL: <http://people.idsia.ch/juergen/joa2009.pdf>

Wilson, D. G., Cussat-Blanc, S., Luga, H. and Miller, J. F. (2018), ‘Evolving simple programs for

playing Atari games'.

URL: <https://arxiv.org/pdf/1806.05695.pdf>

Zoph, B. and Le, Q. V. (2016), 'Neural Architecture Search with Reinforcement Learning'.

URL: <http://arxiv.org/abs/1611.01578>

Appendices

A Initial progress with vanilla policy gradient

The first version of reinforcement learning attempted on the Pong task (see Section 3) was *vanilla policy gradient*. The approach taken was largely influenced from the excellent blog post and associated Python code by [Karpathy \(2016a\)](#). In the experiment conducted as part of the early learning of this dissertation, the RL agent can be described as follows:

- Environment: Pong-v0 from OpenAi ([Brockman et al., 2016](#))
- Policy gradient method
- Policy update after each game episode (when either the built-in Pong AI wins, or the RL agent wins best to 21)
- Preprocessing of game pixels:
 - Original image is 210x160x3 (i.e. 3 colour channels)
 - Retain rows 35 to 195
 - Downsample image by 2
 - Convert to single channel
 - Convert background to 0, everything else to 1
 - The previous preprocessed image is subtracted from the current image as a way to capture movement between frames.
- Policy function approximation:
 - Layer 0 (input): Preprocessed image as 1x6400 vector
 - Layer 1 (hidden): 200 neuron dense layer
 - Layer 1 (hidden): ReLU activation
 - Layer 2 (output): 1 neuron dense layer (i.e. probability of "Up" action)
 - Layer 2 (output): Sigmoid activation (to create a probability)
- Update RMSProp parameters every 10 episodes
- Starting parameters:

- Learning rate = 0.0001
- $\gamma = 0.99$ (Discount factor for reward)
- RMSProp initial decay rate = 0.99

The original pixels before preprocessing (described above) can be seen by the example in Figure 43. In this particular example, the reinforcement learning agent is about to lose the game episode to the built-in Pong AI - resulting in a final episode score of -8.

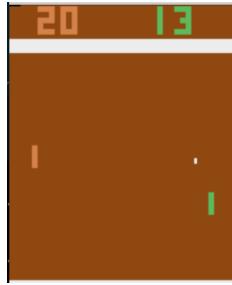


Figure 43: Visualisation of Pong game - RL agent is 1 point away from losing the episode for a net score of -8

The agent was trained on an AWS t2.large EC2 instance using Python3.6. After about 3 days of training the average game episode score (100 game average) is approx. -10. Figure 44 shows the training progress. While the 100 game exponential moving average of the game episode score improves for the duration of the training process, there are a couple other observations:

- The increase in average score appears to slow down after 16000 episodes, and even appears to be stationary from 19000 episodes.
- There is a significant amount of variance in the game results i.e. in the last few thousand episodes there are games where the RL agent beats the built-in AI by 12 points, but also loses by 21 points (the worst possible result).

