THE EFFECT OF GAME COMPLEXITY ON DEEP REINFORCEMENT LEARNING

by

Erdem Emekligil

B.S., Computer Engineering, İstanbul Technical University, 2014

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Computer Engineering

Boğaziçi University

2018

ACKNOWLEDGEMENTS

I would like to start by expressing gratitude to my supervisor Prof. Dr. Ethem Alpaydın, for giving me the opportunity to work on this thesis. This project would not have been possible without Prof. Alpaydın's support. He was always ready to provide me with feedback on my work and give me ideas whenever I found myself stuck on something.

I am grateful to my committee members Prof. Dr. H. Levent Akın and Prof. Dr. Olcay Taner Yıldız for taking the time to read and comment on my thesis.

I would like to thank both Boğaziçi University and İstanbul Technical University instructors for I owe most of my knowledge to their teachings.

I wish to express my thanks to my friend and colleague Burak Muhammed Göncü for his constant academic and moral support.

I would also like to thank my supervisor at work, Seçil Arslan and her predecessor Onur Ağın for their endless support in my master's study. I also owe special thanks to all my friends and colleagues in my office for their friendship and assistance.

Finally, I would like to thank my family for their valuable supports, love and belief in me.

ABSTRACT

THE EFFECT OF GAME COMPLEXITY ON DEEP REINFORCEMENT LEARNING

Deep Reinforcement Learning (DRL) combines deep neural networks with reinforcement learning. These methods, unlike their predecessors, learn end-to-end by extracting high-dimensional representations from raw sensory data to directly predict the actions. DRL methods were shown to master most of the ATARI games, beating humans in a good number of them, using the same algorithm, network architecture and hyper-parameters. However, why DRL works on some games better than others has not been fully investigated. In this thesis, we propose that the complexity of each game is defined by a number of factors (the size of the search space, existence/absence of enemies, existence/absence of intermediate reward, and so on) and we posit that how fast and well a game is learned by DRL depends on these factors. Towards this aim, we use simplified Maze and Pacman environments and we conduct experiments to see the effect of such factors on the convergence of DRL. Our results provide a first step in a better understanding of how DRL works and as such will be informative in the future in determining scenarios where DRL can be applied effectively e.g., outside of games.

ÖZET

OYUN ZORLUĞUNUN DERİN PEKİŞTİRMELİ ÖĞRENMEYE ETKİSİ

Derin pekiştirmeli öğrenme (DRL), derin sinir ağlarını pekiştirmeli öğrenme ile birleştirir. Bu yöntemler, selefflerinden farklı olarak, eylemleri doğrudan tahmin etmek için ham sensör verilerinden yüksek boyutlu gösterimler çıkararak uçtan uca öğrenirler. Aynı algoritmayı, ağ mimarisini ve üstel parametreleri kullanan DRL yöntemlerinin ATARI oyunlarının çoğunda ustalaştığı gösterilmiştir. Ancak, DRL'nin neden bazı oyunlarda ötekilerinden daha iyi çalışmasının sebebi araştırıl-mamıştır. Bu tezde, her oyunun karmaşıklığının bir dizi etkenle (arama alanının büyüklüğü, düşmanların varlığı/yokluğu, ara ödülün varlığı/yokluğu vb.) tanım-landığı ve oyunların öğrenme hızlarının ve başarılarının bu etkenlere bağlı olduğu öne sürülmüştür. Bu amaca yönelik olarak basitleştirilmiş Labirent ve Pacman ortamları kullanılarak bu etkenlerin DRL'nin yakınsaması üzerindeki etkisini görmek için deneyler yapılmıştır. Bu çalışmanın sonuçları DRL'nin nasıl çalıştığını daha iyi anlamak için bir ilk adımdır ve gelecekte DRL'nin etkili bir şekilde uygulanabileceği senaryoları belirlemekte bilgilendirici olacaktır.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

a_t	Action at time t
E^t	Sum of squared error loss
$Q(s_t, a_t)$	Action value function
r_t	Reward at time t
r^t	True value of instance t
s_t	State at time t
$V(s_t)$	Value of state s_t
$v_{k,j}$	Weight between hidden unit j and output unit k in a MLP
$w_{j,i}$	Weight between input unit i and hidden unit j in a MLP
w_i	Weight i of a perceptron
x^t	Input of instance t
y^t	Prediction value of instance t
z_k	Value of hidden unit k in a MLP
γ	Discount factor
ϵ	Exploration probability of an ϵ-greedy policy
η	Learning rate
θ	Network parameters

LIST OF ACRONYMS/ABBREVIATIONS

1D	One Dimensional
2D	Two Dimensional
A3C	Advantage Actor Critic
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
DQN	Deep Q Network
DDPG	Deep Deterministic Policy Gradient
DDQN	Double Deep Q Network
DPG	Deterministic Policy Gradient
DRL	Deep Reinforcement Learning
ER	Experience Replay
GORILA	General Reinforcement Learning Architecture
GPU	Graphical Process Unit
LSTM	Long Short-Term Memory
MACE	Mixture of Actor-Critic Experts
MCTS	Monte Carlo Tree Search
MDP	Markov Decision Process
MLP	Multilayer Perceptron
NN	Neural Network
ReLU	Rectified Linear Unit
RL	Reinforcement Learning
Sarsa	State–action–reward–state–action
SGD	Stochastic Gradient Descent
t-SNE	t-Distributed Stochastic Neighbor Embedding
TD	Temporal Difference
XOR	Exclusive OR operation

1. INTRODUCTION

1.1. Machine Learning

Machine learning investigates algorithms that can learn certain patterns from data and make predictions by using these patterns. These algorithms use the data to figure out the solutions to the problems in terms of these patterns, whereas traditional algorithms cannot infer anything other than what they are programmed to do. Frequently, learning algorithms are tailored to the problems and their features are hand-crafted specifically for each task. Nowadays, popular deep learning algorithms break this tradition by learning from high-dimensional data with little intervention. The idea behind these algorithms is to stack many layers of neural processing units that learn features at different levels of abstraction and then learn how to generate the current output in terms of these features. Deep learning algorithms have proven to be very successful in computer vision [2, 3], natural language processing [4–6] and speech recognition [6, 7] problems.

Reinforcement learning is a subfield of machine learning that focuses on agents learning from their interactions in an environment. Reinforcement learning algorithms imitate living beings in learning how to respond to new conditions by generalizing from their previous experiences. It has been successfully used in mostly robotics [8, 9] and games [10, 11]. Reinforcement learning works in scenarios where agents can observe the impact of their actions in an environment through a reward evaluating the goodness of their actions at each step. Agents take actions to explore their environment and learn from the rewards they get during these trials.

1.2. Deep Reinforcement Learning

By using deep neural networks directly on the raw image, deep reinforcement learning [12] gets rid of the hand-crafted feature extraction part and achieves end-to-end

learning. This means the agent takes actions only by looking at the raw input. In their work, Mnih *et al.* [12] defined the Deep Q Network (DQN) architecture that can learn many Atari games from images of the screen relay without changing the architecture or any other hyperparameters. This is the revolutionary part of the algorithm because learning different games with classical reinforcement algorithms would have required specialized hand-crafted features for each game. The algorithm succeeds in mastering a good number of Atari games and beating human players in some of them.

Deep learning algorithms are often used as black-box learners, since their massive amount of trainable parameters cannot be easily analyzed and their meaning cannot be fully understood. However, there is continuing research to understand how deep neural networks work by visualizing its parameters [13, 14]. Mnih *et al.* [12] have also provided a t-SNE embedding [15] which shows similar values for similar situations of an agent while playing the Space Invaders game. However, this visualization does not show the relationship between games with similar features.

1.3. Factors that Define a Game

Our main idea in this thesis is that there are factors that define the complexity of a game and that games that may appear different at first glance may actually be similar in terms of such abstract factors, or vice versa. For instance, whether the player is the only agent or if there are other agents, possibly hostile, that can act is such a factor. They change the whole structure of a game and ensure that players have different experiences. We assume not only that there are such factors but also that the speed a game is learned, e.g., by DQN, depends on these factors. Towards this aim, we analyzed ATARI 2600 games and came up with factors that define such games. In our experiments, we define simple Maze and Pacman environments to experiment by changing the factors to see if there is a correlation between the complexity of games and the learning performance of a DQN agent. We also cluster ATARI games according to their learning curves to see if the games in same cluster are simlar in terms of factors such as gameplay, visual features and so on.

1.4. Thesis Scope

In this thesis, we take the DQN as it is and test it on a number of settings where we vary the game factors. While we mainly focus on exploring how the complexity of games affect the learning speed and quality of deep reinforcement learning, our aim is also to answer the following questions:

(i) How the learning speed of deep reinforcement learning correlate with state space complexity.

(ii) How the complexity of games can be explained in terms of high level factors.
- How these factors are correlated with the learning speed.
- How those factors can be used for clustering games and if so, how such clusters represent game complexity.

(iii) How adding new factors that change the difficulty of the game affect the performance of the algorithm.

(iv) How the learning curve of deep reinforcement learning correlate with similarity of games.

It should be noted here that our aim is not to understand how DQN learns a particular task (game) or what its hidden units or layers are doing to handle that task, but rather we take the totality of DQN (the network, learning algorithm, and hyper-parameters) as a black-box and want to see what type of task attributes affect DQN's performance and how. The settings we use is simple by design so that we can easily observe the effect of changes on DQN's convergence. We believe that such a study is informative in understanding where and how DQN, or similar approaches, can best be used, and finding out such abstract factors that define a learning task and how such factors effect learning will especially be useful when we want to use models like DQN outside of the game-playing domain.

1.5. Outline

The rest of this study is organized as follows. In Chapter 2, background information about deep reinforcement learning methods are given. A brief summary about recent advances and applications related to deep reinforcement learning is given in Chapter 3. In Chapter 4, some key features from ATARI 2600 games are extracted and analyzed to see if there is any correlation between game factors and network learning complexity. Our experiments regarding the effects of the game complexity to deep reinforcement learning are discussed in Chapter 5. Finally, in Chapter 6, conclusions are drawn and future research points are given.

2. BACKGROUND

2.1. Neural Networks

Machine learning algorithms that use neural networks with of processing units are called deep learning algorithms.

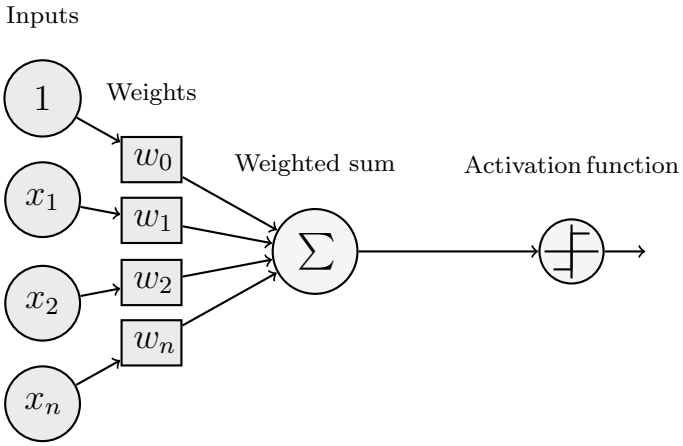


Figure 2.1. A simple perceptron.

The simplest element of a neural network is a perceptron. The inspiration behind comes from the neurons in a brain. A perceptron has input and output connections to other perceptrons. The output of a perceptron is the weighted sum of its inputs. A simple perceptron, as shown in Figure 2.1, has inputs $x_i, i = 1, ..., N$ and the associated connection weights w_i has its output y calculated as

$$y = f\left(\sum_{i=1}^{N} w_i x_i + w_0\right) \tag{2.1}$$

where w_0 is the weight of the bias unit (+1) and f is the activation function that transforms the output [16]. Activation functions such as tanh, sigmoid or ReLU, introduce non-linearity to the network and is important in capturing the significant transformations of the input.