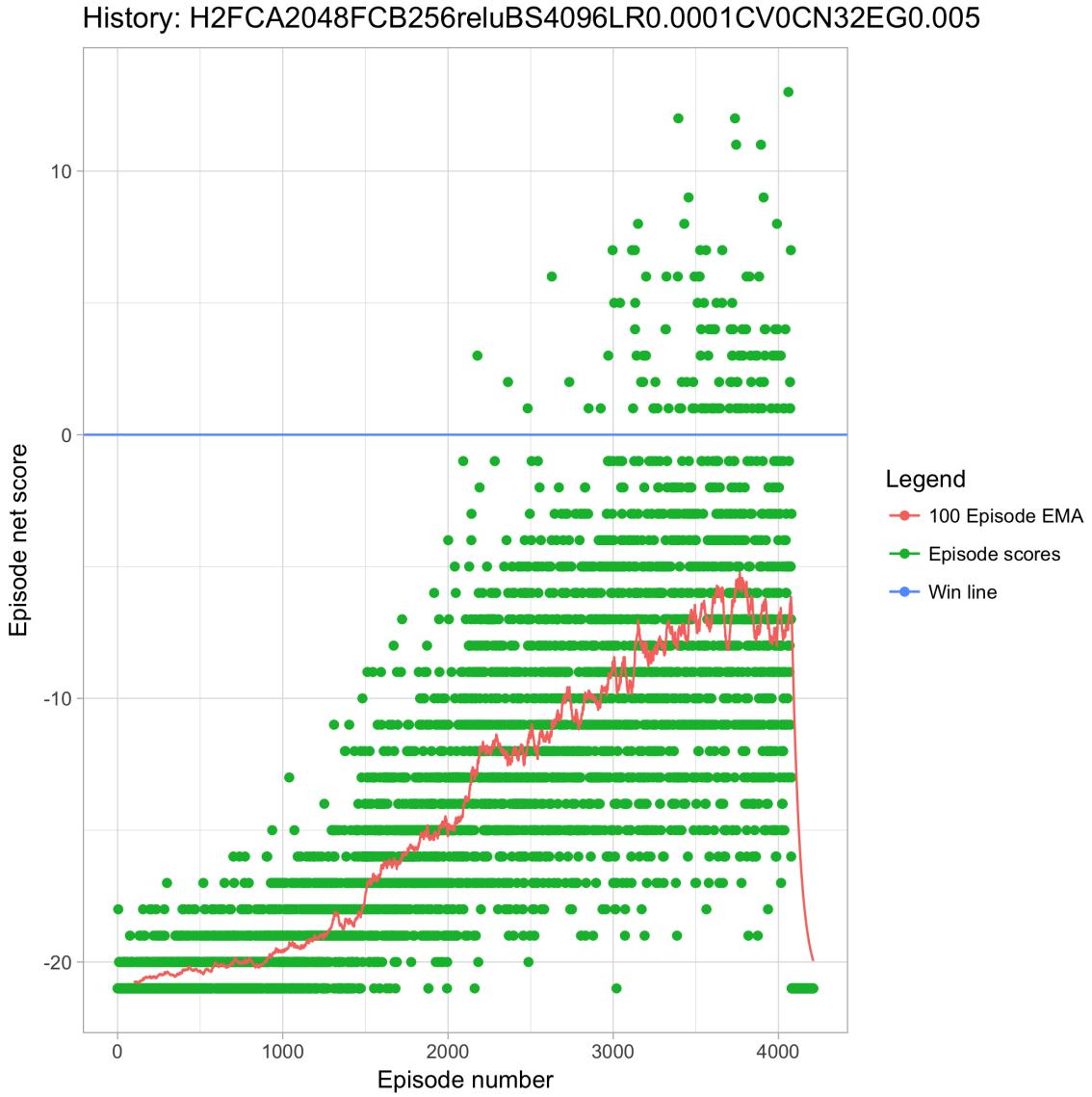


Figure 44: Policy gradient reinforcement learning agent progress on an experiment using a single hidden fully connected layer with size of 200. Notice the high variance during training, and the collapse of the agent in later episodes.

This initial test was a very simple version of what the overall research aims to test. There were no convolutional, recurrent, or residual layers in the network. There was just one hidden layer. The gradient descent optimiser was RMSprop, which is simpler (although often as effective) than Adam. The results are also quite clearly inferior than that achieved with many of the randomly generated models in the main dissertation experiment: worse convergence; much longer training times; far higher variance.

However, vanilla policy gradient is easier to understand when initially learning about reinforcement learning, and the code is easier to implement - it is highly recommended for researchers wishing to begin any work in the area. Additionally, it forms the basis of actor-critic approaches, and there is a significant amount of easier to understand material available to learn about it.

Other initial experiments were conducted with *Q-learning*, another easier to understand approach to RL, with typically better results than vanilla policy gradients. It is also advisable to learn this methodology as it forms another component of actor-critic approaches and introduces the concepts of temporal optimisation in RL. Similarly, there is an abundance of helpful material available to learn and implement the techniques.

B List of models

B.1 Generation 0

LR-00001-P-28064-O-288

Learning rate: 0.000100

Parameters in image layer: 28064

Size of output from image layer to LSTM: 288

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32
1	conv_2d	3	32
2	conv_2d	3	32
3	conv_2d	3	32

LR-0000156-P-28870-O-44

Learning rate: 0.000156

Parameters in image layer: 28870

Size of output from image layer to LSTM: 44

Count of layers

Conv 2D: 4

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	43
1	max_pool_2d	2	
2	conv_2d	3	27
3	conv_2d	3	28
4	max_pool_2d	2	
5	conv_2d	3	44

LR-8e-05-P-38170-O-360

Learning rate: 0.000080

Parameters in image layer: 38170

Size of output from image layer to LSTM: 360

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	25
1	conv_2d	3	40
2	conv_2d	3	40
3	conv_2d	3	40

LR-24e-05-P-644478-O-45

Learning rate: 0.000024

Parameters in image layer: 644478

Size of output from image layer to LSTM: 45

Count of layers

Conv 2D: 1

Max pool 2D: 0

Fully connected: 2

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	47
1	fully_conn		elu
2	fully_conn	45	elu

LR-19e-05-P-88539-O-47

Learning rate: 0.000019

Parameters in image layer: 88539

Size of output from image layer to LSTM: 47

Count of layers

Conv 2D: 0

Max pool 2D: 0

Fully connected: 5

Detailed layers of model:

Type	Size	Number	Activation
0	fully_conn	46	elu
1	fully_conn	42	elu
2	fully_conn	40	relu
3	fully_conn	41	elu
4	fully_conn	47	sigmoid

LR-28e-05-P-20282-O-29

Learning rate: 0.000028

Parameters in image layer: 20282

Size of output from image layer to LSTM: 29

Count of layers

Conv 2D: 0

Max pool 2D: 1
 Fully connected: 2

Detailed layers of model:

Type	Size	Number	Activation
0 max_pool_2d	2		
1 fully_conn	43		relu
2 fully_conn	29		sigmoid

LR-39e-05-P-35934-O-36

Learning rate: 0.000039
 Parameters in image layer: 35934
 Size of output from image layer to LSTM: 36

Count of layers

Conv 2D: 3
 Max pool 2D: 1
 Fully connected: 1

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	47	elu
1 max_pool_2d	2		
2 conv_2d	3	37	leaky_relu
3 conv_2d	3	30	leaky_relu
4 fully_conn	36		relu

LR-0000176-P-72051-O-45

Learning rate: 0.000176
 Parameters in image layer: 72051
 Size of output from image layer to LSTM: 45

Count of layers

Conv 2D: 1
 Max pool 2D: 2
 Fully connected: 3

Detailed layers of model:

Type	Size	Number	Activation
0 max_pool_2d	2		
1 conv_2d	3	43	leaky_relu
2 max_pool_2d	2		
3 fully_conn	44		softmax
4 fully_conn	38		sigmoid
5 fully_conn	45		softmax

LR-52e-05-P-155384-O-36

Learning rate: 0.000052
 Parameters in image layer: 155384

Size of output from image layer to LSTM: 36

Count of layers

Conv 2D: 1

Max pool 2D: 1

Fully connected: 2

Detailed layers of model:

	Type	Size	Number	Activation
0	max_pool_2d	2		
1	conv_2d	3	41	leaky_relu
2	fully_conn		31	elu
3	fully_conn		36	elu

LR-98e-05-P-8338-O-24

Learning rate: 0.000098

Parameters in image layer: 8338

Size of output from image layer to LSTM: 24

Count of layers

Conv 2D: 0

Max pool 2D: 2

Fully connected: 4

Detailed layers of model:

	Type	Size	Number	Activation
0	max_pool_2d	2		
1	max_pool_2d	2		
2	fully_conn		34	relu
3	fully_conn		46	softmax
4	fully_conn		36	relu
5	fully_conn		24	sigmoid

LR-71e-05-P-71871-O-41

Learning rate: 0.000071

Parameters in image layer: 71871

Size of output from image layer to LSTM: 41

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 1

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	47	elu
1	conv_2d	3	44	relu
2	conv_2d	3	44	relu
3	conv_2d	3	46	relu
4	fully_conn		41	relu

LR-0000111-P-9516-O-369

Learning rate: 0.000111

Parameters in image layer: 9516

Size of output from image layer to LSTM: 369

Count of layers

Conv 2D: 2

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	25	relu
1 max_pool_2d	2		
2 max_pool_2d	2		
3 conv_2d	3	41	relu

B.2 Generation 1**LR-00001-P-28064-O-288**

Learning rate: 0.000100

Parameters in image layer: 28064

Size of output from image layer to LSTM: 288

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 conv_2d	3	32	elu
3 conv_2d	3	32	elu

LR-0000111-P-9516-O-369

Learning rate: 0.000111

Parameters in image layer: 9516

Size of output from image layer to LSTM: 369

Count of layers

Conv 2D: 2

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	25	relu
1	max_pool_2d	2		
2	max_pool_2d	2		
3	conv_2d	3	41	relu

LR-8e-05-P-38170-O-360

Learning rate: 0.000080

Parameters in image layer: 38170

Size of output from image layer to LSTM: 360

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	25	leaky_relu
1	conv_2d	3	40	leaky_relu
2	conv_2d	3	40	relu
3	conv_2d	3	40	leaky_relu

LR-0000156-P-28870-O-44

Learning rate: 0.000156

Parameters in image layer: 28870

Size of output from image layer to LSTM: 44

Count of layers

Conv 2D: 4

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	43	leaky_relu
1	max_pool_2d	2		
2	conv_2d	3	27	relu
3	conv_2d	3	28	elu
4	max_pool_2d	2		
5	conv_2d	3	44	leaky_relu

LR-46e-05-P-44236-O-128

Learning rate: 0.000046

Parameters in image layer: 44236

Size of output from image layer to LSTM: 128

Count of layers

Conv 2D: 5

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32
1	conv_2d	3	32
2	conv_2d	3	32
3	conv_2d	3	44
4	conv_2d	3	32

LR-0000111-P-16951-O-369

Learning rate: 0.000111

Parameters in image layer: 16951

Size of output from image layer to LSTM: 369

Count of layers

Conv 2D: 3

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	25
1	max_pool_2d	2	
2	conv_2d	3	28
3	conv_2d	3	41

LR-0000109-P-23730-O-1440

Learning rate: 0.000109

Parameters in image layer: 23730

Size of output from image layer to LSTM: 1440

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	25
1	conv_2d	3	40
2	conv_2d	3	40

LR-0000232-P-30790-O-44

Learning rate: 0.000232

Parameters in image layer: 30790

Size of output from image layer to LSTM: 44

Count of layers

Conv 2D: 4

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	43	leaky_relu
1 max_pool_2d	2		
2 conv_2d	3	30	leaky_relu
3 conv_2d	3	28	elu
4 max_pool_2d	2		
5 conv_2d	3	44	leaky_relu

LR-46e-05-P-34988-O-288

Learning rate: 0.000046

Parameters in image layer: 34988

Size of output from image layer to LSTM: 288

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 conv_2d	3	44	relu
3 conv_2d	3	32	elu

LR-0000111-P-9516-O-369

Learning rate: 0.000111

Parameters in image layer: 9516

Size of output from image layer to LSTM: 369

Count of layers

Conv 2D: 2

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	25	relu
1 max_pool_2d	2		
2 max_pool_2d	2		
3 conv_2d	3	41	relu

LR-0000109-P-20800-O-1440

Learning rate: 0.000109
 Parameters in image layer: 20800
 Size of output from image layer to LSTM: 1440

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	25	leaky_relu
1 conv_2d	3	35	elu
2 conv_2d	3	40	leaky_relu

LR-94e-05-P-22822-O-28

Learning rate: 0.000094
 Parameters in image layer: 22822
 Size of output from image layer to LSTM: 28

Count of layers

Conv 2D: 3

Max pool 2D: 2

Fully connected: 1

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	43	leaky_relu
1 max_pool_2d	2		
2 conv_2d	3	30	leaky_relu
3 conv_2d	3	28	elu
4 max_pool_2d	2		
5 fully_conn	28		elu

B.3 Generation 2**LR-00001-P-28064-O-288**

Learning rate: 0.000100
 Parameters in image layer: 28064
 Size of output from image layer to LSTM: 288

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	conv_2d	3	32	elu
2	conv_2d	3	32	elu
3	conv_2d	3	32	elu

LR-0000111-P-16951-O-369

Learning rate: 0.000111

Parameters in image layer: 16951

Size of output from image layer to LSTM: 369

Count of layers

Conv 2D: 3

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	25	relu
1	max_pool_2d	2		
2	conv_2d	3	28	elu
3	conv_2d	3	41	relu

LR-00001-P-28064-O-288

Learning rate: 0.000100

Parameters in image layer: 28064

Size of output from image layer to LSTM: 288

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	conv_2d	3	32	elu
2	conv_2d	3	32	elu
3	conv_2d	3	32	elu

LR-0000109-P-20800-O-1440

Learning rate: 0.000109

Parameters in image layer: 20800

Size of output from image layer to LSTM: 1440

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	25
1	conv_2d	3	35
2	conv_2d	3	40

LR-46e-05-P-29220-O-324

Learning rate: 0.000046

Parameters in image layer: 29220

Size of output from image layer to LSTM: 324

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32
1	conv_2d	3	32
2	conv_2d	3	32
3	conv_2d	3	36

LR-0000111-P-24035-O-164

Learning rate: 0.000111

Parameters in image layer: 24035

Size of output from image layer to LSTM: 164

Count of layers

Conv 2D: 4

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	25
1	conv_2d	3	28
2	max_pool_2d	2	
3	conv_2d	3	28
4	conv_2d	3	41

LR-1e-06-P-31044-O-288

Learning rate: 0.000001

Parameters in image layer: 31044

Size of output from image layer to LSTM: 288

Count of layers

Conv 2D: 4

Max pool 2D: 0
 Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	42	leaky_relu
1	conv_2d	3	32	elu
2	conv_2d	3	32	elu
3	conv_2d	3	32	elu