The weights are learned by optimizing a loss or error function that compares the ground truth of the data instance and its true value. For a training set $\{x^t, r^t\}, t = 1...N$, the sum of squared error loss function is defined as:

$$E^t = \sum_{t=1}^{N} \left(r^t - y^t\right)^2 \tag{2.2}$$

Generally, optimization is done by stochastic gradient-descent (SGD) that minimizes the loss by tuning parameters in the negative direction of its gradient. To update the weight of a perceptron, the difference between the true value r and the prediction value y for instance t is multiplied with the input x_i, as well as a learning rate η:

$$\Delta {w_i}^t = -\eta \frac{\delta E^t}{\delta {w_i}^t} = \eta(r^t - y^t)x_i^t \tag{2.3}$$

Simple operations such as AND/OR can be modeled using a single perceptron. However, to model more complex operations, like a XOR, multiple perceptrons are needed to be organized in layers. These models are called as multi layer perceptrons (MLP). The hidden layers between the input and the output contain multiple perceptrons which transform the input in consequent layers until we get to the output. For example, a MLP with one hidden layer (Figure 2.2) has connection weights v_i between the hidden and the output layers, that transform the hidden values z_i. Similar to the perceptron, a MLP is trained by minimizing the loss, back propagating the error from the output through the hidden units and tuning parameters in all layers, using chain rule:

$$\frac{\partial E^t}{\partial {w_{h,j}}^t} = \frac{\partial E^t}{\partial {y_i}^t} \frac{\partial y_i^t}{\partial {z_h}^t} \frac{\partial {z_h}^t}{\partial {w_{h,j}}^t} \tag{2.4}$$

If there are many hidden layers in a network architecture, it is called a deep neural network. Calculating the parameters of deep neural networks takes much more time and effort as the number of layers increase.

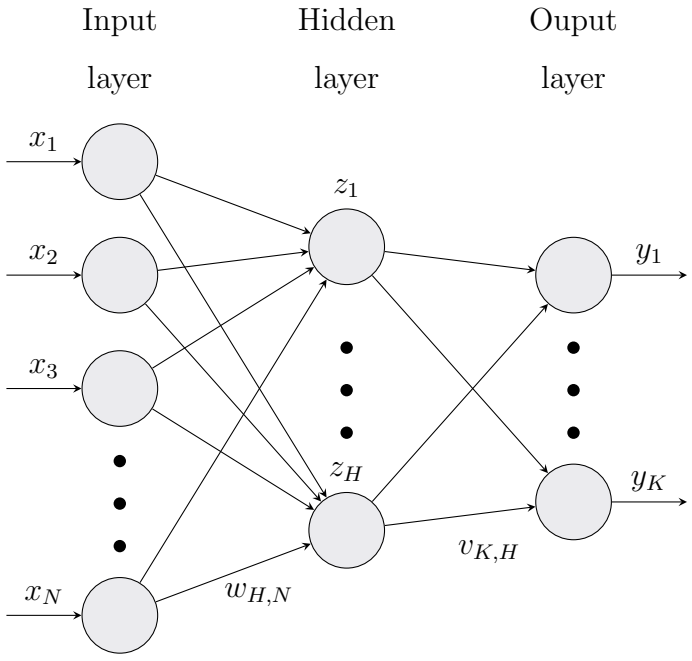


Figure 2.2. A multi layer perceptron with one hidden layer.

2.2. Convolutional Neural Networks

In a fully-connected network, all the nodes between consequent layers are connected. However, in the special case of a convolutional neural network, a local convolution operation is done to calculate the values in the next layer. Since convolution operation is a weighted sum operation over a sliding window, only the nodes corresponding to the window will be used in calculating a certain node. This decreases the total number of connections in a network. Moreover, using a window for such calculations forces the network to focus on local features. For example, the LeNet-5 architecture [1] (Figure 2.3) is defined for optical character recognition and has 32×32 handwritten number image inputs and each convolution window specializes on extracting particular local features like edge or corner.

In most cases, a subsampling operation that takes input over a certain area and reduces that to a single value, is done after a convolution operation. If the subsampling is be calculated by taking the maximum value in the area, it is called max pooling, or else average pooling is used by taking the average. Thus, the network is adjusted automatically to extract the most powerful information from the convolutional parameters.

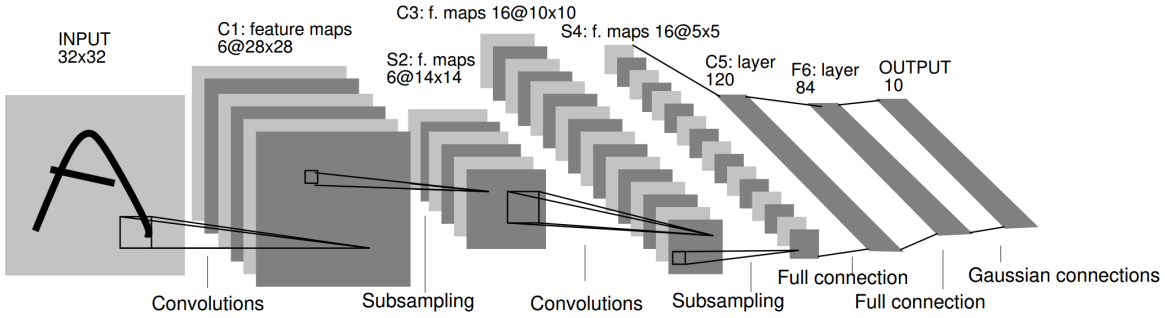


Figure 2.3. LeNet-5 Convolutional Neural Network architecture [1].

Also, the computational complexity decreases since the data reduces e.g. by 75% after a simple 2×2 max pooling operation.

The revolutionary advantage of deep learning comes from the convolution operations' support for 2D data. Most of the applications tend to exploit this by giving only the images as input and let the network learn the ideal features, which cannot always be known in advance and hence cannot be easily handcrafted. Some computer vision tasks like object detection or face recognition can be easily solved using deep learning and better results are achieved. However, though there is significant research [17–20], the parameters learned by deep neural networks are not easy to interpret, so most of the time deep learners are used without understanding what they do.

2.3. Reinforcement Learning

Supervised learning uses labels that are the desired outputs. Providing them may be costly and in some cases having these kind of supervised data is impractical. Instead of using the desired output providing a supervision, reinforcement learning uses data gathered while the agent itself explores its environment. The agent (decision maker) uses its sensors to observe the environment and learn from it by using its actuators to take actions. In the reinforcement learning setup (see Figure 2.4), the agent takes actions that change the environment and also after each action, it receives a reward information that determines if it was a good action or not.

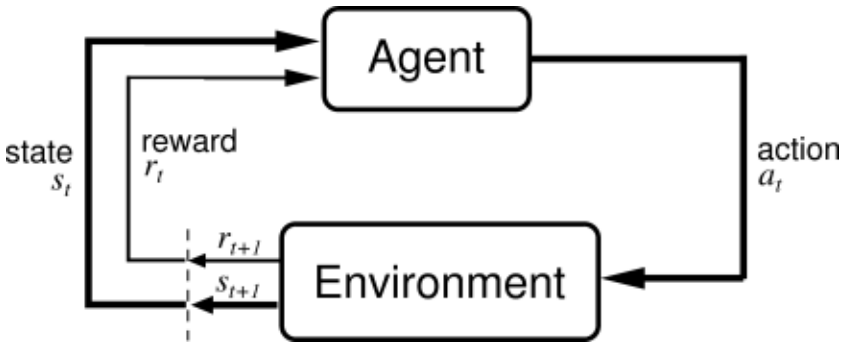


Figure 2.4. Reinforcement learning setup.

Using its state information s_t at time t, the agent takes an action a_t to observe its next state s_{t+1} and a reward r_{t+1}. The reward is given for that action by the environment since the agent changes it. The reward could be positive or negative according to the impact of the action taken. For instance, in self-driving car example, the agent (car) gets a positive reward for stopping at the traffic lights and a negative reward for hitting a pedestrian. This setup can be modeled with a Markov Decision Process (MDP) [21].

MDPs are used for modeling environments where the outcomes of actions are random (stochastic). MDPs use the probability of being in state s_{t+1} when action a_t is taken in state s_t as $P(s_{t+1}|s_t, a_t)$. These probabilities can be calculated using dynamic programming or reinforcement learning. One advantage of reinforcement learning algorithms is that they can solve problems that exact methods cannot solve due to high time complexity. It should be also noted that MDPs have the Markov property that restricts the state probabilities to depend only on the current state s_t and not on previous states $(s_{t-1}, s_{t-2}, ...)$.

Reinforcement learning algorithms try to estimate the state probabilities by learning a value function. The value of a state, as it can be understood from its name, indicates how good that state is in terms of reaching the goal. The value is calculated by summing up all future rewards but discounting them in such way that distant rewards have less importance.

In some problems the agent can observe the whole environment while in other problems the agent has limited perception over the environment. For example, chess has a fully observable environment because the agent can observe the whole board, whereas in self-driving, observation is partial since the car can only sense its immediate surroundings.

2.4. Q-learning

Q-learning is a reinforcement learning algorithm that estimates the value of each action at each state [16]. Since action values denote how good that action is in getting us to the goal, they show which action to take in any state. The $Q(s_t, a_t)$ action-value function is defined for each action a_t taken in each step t and state s_t. After taking the action a_t in state s_t, an immediate reward r_{t+1} will be received and the resulting next state s_{t+1} will be observed. We assume the non-deterministic case where the reward at the next state are sampled from probabilty distributions $p(r_{t+1}|s_t, a_t)$ and the next state from $p(s_{t+1}|s_t, a_t)$. At that step, an update over Q action-value function is defined as

$$Q\left(s_t, a_t\right) = Q\left(s_t, a_t\right) + \eta\left(r_{t+1} + \gamma \underset{a_{t+1}}{max} Q\left(s_{t+1}, a_{t+1}\right) - Q\left(s_t, a_t\right)\right) \tag{2.5}$$

where η is the step size and γ is the discount factor that decreases over time, so that updates in the beginning have higher impact on the Q values and later updates are considered as fine-tuning. By discounting the value of the next action through multiplying with γ, the action values of the states that are closer to the reward will have a greater effect than the states that are distant. Therefore, the algorithm can distinguish between a long action sequence and a short action sequence, thus gaining ability to predict the actions which will get the agent to the reward more quickly.

Temporal difference (TD) learning [22] learns $V(s)$ values instead of $Q(s, a)$, $V(s_t) = \underset{a_t}{max} Q\left(s_t, a_t\right)$. The value of a state s_t at time $V(s_t)$ can be estimated by a single backup as done in Q-learning, by using the value of the next state and the

reward received by transitioning to next state:

$$V(s_t) \leftarrow V(s_t) + \eta \left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right) \tag{2.6}$$

where γ is the discount term and η is the step size. Methods that use this approximation are called TD methods because they use the difference between the current and the next state estimates, not actual values.

To estimate the Q values of all possible action-state pairs in the environment, the agent must explore it. Exploration allows us to take samples from $p(r_{t+1}|s_t, a_t)$ and $p(s_{t+1}|s_t, a_t)$. The way to exploration is to pick actions using a policy that involves some randomness. The ϵ-greedy policy does that by keeping an ϵ value that degrades over time. For example, if the ϵ is 0.2, that means with 0.2 probability the agent will randomly select a new action, and with 0.8 probability the agent will choose the best action from $Q(s, a)$. The pseudocode is given in Figure 2.5.