LR-0000242-P-8160-O-4235

Learning rate: 0.000242
 Parameters in image layer: 8160
 Size of output from image layer to LSTM: 4235

Count of layers

Conv 2D: 2
 Max pool 2D: 0
 Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	25	leaky_relu
1	conv_2d	3	35	elu

LR-46e-05-P-18816-O-1152

Learning rate: 0.000046
 Parameters in image layer: 18816
 Size of output from image layer to LSTM: 1152

Count of layers

Conv 2D: 3
 Max pool 2D: 0
 Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	conv_2d	3	32	elu
2	conv_2d	3	32	elu

LR-0000111-P-13662-O-252

Learning rate: 0.000111
 Parameters in image layer: 13662
 Size of output from image layer to LSTM: 252

Count of layers

Conv 2D: 3
 Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	25	relu
1 conv_2d	3	28	elu
2 max_pool_2d	2		
3 conv_2d	3	28	elu

LR-2e-06-P-37407-O-128

Learning rate: 0.000002

Parameters in image layer: 37407

Size of output from image layer to LSTM: 128

Count of layers

Conv 2D: 5

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	42	leaky_relu
1 conv_2d	3	32	elu
2 conv_2d	3	27	relu
3 conv_2d	3	32	elu
4 conv_2d	3	32	elu

LR-0000242-P-8160-O-1260

Learning rate: 0.000242

Parameters in image layer: 8160

Size of output from image layer to LSTM: 1260

Count of layers

Conv 2D: 2

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 max_pool_2d	2		
1 conv_2d	3	25	leaky_relu
2 conv_2d	3	35	elu

B.4 Generation 3

LR-0000111-P-16951-O-369

Learning rate: 0.000111

Parameters in image layer: 16951

Size of output from image layer to LSTM: 369

Count of layers

Conv 2D: 3

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	25	relu
1 max_pool_2d	2		
2 conv_2d	3	28	elu
3 conv_2d	3	41	relu

LR-0000242-P-8160-O-1260

Learning rate: 0.000242

Parameters in image layer: 8160

Size of output from image layer to LSTM: 1260

Count of layers

Conv 2D: 2

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 max_pool_2d	2		
1 conv_2d	3	25	leaky_relu
2 conv_2d	3	35	elu

LR-00001-P-28064-O-288

Learning rate: 0.000100

Parameters in image layer: 28064

Size of output from image layer to LSTM: 288

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 conv_2d	3	32	elu
3 conv_2d	3	32	elu

LR-0000109-P-20800-O-1440

Learning rate: 0.000109

Parameters in image layer: 20800

Size of output from image layer to LSTM: 1440

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	25	leaky_relu
1 conv_2d	3	35	elu
2 conv_2d	3	40	leaky_relu

LR-0000111-P-24951-O-164

Learning rate: 0.000111

Parameters in image layer: 24951

Size of output from image layer to LSTM: 164

Count of layers

Conv 2D: 4

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	35	elu
1 conv_2d	3	25	relu
2 max_pool_2d	2		
3 conv_2d	3	28	elu
4 conv_2d	3	41	relu

LR-0000691-P-8160-O-315

Learning rate: 0.000691

Parameters in image layer: 8160

Size of output from image layer to LSTM: 315

Count of layers

Conv 2D: 2

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 max_pool_2d	2		
1 conv_2d	3	25	leaky_relu
2 conv_2d	3	35	elu
3 max_pool_2d	2		

LR-46e-05-P-31938-O-288

Learning rate: 0.000046

Parameters in image layer: 31938

Size of output from image layer to LSTM: 288

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	45	relu
1	conv_2d	3	32	elu
2	conv_2d	3	32	elu
3	conv_2d	3	32	elu

LR-0000109-P-12990-O-1440

Learning rate: 0.000109

Parameters in image layer: 12990

Size of output from image layer to LSTM: 1440

Count of layers

Conv 2D: 2

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	max_pool_2d	2		
1	conv_2d	3	35	elu
2	conv_2d	3	40	leaky_relu

LR-0000111-P-28359-O-164

Learning rate: 0.000111

Parameters in image layer: 28359

Size of output from image layer to LSTM: 164

Count of layers

Conv 2D: 4

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	35	elu
1	conv_2d	3	31	elu
2	max_pool_2d	2		
3	conv_2d	3	28	elu
4	conv_2d	3	41	relu

LR-0000691-P-8160-O-1260

Learning rate: 0.000691

Parameters in image layer: 8160

Size of output from image layer to LSTM: 1260

Count of layers

Conv 2D: 2

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	max_pool_2d	2		
1	conv_2d	3	25	leaky_relu
2	conv_2d	3	35	elu

LR-84e-05-P-27774-O-288

Learning rate: 0.000084

Parameters in image layer: 27774

Size of output from image layer to LSTM: 288

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	45	relu
1	conv_2d	3	26	leaky_relu
2	conv_2d	3	32	elu
3	conv_2d	3	32	elu

LR-0000109-P-350-O-4235

Learning rate: 0.000109

Parameters in image layer: 350

Size of output from image layer to LSTM: 4235

Count of layers

Conv 2D: 1

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	max_pool_2d	2	
1	conv_2d	3 35	elu

B.5 Generation 4

LR-0000691-P-8160-O-1260

Learning rate: 0.000691

Parameters in image layer: 8160

Size of output from image layer to LSTM: 1260

Count of layers

Conv 2D: 2

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	max_pool_2d	2	
1	conv_2d	3 25	leaky_relu
2	conv_2d	3 35	elu

LR-0000111-P-16951-O-369

Learning rate: 0.000111

Parameters in image layer: 16951

Size of output from image layer to LSTM: 369

Count of layers

Conv 2D: 3

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3 25	relu
1	max_pool_2d	2	
2	conv_2d	3 28	elu
3	conv_2d	3 41	relu

LR-0000691-P-8160-O-315

Learning rate: 0.000691

Parameters in image layer: 8160

Size of output from image layer to LSTM: 315

Count of layers

Conv 2D: 2

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	max_pool_2d	2	
1	conv_2d	3	25
2	conv_2d	3	35
3	max_pool_2d	2	leaky_relu
			elu

LR-00001-P-28064-O-288

Learning rate: 0.000100

Parameters in image layer: 28064

Size of output from image layer to LSTM: 288

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32
1	conv_2d	3	32
2	conv_2d	3	32
3	conv_2d	3	32
			elu

LR-0000691-P-8160-O-4235

Learning rate: 0.000691

Parameters in image layer: 8160

Size of output from image layer to LSTM: 4235

Count of layers

Conv 2D: 2

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	25
1	conv_2d	3	35
			leaky_relu
			elu

LR-1e-06-P-16951-O-1476

Learning rate: 0.000001

Parameters in image layer: 16951

Size of output from image layer to LSTM: 1476

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	25	relu
1 conv_2d	3	28	elu
2 conv_2d	3	41	relu

LR-0000691-P-10760-O-315

Learning rate: 0.000691

Parameters in image layer: 10760

Size of output from image layer to LSTM: 315

Count of layers

Conv 2D: 2

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 max_pool_2d	2		
1 conv_2d	3	33	relu
2 conv_2d	3	35	elu
3 max_pool_2d	2		

LR-00001-P-18816-O-1152

Learning rate: 0.000100

Parameters in image layer: 18816

Size of output from image layer to LSTM: 1152

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 conv_2d	3	32	elu

LR-1e-06-P-109824-O-24

Learning rate: 0.000001

Parameters in image layer: 109824

Size of output from image layer to LSTM: 24

Count of layers

Conv 2D: 2

Max pool 2D: 0

Fully connected: 1

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	25	leaky_relu
1 conv_2d	3	35	elu
2 fully_conn		24	sigmoid

LR-1e-06-P-16951-O-369

Learning rate: 0.000001

Parameters in image layer: 16951

Size of output from image layer to LSTM: 369

Count of layers

Conv 2D: 3

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	25	relu
1 conv_2d	3	28	elu
2 max_pool_2d	2		
3 conv_2d	3	41	relu

LR-0000691-P-22129-O-140

Learning rate: 0.000691

Parameters in image layer: 22129

Size of output from image layer to LSTM: 140

Count of layers

Conv 2D: 3

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 max_pool_2d	2		
1 conv_2d	3	38	relu
2 conv_2d	3	33	relu
3 conv_2d	3	35	elu
4 max_pool_2d	2		

LR-0000133-P-9568-O-1152

Learning rate: 0.000133

Parameters in image layer: 9568

Size of output from image layer to LSTM: 1152

Count of layers

Conv 2D: 2
 Max pool 2D: 1
 Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	max_pool_2d	2	
1	conv_2d	3	32 elu
2	conv_2d	3	32 elu

B.6 Generation 5

LR-00001-P-18816-O-1152

Learning rate: 0.000100
 Parameters in image layer: 18816
 Size of output from image layer to LSTM: 1152

Count of layers

Conv 2D: 3
 Max pool 2D: 0
 Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32 elu
1	conv_2d	3	32 elu
2	conv_2d	3	32 elu

LR-00001-P-18816-O-1152

Learning rate: 0.000100
 Parameters in image layer: 18816
 Size of output from image layer to LSTM: 1152

Count of layers

Conv 2D: 3
 Max pool 2D: 0
 Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32 elu
1	conv_2d	3	32 elu
2	conv_2d	3	32 elu

LR-0000691-P-8160-O-315

Learning rate: 0.000691
 Parameters in image layer: 8160
 Size of output from image layer to LSTM: 315

Count of layers

Conv 2D: 2

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	max_pool_2d	2	
1	conv_2d	3	25
2	conv_2d	3	35
3	max_pool_2d	2	

LR-00001-P-28064-O-288

Learning rate: 0.000100

Parameters in image layer: 28064

Size of output from image layer to LSTM: 288

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32
1	conv_2d	3	32
2	conv_2d	3	32
3	conv_2d	3	32

LR-00001-P-17660-O-1008

Learning rate: 0.000100

Parameters in image layer: 17660

Size of output from image layer to LSTM: 1008

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32
1	conv_2d	3	32
2	conv_2d	3	28

LR-00001-P-9568-O-1152

Learning rate: 0.000100

Parameters in image layer: 9568

Size of output from image layer to LSTM: 1152

Count of layers

Conv 2D: 2

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 max_pool_2d	2		

LR-0000691-P-11085-O-315

Learning rate: 0.000691

Parameters in image layer: 11085

Size of output from image layer to LSTM: 315

Count of layers

Conv 2D: 2

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 max_pool_2d	2		
1 conv_2d	3	34	leaky_relu
2 conv_2d	3	35	elu
3 max_pool_2d	2		

LR-00001-P-40202-O-168

Learning rate: 0.000100

Parameters in image layer: 40202

Size of output from image layer to LSTM: 168

Count of layers

Conv 2D: 5

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 conv_2d	3	32	elu
3 conv_2d	3	32	elu
4 conv_2d	3	42	relu

LR-53e-05-P-8412-O-3388

Learning rate: 0.000053

Parameters in image layer: 8412

Size of output from image layer to LSTM: 3388

Count of layers

Conv 2D: 2

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	conv_2d	3	28	leaky_relu

LR-00001-P-320-O-3872

Learning rate: 0.000100

Parameters in image layer: 320

Size of output from image layer to LSTM: 3872

Count of layers

Conv 2D: 1

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	max_pool_2d	2		

LR-0000691-P-11085-O-140

Learning rate: 0.000691

Parameters in image layer: 11085

Size of output from image layer to LSTM: 140

Count of layers

Conv 2D: 2

Max pool 2D: 3

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	max_pool_2d	2		
1	conv_2d	3	34	leaky_relu
2	max_pool_2d	2		
3	conv_2d	3	35	elu
4	max_pool_2d	2		

LR-1e-06-P-30954-O-168

Learning rate: 0.000001
 Parameters in image layer: 30954
 Size of output from image layer to LSTM: 168

Count of layers

Conv 2D: 4

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	max_pool_2d	2	
1	conv_2d	3	elu
2	conv_2d	3	elu
3	conv_2d	3	elu
4	conv_2d	3	relu