Initialize all $Q(s, a)$ values
for all episodes **do**
  Initialize state s
  repeat
    Choose action a using ϵ-greedy policy
    Take action a, observe reward r and new state s'
    $Q(s, a) \Leftarrow Q(s, a) + \eta \left(r + \gamma \underset{a'}{max} Q_t\left(s', a'\right) - Q_t\left(s, a\right)\right)$
    $s \Leftarrow s'$
  until s is a terminal state
end for

Figure 2.5. Pseudocode of Q-learning algorithm.

Q-learning is an off-policy method which uses the value of the best next action when updating action values. On-policy methods like Sarsa and actor-critic, use an ϵ-greedy policy to also select the next action and use its value when updating.

2.5. Deep Q-learning

When s and a are discrete and there are few possible values, $Q(s,a)$ can be stored in a table. Deep Q-learning replaces $Q(s,a)$ action value with a regression network [12] that we denote by $Q(s,a;\theta)$ whose inputs are s and a, and θ are the network parameters. Such an approach combines Q-learning and deep neural networks by defining a loss function that has been derived from the Q-learning update step:

$$L = \left(r_{t+1} + \gamma \underset{a'}{max} \hat{Q}\left(s_{t+1}, a'; \theta^{-}\right) - Q\left(s_t, a; \theta\right)\right)^2 \tag{2.7}$$

where θ^{-} are the parameters of the target network that has been cloned from the actual network and $\hat{Q}$ is the output of the target network. In the end-to-end scenario, as we have in arcade games, s is the raw source image.

In addition to the basic change of action value function, there are two improvements performed over Q-learning algorithm. Firstly, experience replay technique [24] is adapted to the algorithm. Each of the action-state-reward transitions (experiences) are stored in memory and updates are done using randomly sampled mini-batches from this memory. This modification forces the network to learn without specializing too much on the last steps and breaks the correlations between samples. It also increases the learning stability since the behaviour distribution is smoothed over different previous states. Moreover, experience replay increases data efficiency because each experience is used in more than one batch update.

The second improvement is that the network is cloned every C steps to a target network that is used for generating the targets when updating. Thus, updates are done in a delayed manner and oscillations in the Q-values are prevented making the algorithm more stable. The pseudocode of Deep Q-learning algorithm is given in Figure 2.7. Just like Q-learning, the DQN algorithm also explores the environment using an ϵ-greedy policy.

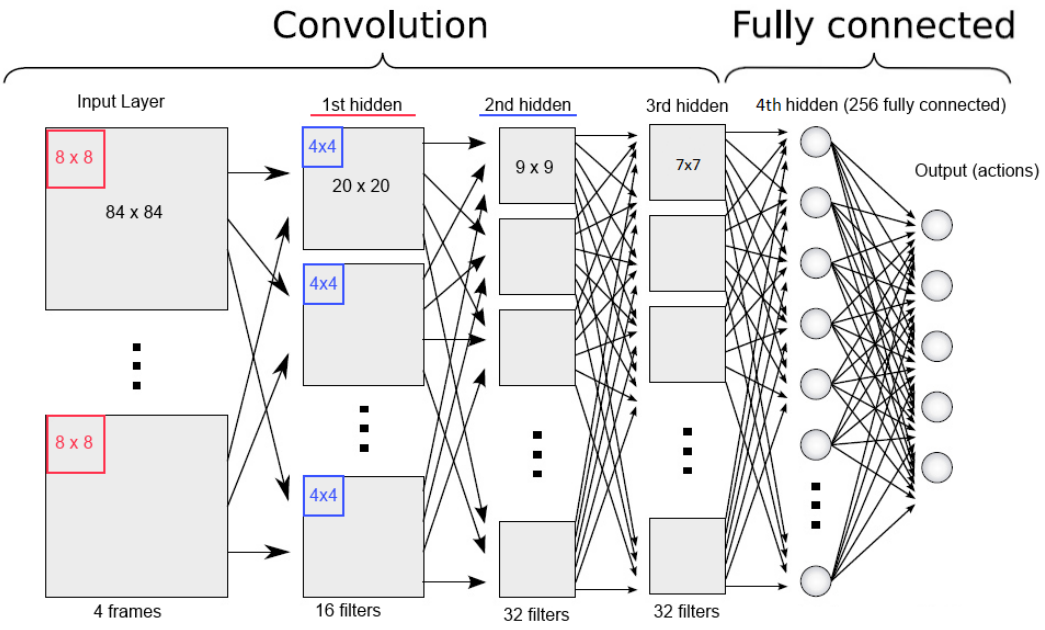


Figure 2.6. Deep Q-learning network architecture [23].

The DQN architecture (see Figure 2.6) takes four subsequent grayscale frames, each sized 84×84, in its input layer. The first three hidden layers convolve their input with 8×8 sized 32 filters, 4×4 sized 64 filters and 3×3 sized 64 filters respectively. These convolutional layers are followed by a fully connected layer with 512 units. Each hidden layer has an activation of rectified nonlinearity (ReLu). The output layer is also fully connected to the previous layer and they correspond to the possible actions whose number varies between 4 and 18 for different games. The details of the network architecture are given in Appendix E.

We have introduced Stochastic Gradient Descent (SGD), the most frequently used optimization algorithm for neural network training, in Section 2.1. However, Mnih *et al.* decided to use RMSProp [25] in their work, justifiably so because SGD tends to be slower than its alternatives. RMSProp is an adaptive learning rate method that has been built on another optimization technique called Adagrad [26].

A set of 49 different Atari 2600 games are tested with the trained DQN agent. While training on these games, the algorithm, the network architecture and the hyper-parameters of the learning algorithm are kept unchanged. DQN agents are shown to

Initialize replay memory D to capacity N

Initialize action value function Q with random weights θ

Initialize target action value function $\hat{Q}$ with random weights $\theta^- = \theta$

for episode = 1,M **do**

  Initialize sequence s_1 and preprocessed sequence $\phi_1 = \phi(s_1)$

  for $t = 1, T$ **do**

    With probability ϵ select random action a_t

      otherwise select $a_t = argmax_a Q\left(\phi(s_t), a; \theta\right)$

    Execute action a_t and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

    Sample random minibatch of transition $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ w.r.t. θ

    Every C steps reset $\hat{Q} = Q$

  end for

end for

Figure 2.7. Pseudocode of the Deep Q-learning algorithm [12].

play better than humans in 29 of 49 Atari 2600 games, achieving more than 75% of human score [12].

3. RELATED WORK

3.1. Recent Applications of Deep Reinforcement Learning

Training real world applications with DRL is sometimes not feasible, because training deep neural networks with many layers requires millions of data samples, which are not always easy to acquire. For example, Levine *et al.* [27] pre-trained CNNs in a supervised manner with the image and position data gathered from a robotic arm. Training such network only with reinforcement learning methods would have been impractical since controlling a robot arm takes too much time. After pre-training, they used RL along with guided policy search to optimize the network. The resulting agent uses only the raw image input to fully control the robot arm and is shown to work for certain control tasks.

Similar to the robot arm example, having a human expert play the Go board game against a learning agent is not feasible, since it would take too long for the agent to learn the game. Therefore, Silver *et al.* [28] pre-trained their neural network with supervised data taken from previous expert games and then used reinforcement learning to further improve by having the agent play against itself. After these training steps, the network was able to beat Pachi [29], one of the strongest Go program, winning 85% of the games. However, this well-trained policy network was not sufficient to beat a human expert and they introduced a value network to evaluate the current state of the board and trained it in the same way as the policy network. Both networks are used together with Monte Carlo Tree Search (MCTS) [30] to decide better actions by expanding and evaluating results of possible actions recursively. With these developments, the AlphaGo program was able to beat Lee Sadol, the player with the second best Elo rating at the time.

AlphaGo has achieved what everyone had previously considered to be infeasible. Nevertheless, the algorithm has some points that can be improved. Its value and policy

functions are implemented as two distinct networks which are trained and maintained separately. A third, relatively weak and fast rollout network, is also used instead of the main policy network after a certain depth when searching with MCTS, since the policy network is too slow to perform many simulations. The policy network is trained with the aid of supervised human gameplay data up to some point and it is later improved with RL self-play. AlphaGo Zero [31] that is trained using only RL self-play, learns the same gameplay strategies of human experts. Throughout the training, it has been observed that the agent quits using some of these human strategies, developing some new strategies, thereby surpassing human experts and AlphaGo. This shows that the newly developed strategies might be better than the existing human strategies and these new strategies might be used by human players in the future. In AlphaGo Zero's algorithm, the value and the policy networks are merged into one network and there is no additional rollout network. Unlike the previous version of the algorithm that uses many hand-crafted features as state information along with the board status, the improved algorithm only takes the current board status and the history of the last seven moves as state information. With these improvements, the algorithm becomes more generic and can be adapted to other games. In their recent work, Silver *et al.* [32] has introduced AlphaZero, which uses the same algorithm as AlphaGo Zero, but in a more generic way. Only the rules of the game are given as domain knowledge to the algorithm and no other hand-crafted features are used when training the network. AlphaZero is trained with RL self-play for 24 hours and it learns to play Chess and Shogi better than the best program for each game.

DRL is also used for developing agents that simulate animal movements on different terrains. Peng *et al.* [33] combined DRL with their previous work [34] lessening the need for handcrafted feature extraction. Their neural network architecture is quite different from other examples: 1D terrain features are passed through three convolutional layers and the output of these convolutional layers together with the character features are given as input to a fully connected layer. Finally, the network branches into two, predicting the action and the value for each actuator. Updates are done using the Mixture of Actor-Critic Experts (MACE) algorithm, which they have introduced. As can

be understood from its name, different movements are learned by different actor-critic pairs. Although this work is a good 2D simulation of animal movements, adapting this work to a real robot would be hard since a well-defined upcoming terrain data needs to be given to the agent by the environment. Providing a real robot with a well-defined terrain data is a whole other problem, which could also be solved by using another neural network that extracts the terrain data from camera images or other sensors.

DQN has been shown to work in high-dimensional state spaces and simple discrete action spaces, by using a NN as a function approximator and optimizing the policy by using Q-learning. However, real world control tasks like cartpole, locomotion and car driving have high-dimensional continuous action spaces. Training a DQN with known methods by discretizing a high-dimensional continuous action space is impractical since the number of actions increase exponentially, making learning harder and also there is loss of information in the discretization process. Lillicrap *et al.* [35] address this problem by introducing a model free, off-policy actor-critic algorithm that they named deep deterministic policy gradient (DDPG) which stems from the work of the deterministic policy gradient (DPG) algorithm by Silver *et al.* [36] that deals with high dimensional problems by using deep function approximators. DDPG applies DQN directly to the continuous state problem and solves the stability issues of DQN in continuous states by not modifiying the weights of the algorithm directly, but instead creating a copy of the actor-critic and using these to gradually transition their learned values to the main policy.

3.2. Double DQN

One of the problems of Q-learning is the overestimation of action values. This is caused by the algorithm's use of the maximum action value as the expected action value. Hasselt [37] introduces Double Q-learning, which is a double estimator method to overcome these overestimation problem by underestimating the action values. Two action value functions are defined and at each step, a randomly selected one is used for selecting the action with the maximum value, while the other one's value for that

action is used when updating. Since both of the action value functions are trained on the same problem with different experiences, the bias that causes overestimation is eliminated.

In his recent article [38], Hesselt combined the idea of Double Q-learning with Deep Q-learning and introduced the Double DQN (DDQN) algorithm. Deep Q-learning already has a target (backup) network which could be used as a second action value function. The actual network is used for selecting the action with the maximum value, whereas the target network is used for estimating that action's value. To understand better, we can compare Deep Q-learning's target

$$y_j = r_j + \gamma max_{a'} \hat{Q} \left(\phi_{j+1}, a'; \theta^- \right) \tag{3.1}$$

with Double DQN's target

$$y_j = r_j + \gamma \hat{Q} \left(\phi_{j+1}, argmax_{a'} \left(\phi_{j+1}, a'; \theta \right); \theta^- \right) \tag{3.2}$$

Only with this simple change in the algorithm and no other improvements, Double DQN resulted in more stable learning and acquired better results than DQN on most Atari 2600 games.