B.7 Generation 6**LR-00001-P-40202-O-168**

Learning rate: 0.000100
 Parameters in image layer: 40202
 Size of output from image layer to LSTM: 168

Count of layers

Conv 2D: 5

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	elu
1	conv_2d	3	elu
2	conv_2d	3	elu
3	conv_2d	3	elu
4	conv_2d	3	relu

LR-00001-P-40202-O-168

Learning rate: 0.000100
 Parameters in image layer: 40202
 Size of output from image layer to LSTM: 168

Count of layers

Conv 2D: 5

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32 elu
1	conv_2d	3	32 elu
2	conv_2d	3	32 elu
3	conv_2d	3	32 elu
4	conv_2d	3	42 relu

LR-0000691-P-8160-O-315

Learning rate: 0.000691

Parameters in image layer: 8160

Size of output from image layer to LSTM: 315

Count of layers

Conv 2D: 2

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	max_pool_2d	2	
1	conv_2d	3	25 leaky_relu
2	conv_2d	3	35 elu
3	max_pool_2d	2	

LR-00001-P-18816-O-1152

Learning rate: 0.000100

Parameters in image layer: 18816

Size of output from image layer to LSTM: 1152

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32 elu
1	conv_2d	3	32 elu
2	conv_2d	3	32 elu

LR-00001-P-48116-O-42

Learning rate: 0.000100

Parameters in image layer: 48116

Size of output from image layer to LSTM: 42

Count of layers

Conv 2D: 6

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 conv_2d	3	32	elu
3 conv_2d	3	32	elu
4 conv_2d	3	30	relu
5 conv_2d	3	42	relu

LR-00001-P-36445-O-29

Learning rate: 0.000100

Parameters in image layer: 36445

Size of output from image layer to LSTM: 29

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 1

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 conv_2d	3	32	elu
3 conv_2d	3	32	elu
4 fully_conn		29	relu

LR-1e-06-P-15220-O-140

Learning rate: 0.000001

Parameters in image layer: 15220

Size of output from image layer to LSTM: 140

Count of layers

Conv 2D: 3

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	31	leaky_relu
1 max_pool_2d	2		
2 conv_2d	3	25	leaky_relu
3 conv_2d	3	35	elu
4 max_pool_2d	2		

LR-0000119-P-70701-O-45

Learning rate: 0.000119
 Parameters in image layer: 70701
 Size of output from image layer to LSTM: 45

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 1

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 conv_2d	3	32	elu
3 fully_conn		45	sigmoid

LR-00001-P-55040-O-42

Learning rate: 0.000100
 Parameters in image layer: 55040
 Size of output from image layer to LSTM: 42

Count of layers

Conv 2D: 6

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	44	relu
2 conv_2d	3	32	elu
3 conv_2d	3	32	elu
4 conv_2d	3	30	relu
5 conv_2d	3	42	relu

LR-00001-P-43045-O-29

Learning rate: 0.000100
 Parameters in image layer: 43045
 Size of output from image layer to LSTM: 29

Count of layers

Conv 2D: 4

Max pool 2D: 0

Fully connected: 1

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	conv_2d	3	32	elu
2	conv_2d	3	32	elu
3	conv_2d	3	44	elu
4	fully_conn		29	relu

LR-2e-06-P-25415-O-35

Learning rate: 0.000002

Parameters in image layer: 25415

Size of output from image layer to LSTM: 35

Count of layers

Conv 2D: 4

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	31	leaky_relu
1	conv_2d	3	34	leaky_relu
2	max_pool_2d	2		
3	conv_2d	3	25	leaky_relu
4	conv_2d	3	35	elu
5	max_pool_2d	2		

LR-0000119-P-91700-O-45

Learning rate: 0.000119

Parameters in image layer: 91700

Size of output from image layer to LSTM: 45

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 1

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	conv_2d	3	32	elu
2	conv_2d	3	43	elu
3	fully_conn		45	sigmoid

B.8 Generation 7**LR-0000119-P-70701-O-45**

Learning rate: 0.000119

Parameters in image layer: 70701

Size of output from image layer to LSTM: 45

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 1

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32
1	conv_2d	3	32
2	conv_2d	3	32
3	fully_conn		45

LR-0000119-P-91700-O-45

Learning rate: 0.000119

Parameters in image layer: 91700

Size of output from image layer to LSTM: 45

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 1

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32
1	conv_2d	3	32
2	conv_2d	3	43
3	fully_conn		45

LR-00001-P-18816-O-1152

Learning rate: 0.000100

Parameters in image layer: 18816

Size of output from image layer to LSTM: 1152

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32
1	conv_2d	3	32
2	conv_2d	3	32

LR-0000691-P-8160-O-315

Learning rate: 0.000691

Parameters in image layer: 8160

Size of output from image layer to LSTM: 315

Count of layers

Conv 2D: 2

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	max_pool_2d	2		
1	conv_2d	3	25	leaky_relu
2	conv_2d	3	35	elu
3	max_pool_2d	2		

LR-1e-06-P-183853-O-45

Learning rate: 0.000001

Parameters in image layer: 183853

Size of output from image layer to LSTM: 45

Count of layers

Conv 2D: 2

Max pool 2D: 0

Fully connected: 1

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	conv_2d	3	32	elu
2	fully_conn		45	sigmoid

LR-0000388-P-93724-O-44

Learning rate: 0.000388

Parameters in image layer: 93724

Size of output from image layer to LSTM: 44

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 2

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	conv_2d	3	32	elu
2	conv_2d	3	43	elu
3	fully_conn		45	sigmoid
4	fully_conn		44	sigmoid

LR-4e-06-P-15354-O-1152

Learning rate: 0.000004

Parameters in image layer: 15354

Size of output from image layer to LSTM: 1152

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	26	elu
2 conv_2d	3	32	elu

LR-1e-06-P-350-O-315

Learning rate: 0.000001

Parameters in image layer: 350

Size of output from image layer to LSTM: 315

Count of layers

Conv 2D: 1

Max pool 2D: 3

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 max_pool_2d	2		
1 max_pool_2d	2		
2 conv_2d	3	35	elu
3 max_pool_2d	2		