3.3. Prioritizing the Experience Replay

We have already mentioned the two problems that Experience Replay (ER) solves. DQN updates are done with SGD, which assumes the data to be independent and identically distributed. While consequent correlated updates disrupt this assumption, ER resolves this problem by picking experiences uniformly from a replay memory. Moreover, when it is used with mini-batch updates, experiences are used more than once. Therefore, it also takes measures against rapid forgetting of important experiences, since an experience is used many times.

However, picking experiences uniformly from the ER memory has some flaws. Less important experiences might be replayed redundantly while important ones might be forgotten. In their work, Schaul *et al.* [39] address this problem by defining different methods that prioritize the picks from replay memory and show that prioritizing achieve major improvements over uniform replay. These methods use temporal difference (TD) error that measures the unexpectedness of the experience to prioritize the transitions. Greedily selecting only the experiences with highest TD-error performs better than uniform sampling. Errors of selected experiences are updated after they are replayed. However, TD-errors of the experiences with the highest error decrease slowly. In practice, this causes the greedy approach to work on a small subset of the whole replay memory. Schaul *et al.* introduce a stochastic sampling method that defines the sampling probability of transition i as

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{3.3}$$

where the priority (p_i) is proportional to its TD-error and the constant α regulates prioritization. This stochastic sampling method resulted in a good speed-up over uniform and greedy sampling. Furthermore, when this method is used with Double DQN, state of the art results on the ATARI environment are achieved and learning speeds are increased nearly to their double.

3.4. Asynchronous Methods

The 2D convolution operations of deep learning algorithms are implemented on GPUs, since these operations can be completed in parallel on several GPUs. However, these devices are more expensive than CPUs and not as available, especially on cloud servers. The power of cloud computing environments comes from their CPU quantity and this could not be exploited with single threaded applications. Moreover, in the case of large models that does not fit into GPU memory, the transfer bottleneck between the CPU and the GPU causes slowdowns. DistBelief [40] addresses these problems by using thousands of distributed CPU cores. It defines a central parameter server to hold

the master copy of model and updates it when gradients are received from replicas. Each replica holds a copy of the model and sends the calculated gradients of a given mini-batch to the parameter server. Also, the updated model is transferred from the parameter server to the replicas periodically. An asynchronous version of SGD is used for updates.

Inspired from DistBelief, Nair *et al.* [41] introduced General Reinforcement Learning Architecture (GORILA), a framework for DQN, that works on distributed systems. Similar to DistBelief, GORILA also has a central parameter server and multiple replicas. Multiple actors play ATARI games in parallel and store their generated data in a globally distributed replay memory. Each replica samples mini-batches from the replay memory, applies DQN to the mini-batch and sends its calculated gradients to the parameter server. The parameter server applies the received gradients using an asynchronous SGD algorithm. GORILA surpassed DQN on 41 of the 49 ATARI games, using approximately half of the training time of DQN.

Using multiple CPU systems for DRL has proven to work well with GORILA. But since GORILA needs huge distributed systems that are not so easily attainable, Mnih *et al.* [42] proposed a framework that can run on a single machine with a multi-core CPU. Calculations are done on the single CPU with a shared memory and this allows transfer of messages between cores without too much communication overhead. Each of the learners has its own environment and by using different exploration strategies with each of them, one can assume that the data generated by these learners are very diverse. The gradients calculated by actor-learners with the diverse data are applied in parallel. Since the data is less correlated than the data generated from a single environment, the need for the replay memory decreases. Therefore, on-policy reinforcement learning algorithms can be used since expreience replay is no longer used, which was the reason why on-policy algorithms have never been used with DRL. Four different versions of on-policy and off-policy reinforcement learning algorithms, named one-step Q-learning, one-step Sarsa, n-step Q-learning [43] and advantage actor-critic (A3C), are compared. The agent trained with the asynchronous A3C method on a

16-core CPU nearly quadrupled the mean score of a GPU DQN agent with the same parameters on 57 ATARI games. They also tried changing the network architecture by adding recurrency: with 256 LSTM cells after the final hidden layer of the CNN, A3C achieved more than five times of the mean DQN score.

3.5. Combining the Methods

To summarize, Double-DQN (DDQN) uses a backup network as the target action value function [38]. The actual network is used for selecting the action with the maximum value whereas the target network is used for estimating that action's value. Instead of random picks in experience replay, Schaul *et al.* [39] define different methods to prioritize the picks from replay memory and show that prioritizing achieve major improvements. Wang *et al.* [44] introduce a new architecture that combines a state value function and an action advantage function. DQN is also adapted to work on distributed [41] and multi-core CPU systems [42] using actor-critic (A3C) methods with LSTMs. Unlike previous methods, Distributional DQN [45] can distinguish risky actions since it models the value distributions.

The Rainbow method [46] unites these developments over DQN and show that these different methods achieve better results when they are combined. In their work, Hessel *et al.* replace the distributional loss defined in [45] with the multi-step loss [43]. The multi-step distributional loss is adapted to the target network of DDQN that evaluates actions. Experience replay is prioritized using the multi-step distributional loss instead of the TD error. Instead of the original network architecture, the dueling network architecture that also models the value distributions is preferred.

4. ANALYSIS OF ATARI GAMES

Though the DQN algorithm works well, we cannot interpret what elements of the game are learned by the network. We tried to examine the parameters of the deep neural network layers by creating a visual interface; the details of this interface are given in Appendix A. This visual interface shows the weights between the layers and their forward-pass activations (outputs) for each frame as the gameplay continues. The learned weights change for each ATARI game even if the network architecture stays same, and it is not easy to determine which part of the network learns which features of the games by examining the network weights and activations.

It is our contention in this thesis that games that seem very different at first glance may be very similar at a more abstract level, and beyond their immediate facade games can be defined in terms of a number of factors; furthermore it is these factors that define the complexity of a game and the best strategy to play it well. Specifically, we take DQN as the game-learner and consider games that can be learned by DQN, simplified versions of the ATARI 2600 games, where we can define and test the effect of such factors.

4.1. Factors That Define A Game

A neural network extracts high level features in its last layers. These high level features check for complex patterns that have important effect on the output. For instance, the last layers of a CNN that is trained for face recognition task, is expected to capture complex shapes like eyes, mouths, or even facial expressions. Similarly, DQN is expected to capture some features that define games in an abstract manner. For example, we found that in the game Seaquest, the oxygen level, which is the time constraint of the game, is captured by some units in the last convolutional layer of the network, as shown in Figure 4.1. The units at the bottom row of this layer indicate the oxygen level. The leftmost unit get activated as the oxygen level drops and rightmost

unit gets activated once the oxygen is filled. We believe that such abstract features can be mapped to some key factors that point out the basic structure of a game, and furthermore that the presence or absence of these factors affect how fast and how well the game is learned by DQN.

Figure 4.1. The activation units that keep track of oxygen level in the Seaquest game. Some of the 64 units in the last convolutional layer keeps track of the oxygen level in the Seaquest game.

There are plenty of literature in game design research on the various characteristics of games and how they effect playability for humans and AI. Elias *et al.* [47] define charasteristics of games and explore their effects on human players. Anderson *et al.* [48] compare performances of several tree based AI algorithms such as MinMax, MCTS and A* on seven different games that they have created. Yannakakis *et al.* [49] define five different factors based on Elias *et al.*'s book and explain their effects on AI methods:

- Number of players. Does the game played by a single player, multiple players or a single player plays with/against computer controlled enemy units?
- Stochasticity. Does the outcome of the game determined only by the player?
- Time granularity. Is the game turn-based or real-time?
- Observability. Does the game partially observable or the player has perfect information?

- Action space size. Number of the actions the player can take.

We analyze ATARI 2600 games and try to characterize these games according to their similar elements. Some of these elements are related to the rules of games. For instance, time constraints or the presence of an enemy makes the game harder. Other factors are selected because they are related to the learning mechanism of humans and the DQN algorithm. For example, if a game depends highly on quick action, it is relatively harder for a human to master this game, whereas this should not matter for a computer program. The following factors we believe are important and first three of them are similar with the first three of the Yannakakis *et al.*'s definitions [49]:

- Does the game have any enemy units?
- Is the environment stochastic or deterministic?
- Does the game depend on reflexes?
- Does the game have any time constraints?
- Does the game speed increases over time?
- Is death (game over) an option?
- Do the shape or color of objects change?
- Does the game provide an intermediate reward?
- How many sequential actions does the game require to get to the goal?

By playing 45 ATARI games we answer these questions and generate a data matrix given in Appendix B. We apply hierarchical clustering on this data using Hamming distance, since our data consist of categorical features. The resulting dendrogram is given in Figure 4.2.

Unfortunately, clustering results are not informative. In the following sections, we group games according to their learning curves, hoping to see a correlation between these groups and the factors.

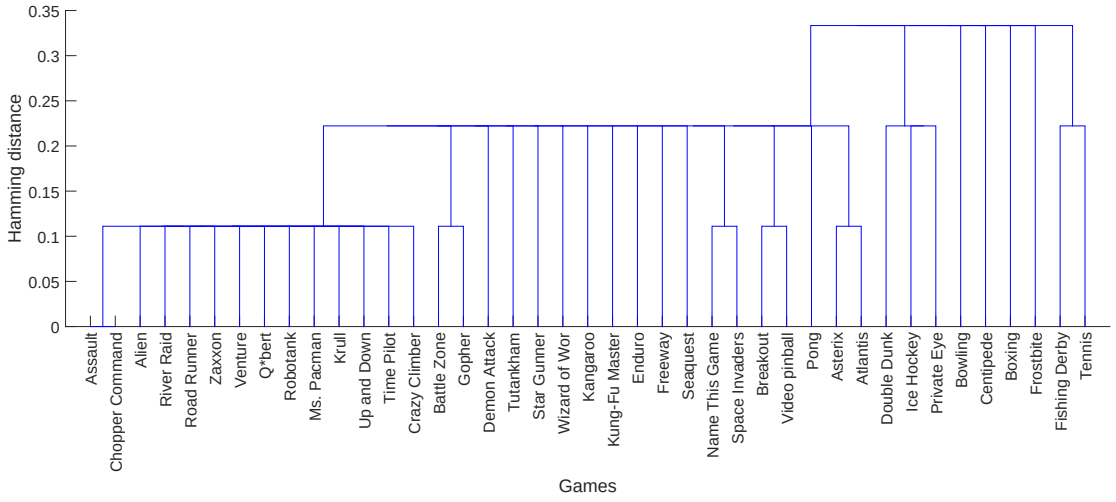
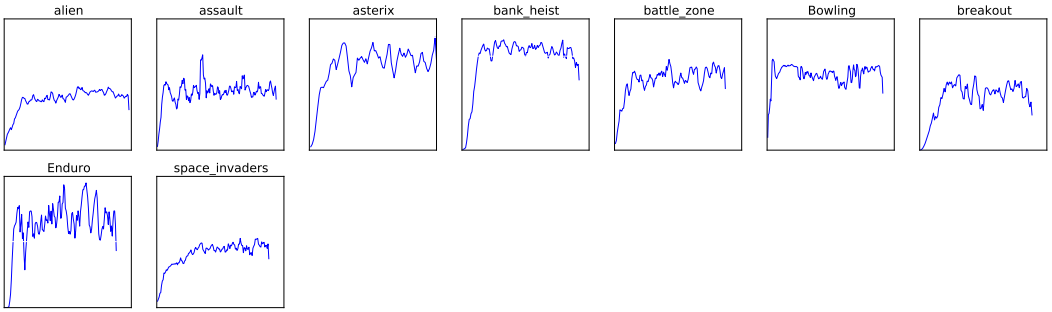


Figure 4.2. Hierarchical clustering of the games by their factors.