LR-1e-06-P-185003-O-25

Learning rate: 0.000001

Parameters in image layer: 185003

Size of output from image layer to LSTM: 25

Count of layers

Conv 2D: 2

Max pool 2D: 0

Fully connected: 2

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 fully_conn		45	sigmoid
3 fully_conn		25	relu

LR-0000332-P-79387-O-44

Learning rate: 0.000332

Parameters in image layer: 79387

Size of output from image layer to LSTM: 44

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 2

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 conv_2d	3	43	elu
3 fully_conn		36	elu
4 fully_conn		44	sigmoid

LR-4e-06-P-14378-O-1152

Learning rate: 0.000004

Parameters in image layer: 14378

Size of output from image layer to LSTM: 1152

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	28	relu
1 conv_2d	3	26	elu
2 conv_2d	3	32	elu

LR-1e-06-P-350-O-315

Learning rate: 0.000001

Parameters in image layer: 350

Size of output from image layer to LSTM: 315

Count of layers

Conv 2D: 1

Max pool 2D: 3

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	max_pool_2d	2		
1	max_pool_2d	2		
2	conv_2d	3	35	elu
3	max_pool_2d	2		

B.9 Generation 8

LR-00001-P-18816-O-1152

Learning rate: 0.000100

Parameters in image layer: 18816

Size of output from image layer to LSTM: 1152

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	conv_2d	3	32	elu
2	conv_2d	3	32	elu

LR-0000119-P-70701-O-45

Learning rate: 0.000119

Parameters in image layer: 70701

Size of output from image layer to LSTM: 45

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 1

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	conv_2d	3	32	elu
2	conv_2d	3	32	elu
3	fully_conn		45	sigmoid

LR-0000691-P-8160-O-315

Learning rate: 0.000691

Parameters in image layer: 8160

Size of output from image layer to LSTM: 315

Count of layers

Conv 2D: 2

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	max_pool_2d	2	
1	conv_2d	3	25
2	conv_2d	3	35
3	max_pool_2d	2	

LR-0000388-P-93724-O-44

Learning rate: 0.000388

Parameters in image layer: 93724

Size of output from image layer to LSTM: 44

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 2

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32
1	conv_2d	3	32
2	conv_2d	3	43
3	fully_conn		sigmoid
4	fully_conn	44	sigmoid

LR-00001-P-9568-O-3872

Learning rate: 0.000100

Parameters in image layer: 9568

Size of output from image layer to LSTM: 3872

Count of layers

Conv 2D: 2

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32
1	conv_2d	3	32

LR-0000105-P-87882-O-45

Learning rate: 0.000105

Parameters in image layer: 87882

Size of output from image layer to LSTM: 45

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 1

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 conv_2d	3	41	relu
3 fully_conn		45	sigmoid

LR-0000691-P-23012-O-188

Learning rate: 0.000691

Parameters in image layer: 23012

Size of output from image layer to LSTM: 188

Count of layers

Conv 2D: 3

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 max_pool_2d	2		
1 conv_2d	3	25	leaky_relu
2 conv_2d	3	35	elu
3 conv_2d	3	47	relu
4 max_pool_2d	2		

LR-0000986-P-84476-O-44

Learning rate: 0.000986

Parameters in image layer: 84476

Size of output from image layer to LSTM: 44

Count of layers

Conv 2D: 2

Max pool 2D: 1

Fully connected: 2

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 max_pool_2d	2		
2 conv_2d	3	43	elu
3 fully_conn		45	sigmoid
4 fully_conn		44	sigmoid

LR-00001-P-320-O-3872

Learning rate: 0.000100

Parameters in image layer: 320

Size of output from image layer to LSTM: 3872

Count of layers

Conv 2D: 1

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 max_pool_2d	2		

LR-0000105-P-21417-O-1476

Learning rate: 0.000105

Parameters in image layer: 21417

Size of output from image layer to LSTM: 1476

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 conv_2d	3	41	relu

LR-0001635-P-37852-O-35

Learning rate: 0.001635

Parameters in image layer: 37852

Size of output from image layer to LSTM: 35

Count of layers

Conv 2D: 3

Max pool 2D: 1

Fully connected: 1

Detailed layers of model:

Type	Size	Number	Activation
0 max_pool_2d	2		
1 conv_2d	3	25	leaky_relu
2 conv_2d	3	35	elu
3 conv_2d	3	47	relu
4 fully_conn		35	sigmoid

LR-0000986-P-91020-O-44

Learning rate: 0.000986

Parameters in image layer: 91020

Size of output from image layer to LSTM: 44

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 2

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	28	relu
2 conv_2d	3	43	elu
3 fully_conn		45	sigmoid
4 fully_conn		44	sigmoid

B.10 Generation 9

LR-0000691-P-8160-O-315

Learning rate: 0.000691

Parameters in image layer: 8160

Size of output from image layer to LSTM: 315

Count of layers

Conv 2D: 2

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 max_pool_2d	2		
1 conv_2d	3	25	leaky_relu
2 conv_2d	3	35	elu
3 max_pool_2d	2		

LR-0000691-P-23012-O-188

Learning rate: 0.000691

Parameters in image layer: 23012

Size of output from image layer to LSTM: 188

Count of layers

Conv 2D: 3

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	max_pool_2d	2		
1	conv_2d	3	25	leaky_relu
2	conv_2d	3	35	elu
3	conv_2d	3	47	relu
4	max_pool_2d	2		

LR-0000691-P-23012-O-188

Learning rate: 0.000691

Parameters in image layer: 23012

Size of output from image layer to LSTM: 188

Count of layers

Conv 2D: 3

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	max_pool_2d	2		
1	conv_2d	3	25	leaky_relu
2	conv_2d	3	35	elu
3	conv_2d	3	47	relu
4	max_pool_2d	2		

LR-0000105-P-21417-O-1476

Learning rate: 0.000105

Parameters in image layer: 21417

Size of output from image layer to LSTM: 1476

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	conv_2d	3	32	elu
2	conv_2d	3	41	relu

LR-0000691-P-8160-O-1260

Learning rate: 0.000691

Parameters in image layer: 8160

Size of output from image layer to LSTM: 1260

Count of layers

Conv 2D: 2

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	25	leaky_relu
1 conv_2d	3	35	elu
2 max_pool_2d	2		