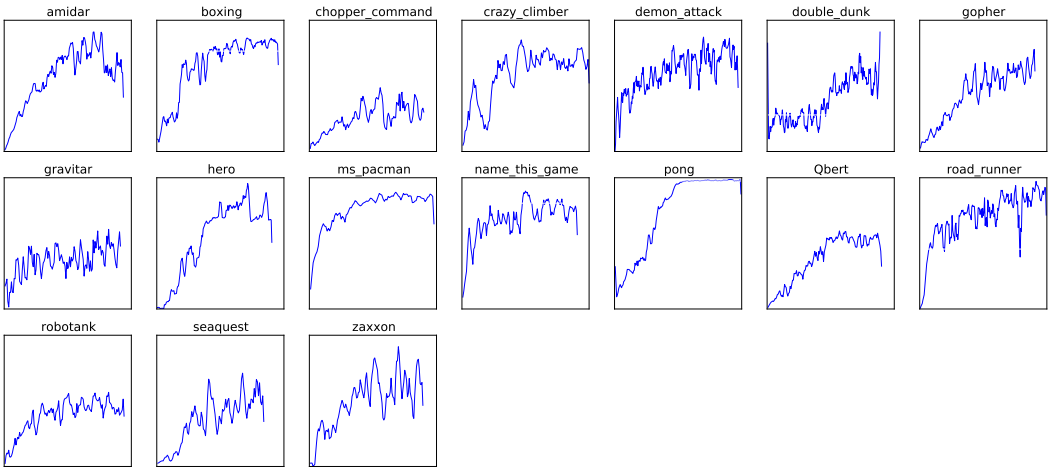
4.2. Learning Speed of DQN

We try to categorize the learning curves of DQNs that have been trained on ATARI games (see Appendix E for our DQN implementation details and Appendix F for the environment in which it runs). We seperate the learning behaviors of 45 ATARI games into four categories by manually analyzing the shapes of their learning curves. These four categories are:

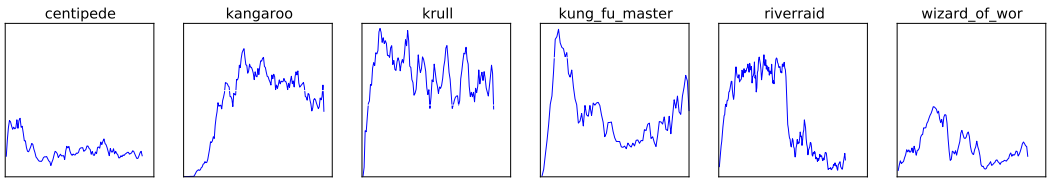
- Slowly learning over time: Most of the games fall into this category, since it is the most expected behavior. The network learns to get better at the game slowly until there is convergence.
- Rapidly learning: Suprisingly, the network learns the games in this category really fast.
- Forgetting after some point: In some games, the network seems to unlearn what it has learned up to some point.
- High variance: In these games, the network cannot converge since there are huge differences between consecutive epochs.



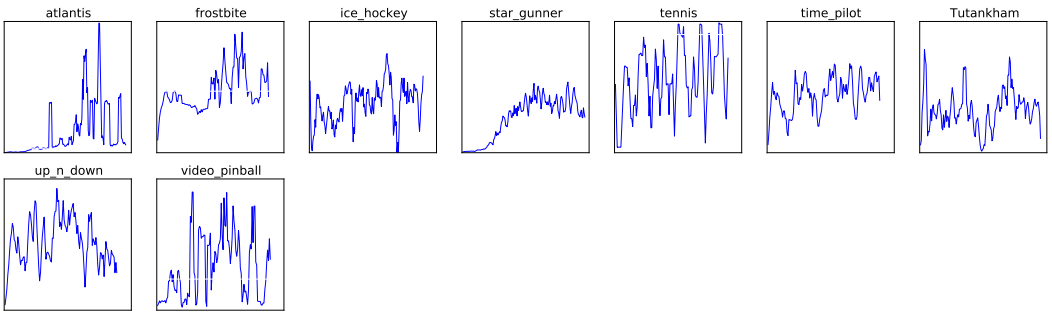
(a) Rapidly learning



(b) Slowly learning over time



(c) Forgetting after some point



(d) High variance

Figure 4.3. The four categories of ATARI games manually divided according to their network learning behaviours. Networks are trained for nearly 150 epochs and a testing epoch is run to calculate the average score at the end of training epochs.

If we compare the clustering results in Figure 4.2 with those in Figure 4.3, we cannot see a direct correlation between them. It is likely that representing ATARI games only with nine high level features is insufficient since complex environments like ATARI games should be represented with much more features. Therefore in the next section, we cluster games according their learning curves using a clustering algorithm in search of a correlation between games and their learning complexity.

4.3. Games with Similar Learning Curves

We run DQN on 45 games and for each we record the convergence of DQN in terms of average game scores. We then take 32 equidistant samples from these plots, normalize them between 0 and 1, and use average-linkage hierarchical clustering with Euclidean distance on these 45 vectors each of which is 32-dimensional (see Figure 4.4).

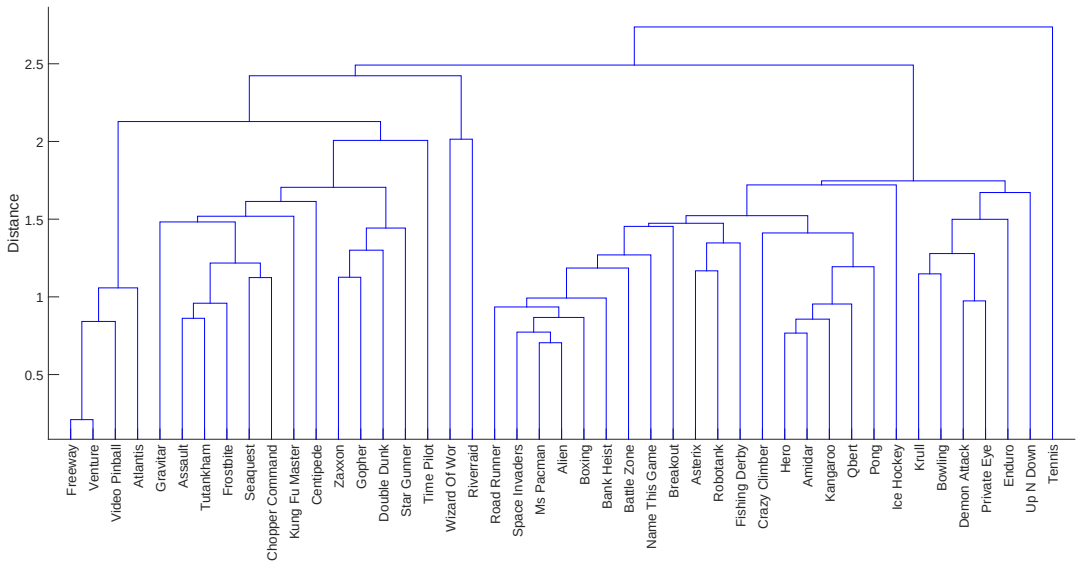


Figure 4.4. Hierarchical clustering results of ATARI games with Euclidean distance.

Games whose convergence plots are similar are placed nearby in the tree and for certain cases, we can see that actually they correlate with similarities between the games. For example, Ms Pacman and Alien, connected early on, are very similar both in terms of how DQN learns them and also in terms of how they are played. They

are very similar in many aspects such as actions, objective, rewards and so on; in both games, the agent tries to collect as many reward (food items) as possible without getting caught by the enemy and both have power-up units that temporarily give the ability to destroy opponents. Interestingly, Bank Heist looks similar to these but in Bank Heist, instead of power-up, the agent can counter enemies by bombs which changes the strategy. This difference changes the convergence of DQN and this explains the distance of Bank Heist to Ms Pacman and Alien on the tree. Wizard of Wor and Alien are games that look very similar visually, but they are played differently and hence the convergence behavior of DQN are different and they are very distant in the dendrogram despite the visual similarity.

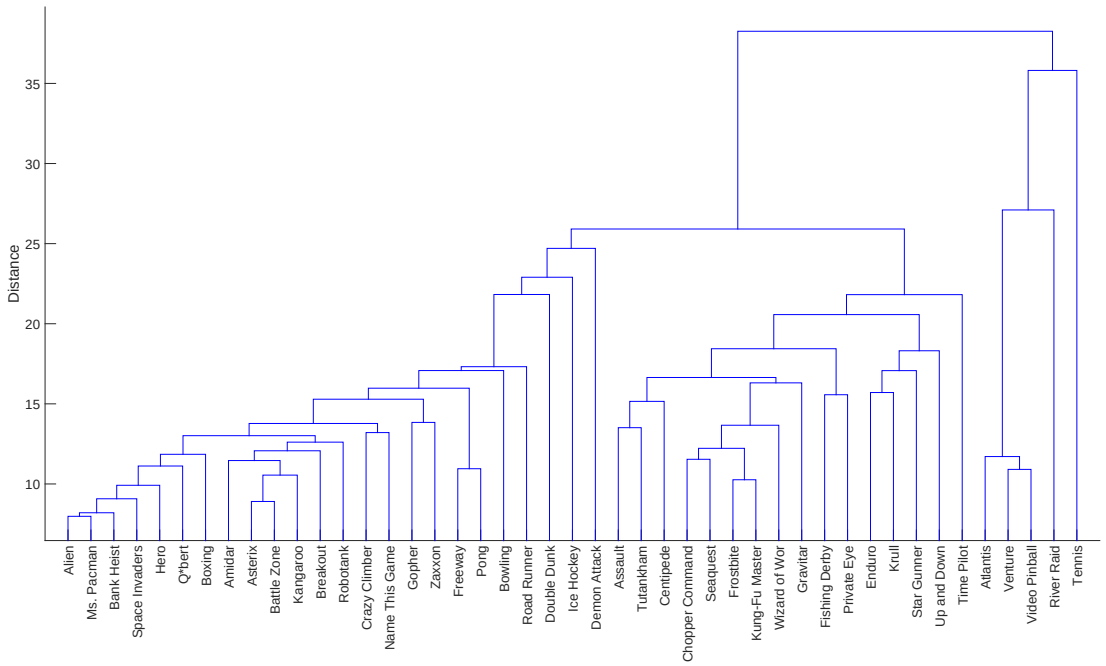


Figure 4.5. Hierarchical clustering results of ATARI games with dynamic time warping.

As an alternative, we also cluster these games without doing any sampling on the data, instead of Euclidean distance, we use dynamic time warping [50] to measure distance between vectors of different lengths, since each game runs for different number of epochs. We can see from the dendrogram of the average-linkage hierarchical clustering shown in Figure 4.5 that most games like Alien, Ms Pacman and Bank Heist stay in the same sub-tree even if the distance metric changes. In Zaxxon and Robotank, the

agent controls an airplane or a tank and the main objective is survival whether from an airplane or a tank crash. The games are played similarly and DQN convergence curves on them are similar and that is why they are not far from each other on the dendrogram. Atlantis, Tennis, Venture and Video Pinball are four games on which DQN fails to converge and that is why they are clustered immediately. Three major clusters can be seen if the dendrogram is inspected roughly. Games in these clusters are shown together with their convergence plots and sample screenshots in Appendix C.

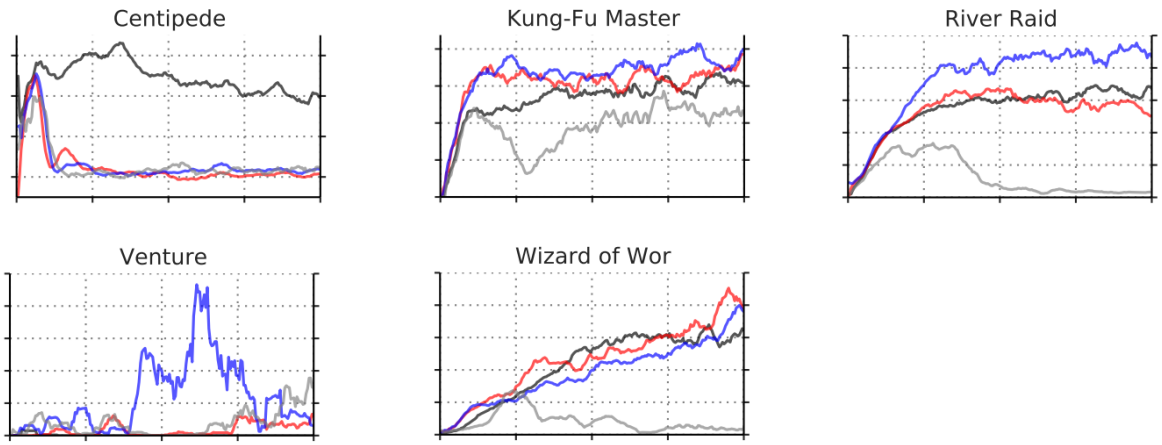


Figure 4.6. Learning curves of Centipede, Kung-Fu Master, River Raid, Venture and Wizard of Wor games for three different prioritization methods (black, red, blue) and original DQN (gray) [39].

As we see in Figure 4.6, the average reward curves of Centipede, Kung-Fu Master, River Raid and Wizard of Wor games drop after being trained for certain time. These games seem to first learn to play according to the basic gameplay mechanics, then switch to advanced gameplay tactics. While trying to increase their efficiency by learning sophisticated movements, they seem to forget some of the basic moves that are essential for gaining higher rewards. For instance, in Centipede, the objective is to shoot the centipede that is moving from the top to the bottom. When a node of the centipede gets hit, it becomes an obstacle that blocks the centipedes path. Therefore, an intelligent tactic would be shooting from edges of the gameplay area to corner the centipede between the edge and obstacles. When we watched the gameplay of the DQN agent, we saw that the agent is able to learn to track the movements of the centipede

and shoot at it. However, it tries to learn the "edge tactic" and seem to get stuck at the edge, forgetting the tracking maneuvers and unable to achieve high scores. These issues are solved in Schaul *et al.*'s work [39] that prioritizes important actions by picking transitions from memory with different probabilities instead of uniform distribution. Figure 4.6 compares the three different prioritization methods with base DQN results. Venture game cannot be learned by the DQN algorithm but it is learned up to some point with the aid of proportional prioritization (blue lines in Figure 4.6).

5. EXPERIMENTAL RESULTS

In this chapter, the aim is to generalize from our experiments on ATARI games. By selecting three of our game factors which we have introduced in Chapter 4, we define a simple maze and a simple Pacman environment, and train DQN to see their effect on learning. The features we test are:

(i) How many sequential actions does the game require to get to the goal?
- Increasing the size of the maze.
- Adding blocking walls.

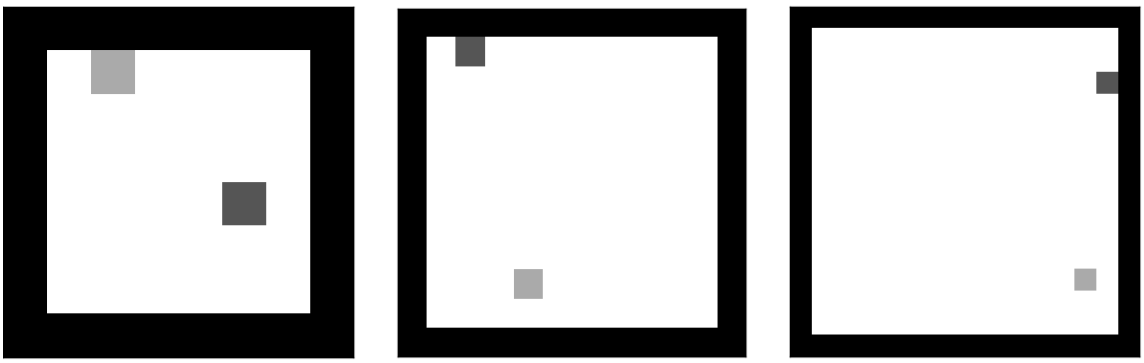
(ii) Is death (game over) an option?
- Generating hostiles that ends the game upon contact.

(iii) Does the game provide an intermediate reward?
- Giving an intermediate reward as a cue to quicken access to the goal state.

In our experiments, we keep the original DQN network, learning algorithm, and hyper-parameters of [12] since they have been proven to work in many different games. The only parameter changed is the decay rate ϵ, which is adapted to the complexity of the maze by setting it to its minimal value that allows convergence. Since our generated mazes are much smaller, they are stretched to fit 84×84; the mazes contain the outer walls so the actual playable area is one less on all four sides. Each agent is evaluated after every 250,000 training frames for 125,000 test frames and the average episode score is plotted. As we will see in the experiments, maze examples take many more epochs to converge than the ATARI games. Also, in all experiments, we did five runs with different random seeds and plot the one that best represents the average behavior. Detailed plots that show all five runs are given in Appendix D.

5.1. The Effect of the Size of the Search Space

We start by testing the effect of the maze size, which is an indication of the size of the search space: A larger maze requires longer sequences of actions. We use mazes of 8×8, 12×12 and 16×16. In our simplified experiments, a maze contains the outer walls, the agent and the target. The walls are colored white, the target is dark gray, the agent is light gray. The background is black (encoded as 0) to help the network learn faster. We run many episodes where the position of the agent and the target are randomly chosen in each episode (see Figure 5.1; the colors are inverted to save from ink). In each episode, a score of 100 is awarded when the agent reaches the target before the maximum number of allowed moves , which we define as $(\text{Width} + \text{Height}) * 10$.



(a) 8×8 maze (b) 12×12 maze (c) 16×16 maze

Figure 5.1. Example mazes of different sizes with outer walls (black), the target (dark gray) and the agent (light gray).

Because increasing the maze size increases the average number of actions required to get to the goal, as expected, and as we see in Figure 5.2, the number of training epochs it takes for DQN to converge also increases.

5.2. The Effect of Obstacles

The complexity of the path to solve the maze can be increased by adding obstacles. The agent cannot just take any path to the goal but needs to recognize and avoid the obstacles. In our experiments, we simulate this by a wall with a single gate. The positions of agent and target, as well as the location of the gate and the wall orientation

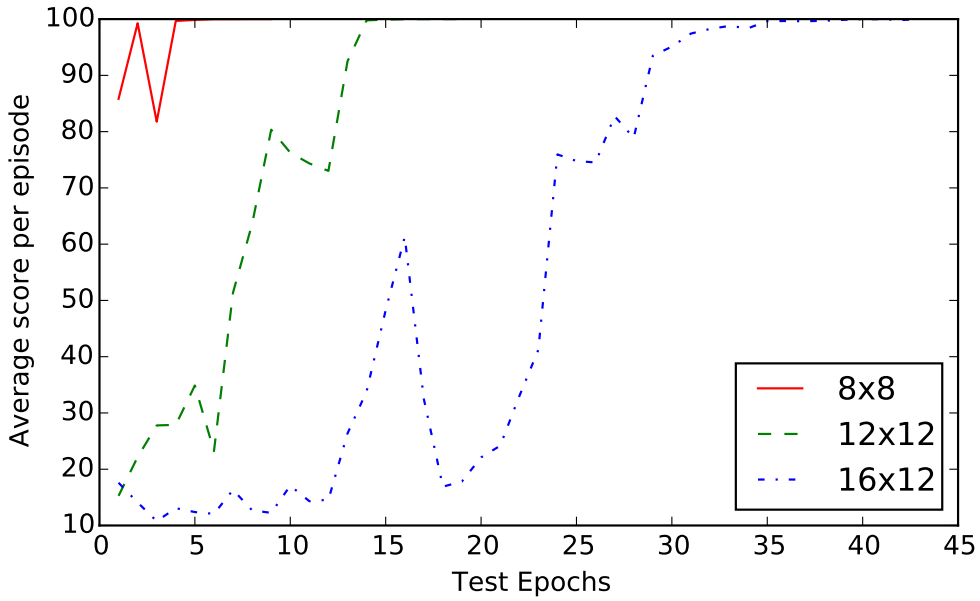


Figure 5.2. Convergence of DQN as a function of maze size. As expected, larger mazes take longer to learn.

are also randomly assigned in each episode. To make the task more complex, we also experiment with two intersecting walls that divide the maze into four playable areas connected by three randomly located gates. Three randomly generated examples are shown in Figure 5.3.

(a) Maze with a horizontal wall (b) Maze with a vertical wall (c) Maze with two walls

Figure 5.3. 12×12 maze samples with different wall structures.

We train DQN on these three setups with different obstacle structures (no wall, one wall, two walls) and three different sizes, just like in the previous experiment. In Figure 5.4, we see that because adding walls increases the path complexity and consequently the number of actions to achieve the goal, the convergence of DQN is

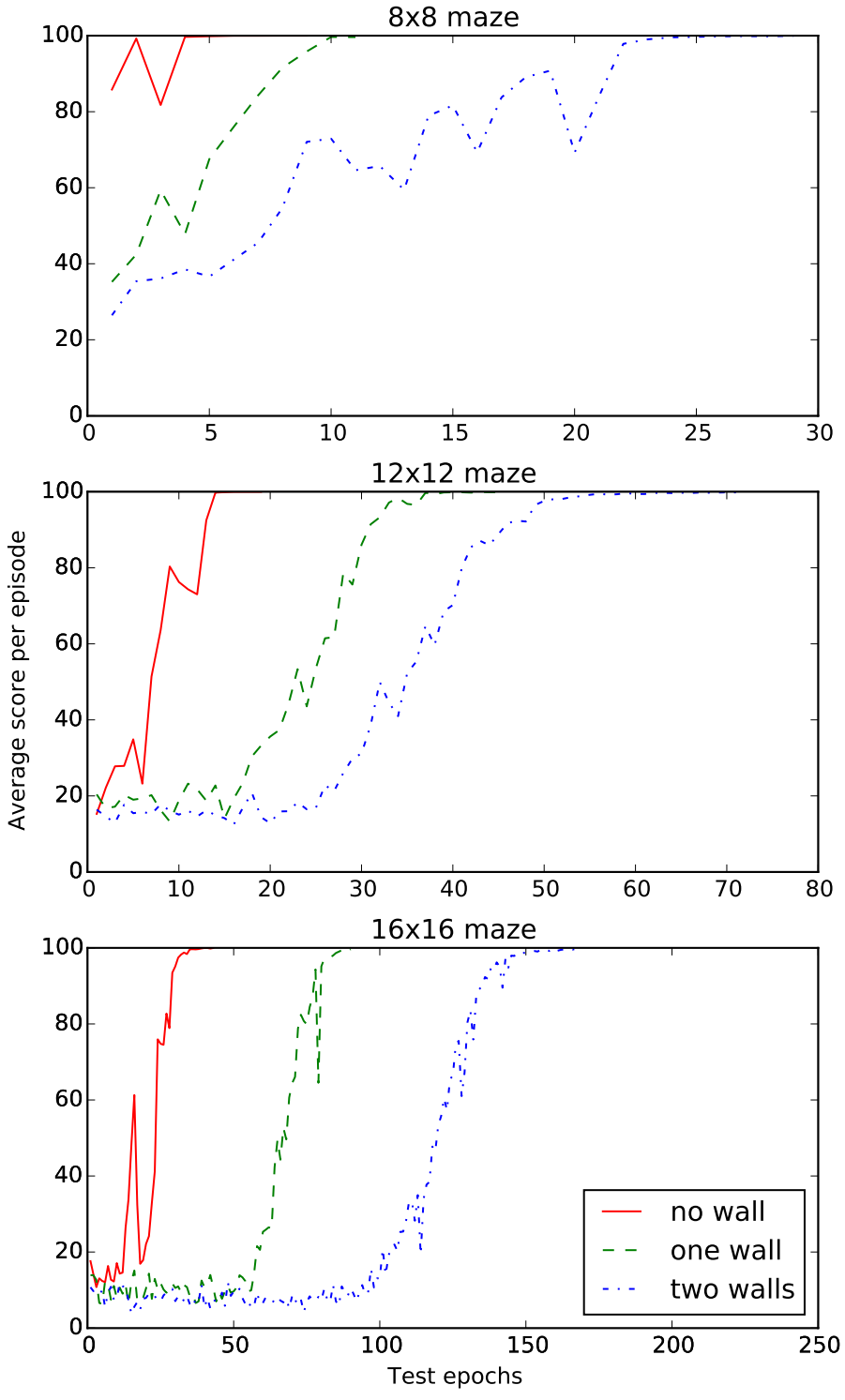


Figure 5.4. Convergence of DQN as a function of maze sizes and wall structures. With more obstacles, learning gets slower.

much slower needing more training iterations; with larger mazes, the differences get larger.