LR-0001602-P-10872-O-188

Learning rate: 0.001602

Parameters in image layer: 10872

Size of output from image layer to LSTM: 188

Count of layers

Conv 2D: 2

Max pool 2D: 3

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 max_pool_2d	2		
1 conv_2d	3	25	leaky_relu
2 max_pool_2d	2		
3 conv_2d	3	47	relu
4 max_pool_2d	2		

LR-0000471-P-28427-O-188

Learning rate: 0.000471

Parameters in image layer: 28427

Size of output from image layer to LSTM: 188

Count of layers

Conv 2D: 4

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	24	leaky_relu
1 conv_2d	3	25	leaky_relu
2 conv_2d	3	35	elu
3 conv_2d	3	47	relu
4 max_pool_2d	2		

LR-0000105-P-21995-O-1548

Learning rate: 0.000105

Parameters in image layer: 21995

Size of output from image layer to LSTM: 1548

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	32
1	conv_2d	3	32
2	conv_2d	3	43

LR-0000727-P-20800-O-360

Learning rate: 0.000727

Parameters in image layer: 20800

Size of output from image layer to LSTM: 360

Count of layers

Conv 2D: 3

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	conv_2d	3	25
1	conv_2d	3	35
2	conv_2d	3	40
3	max_pool_2d	2	

LR-0001602-P-10872-O-188

Learning rate: 0.001602

Parameters in image layer: 10872

Size of output from image layer to LSTM: 188

Count of layers

Conv 2D: 2

Max pool 2D: 3

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0	max_pool_2d	2	
1	conv_2d	3	25
2	max_pool_2d	2	
3	conv_2d	3	47
4	max_pool_2d	2	

LR-1e-06-P-42419-O-132

Learning rate: 0.000001

Parameters in image layer: 42419

Size of output from image layer to LSTM: 132

Count of layers

Conv 2D: 5

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	24	leaky_relu
1	conv_2d	3	25	leaky_relu
2	conv_2d	3	35	elu
3	conv_2d	3	47	relu
4	conv_2d	3	33	relu

LR-0000267-P-18615-O-1548

Learning rate: 0.000267

Parameters in image layer: 18615

Size of output from image layer to LSTM: 1548

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	conv_2d	3	27	relu
2	conv_2d	3	43	elu

B.11 Generation 10

LR-0000471-P-28427-O-188

Learning rate: 0.000471

Parameters in image layer: 28427

Size of output from image layer to LSTM: 188

Count of layers

Conv 2D: 4

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	24	leaky_relu
1	conv_2d	3	25	leaky_relu
2	conv_2d	3	35	elu
3	conv_2d	3	47	relu
4	max_pool_2d	2		

LR-0000471-P-28427-O-188

Learning rate: 0.000471

Parameters in image layer: 28427

Size of output from image layer to LSTM: 188

Count of layers

Conv 2D: 4

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	24	leaky_relu
1 conv_2d	3	25	leaky_relu
2 conv_2d	3	35	elu
3 conv_2d	3	47	relu
4 max_pool_2d	2		

LR-0000105-P-21995-O-1548

Learning rate: 0.000105

Parameters in image layer: 21995

Size of output from image layer to LSTM: 1548

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 conv_2d	3	43	elu

LR-0000105-P-21995-O-1548

Learning rate: 0.000105

Parameters in image layer: 21995

Size of output from image layer to LSTM: 1548

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu
2 conv_2d	3	43	elu

LR-0000471-P-13575-O-140

Learning rate: 0.000471

Parameters in image layer: 13575

Size of output from image layer to LSTM: 140

Count of layers

Conv 2D: 3

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	24	leaky_relu
1 conv_2d	3	25	leaky_relu
2 conv_2d	3	35	elu
3 max_pool_2d	2		
4 max_pool_2d	2		

LR-0000471-P-22687-O-423

Learning rate: 0.000471

Parameters in image layer: 22687

Size of output from image layer to LSTM: 423

Count of layers

Conv 2D: 3

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	24	leaky_relu
1 conv_2d	3	35	elu
2 conv_2d	3	47	relu
3 max_pool_2d	2		

LR-0000162-P-9568-O-3872

Learning rate: 0.000162

Parameters in image layer: 9568

Size of output from image layer to LSTM: 3872

Count of layers

Conv 2D: 2

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu

LR-0000105-P-9568-O-3872

Learning rate: 0.000105
 Parameters in image layer: 9568
 Size of output from image layer to LSTM: 3872

Count of layers

Conv 2D: 2

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	32	elu
1 conv_2d	3	32	elu

LR-0000471-P-7835-O-315

Learning rate: 0.000471
 Parameters in image layer: 7835
 Size of output from image layer to LSTM: 315

Count of layers

Conv 2D: 2

Max pool 2D: 2

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	24	leaky_relu
1 conv_2d	3	35	elu
2 max_pool_2d	2		
3 max_pool_2d	2		

LR-0000471-P-32269-O-188

Learning rate: 0.000471
 Parameters in image layer: 32269
 Size of output from image layer to LSTM: 188

Count of layers

Conv 2D: 4

Max pool 2D: 1

Fully connected: 0

Detailed layers of model:

Type	Size	Number	Activation
0 conv_2d	3	24	leaky_relu
1 conv_2d	3	35	elu
2 conv_2d	3	33	relu
3 conv_2d	3	47	relu
4 max_pool_2d	2		

LR-000025-P-16508-O-1152

Learning rate: 0.000250

Parameters in image layer: 16508

Size of output from image layer to LSTM: 1152

Count of layers

Conv 2D: 3

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	conv_2d	3	28	leaky_relu
2	conv_2d	3	32	elu

LR-1e-06-P-11591-O-4719

Learning rate: 0.000001

Parameters in image layer: 11591

Size of output from image layer to LSTM: 4719

Count of layers

Conv 2D: 2

Max pool 2D: 0

Fully connected: 0

Detailed layers of model:

	Type	Size	Number	Activation
0	conv_2d	3	32	elu
1	conv_2d	3	39	elu