5.3. The Effect of Hostile Agents

If the game-player is not the only agent that can change the environment, the presence and actions of other agents make the task harder. In our maze experiments, we add unit-sized enemies that ends the game on contact (without any reward), to test if the network can learn to recognize and avoid them efficiently. As shown in Figure 5.5, a different gray level is chosen to encode the enemy, and the locations of these enemy units are chosen randomly in each episode.

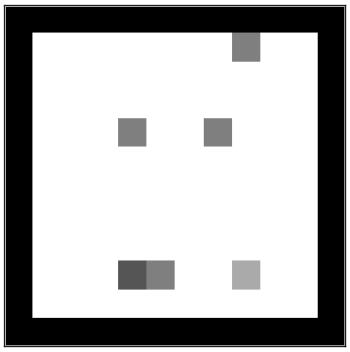


Figure 5.5. A 12×12 sample maze with outer walls (black), a target (dark gray), an agent (light gray) and four enemies (medium gray).

Our results are given in Figure 5.6, where we see that the presence of an enemy makes the task harder to learn, regardless of the maze size. The number of enemies, as long as it is nonzero, does not seem to have a drastic effect. Plots with 2, 3, 4 enemies seem to be clustered together for mazes of 8×8 and 12×12; for the maze of 16×16, we believe that the variability is due to chance. Once DQN learns to recognize an enemy and how to avoid it, and it is enough to do enough episodes with a single enemy for that, DQN can then recognize and avoid any number of enemies that it later encounters.

5.4. The Effect of Intermediate Reward

In most games, the reward is given not only at the end but also at some special intermediate state, such as destroying an enemy unit or passing through a checkpoint. Such an intermediate reward is useful in hinting the learning agent that it is on the

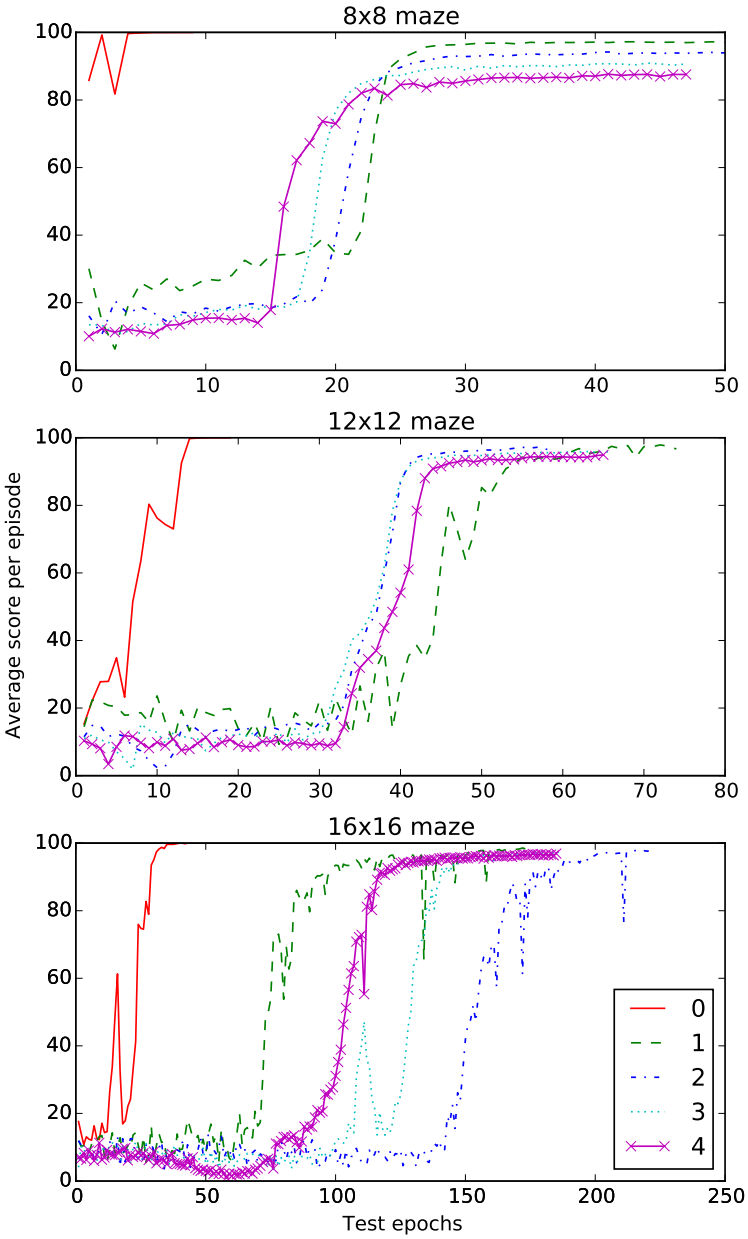
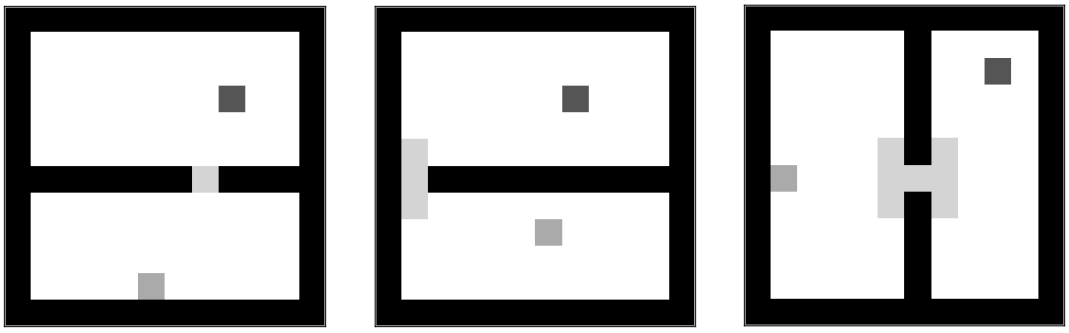


Figure 5.6. Convergence of DQN as a function of maze size and the number of enemies. It is whether there is any enemy or not, rather than the number of enemies, that slows down learning.

correct path to the goal. In our maze experiments, we implement this in the case with one wall with a gate and by giving a reward of 10 as an intermediate reward upon reaching the gate. Afterwards, if the agent achieves its goal, an additional 90 points is given to get the same total of 100. The intermediate reward is given in a color that is close to white (Figure 5.7).



(a) Maze with 1×1 intermediate reward

(b) Maze with 1×3 intermediate reward

(c) Maze with 3×3 intermediate reward

Figure 5.7. 12×12 mazes with different size of intermediate rewards.

In this experiment, we use different sized mazes all having one wall. The target and agent locations are forced to be on different sides of the wall (this was not forced in previous experiments) and the intermediate reward area is given in a different color, close to white. We test using different sizes of intermediate reward areas to check if extending the reward area increases the learning speed (see Figure 5.7). As we can see in Figure 5.8, adding an intermediate reward increases the learning speed as the maze size gets larger. In the 8×8 setting, the search space is already so small that no intermediate reward seems necessary. But especially in the 16×16 maze with its larger search space it helps and is more helpful when the intermediate reward area gets larger.

It should be noted that we also tried this experiment without any visual target, but it seemed not to converge since the agent got stuck near the gate, hoping to get more of the ten point intermediate reward.

5.5. Pacman Maze Experiments

For our next set of experiments, we use the game of Pacman, which is a more interesting maze game. We use the Pacman environment prepared for the course UC Berkeley CS188 Introduction to AI [51]. This environment provides customizable mazes and basic AI options for enemy units. We adapt the maze style of the Ms. Pacman ATARI game to this environment and select enemy units with random movement

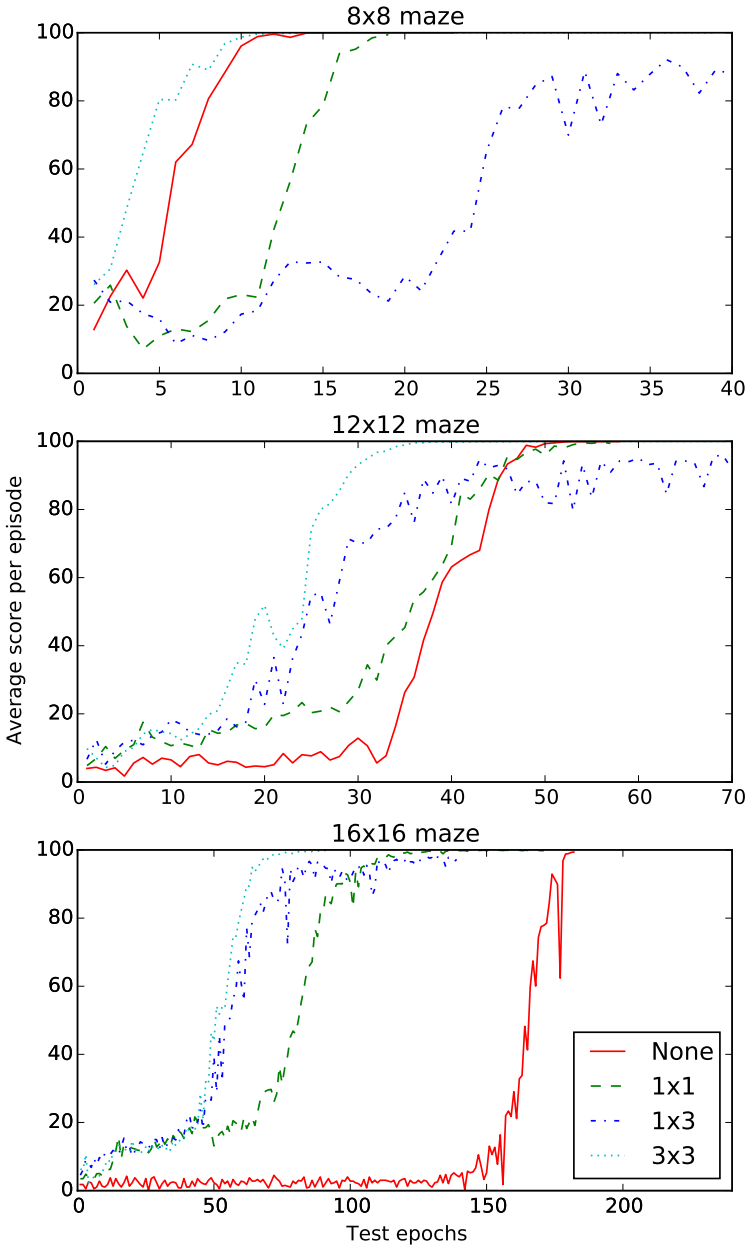
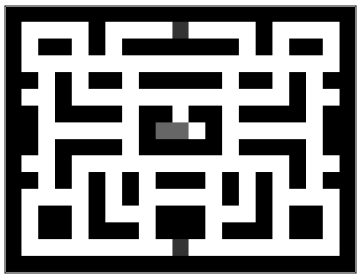
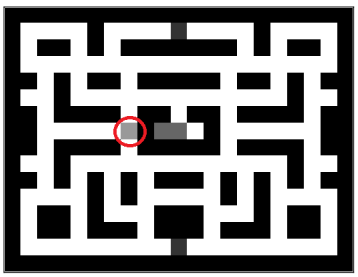


Figure 5.8. Convergence of DQN as a function of maze size and the intermediate reward area. With small mazes intermediate reward does not help, but as the maze gets larger, larger areas of intermediate reward help more.

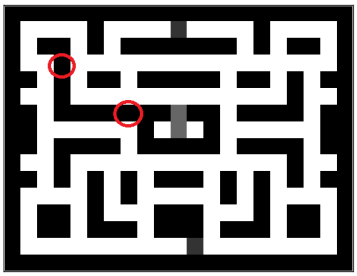
capabilities. These enemies pursue their path until they reach to the end and select their next path randomly when they are at a junction point. A contact with an enemy ends the game with -500 reward points. Intermediate reward gives $+10$ points whereas the goal gives $+500$. The agent starts from the bottom and tries to get the goal at the top, whereas the enemies spawn in the center of the maze.



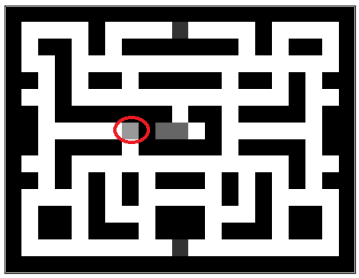
(a) Original setting



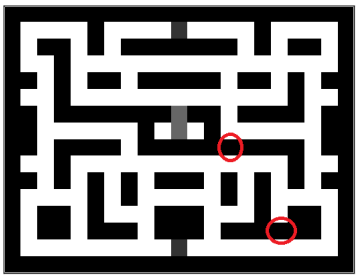
(b) With intermediate reward



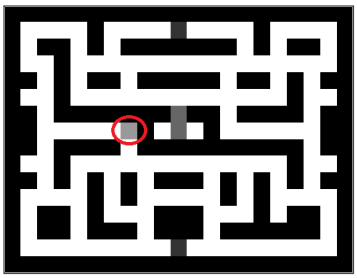
(c) Harder left path



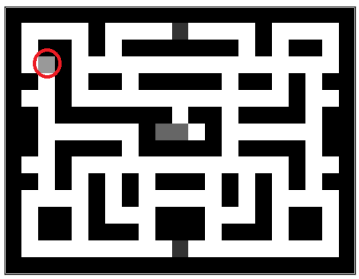
(d) Harder left path with intermediate reward



(e) Harder paths on both sides



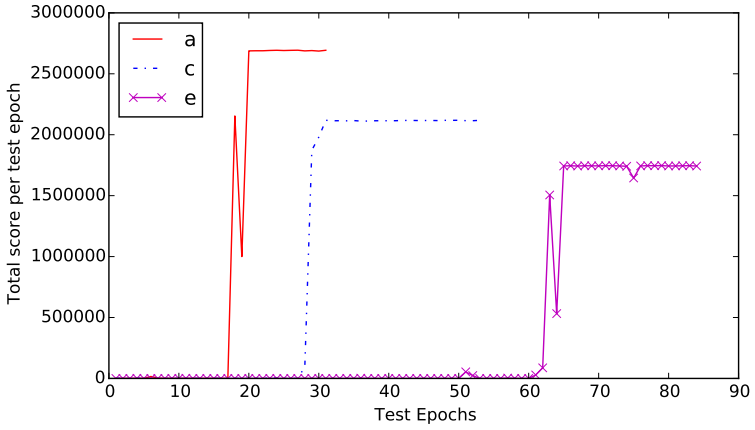
(f) Harder paths on both sides with intermediate reward



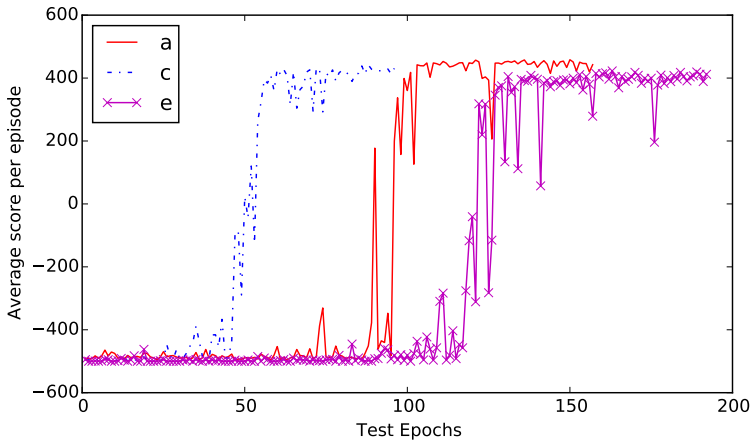
(g) Harder left path with further intermediate reward

Figure 5.9. The different setups of Ms. Pacman environment we tested DQN on. Intermediate reward is colored in brighter gray. Changes between setups are denoted with red circles.

Just like in the maze experiments, we test for factors that change the complexity of the game to see their effect on the convergence of DQN. There are three factors: (a) There are two paths from the initial position at the bottom to the goal at the top, and one can block either of them or not, (b) There may be enemy units or not to avoid contact with, and (c) There may be intermediate rewards on the correct path. We also try combinations of those factors and in Figure 5.9, we show the seven different setups. By training DQN in each of these setups with zero or two enemy units, we experiment with a total of fourteen different setups.



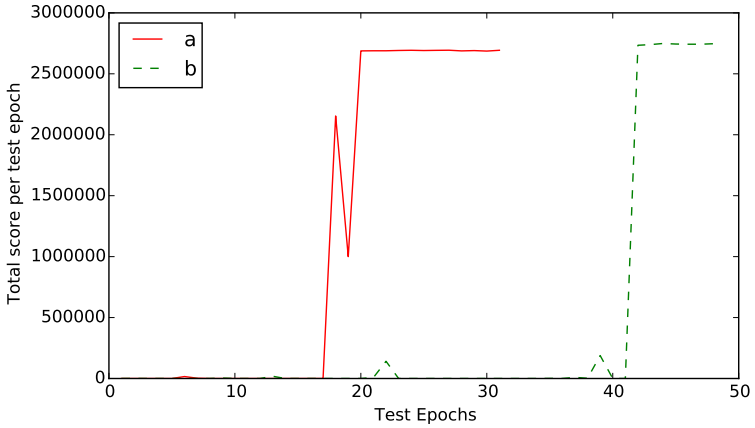
(a) No enemies



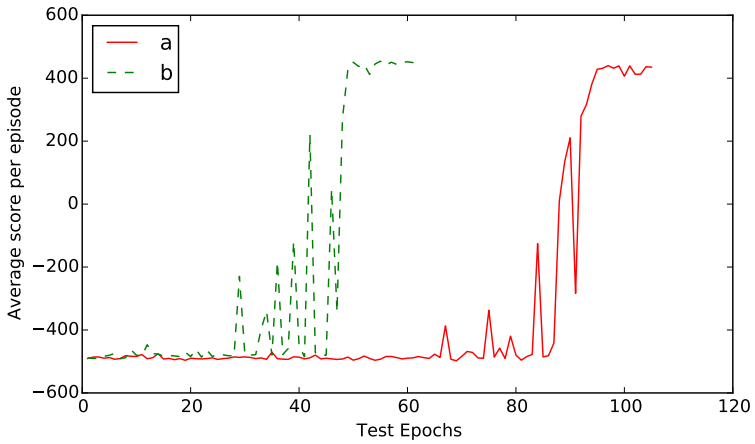
(b) Two enemies

Figure 5.10. Comparison of DQN convergences on original Pacman setup (a), harder left path (c) and harder paths on both sides (e).

We start by closing some paths by adding obstacles (see Figure 5.10). In setup (c), we make one of the two possible routes longer by closing some paths and in setup (e), both the leftmost and the rightmost solution paths are extended by adding obstacles. We can see in Figure 5.10a that extending the leftmost path (c) decreases learning with respect to (a). However, if we examine Figure 5.10b, we can see that adding some enemy units to (c) increases the learning speed, since most of the time the agent gets destroyed in the leftmost path by the enemies, thus the agent learns that it should not dwell on the leftmost path concentrating on the correct path which is the rightmost path. If we compare setup (e) with (a) and (c), we see that making all possible paths longer decreases the learning speed, as expected.



(a) No enemies

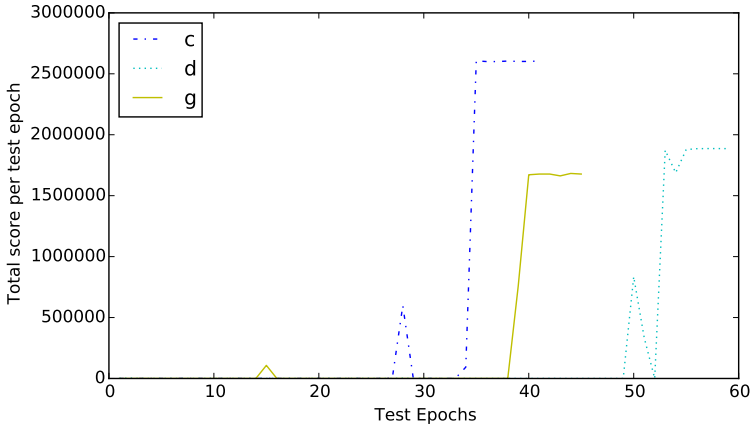


(b) Two enemies

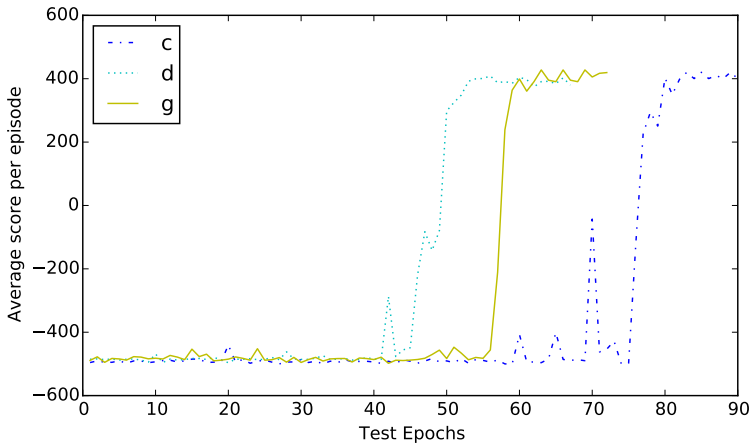
Figure 5.11. Comparison of DQN convergences on original Pacman setup (a) and intermediate reward (b).

If we examine setups (a) and (b) (see Figure 5.11), we see that adding an intermediate reward increases DQN's learning speed considerably in the two-enemy setup. This is expected since the intermediate reward helps algorithm to focus on one of the two paths. In the no-enemy setup however, the agent learns getting the intermediate reward quickly, but it tends to stay near the intermediate reward for many epochs to come since ϵ value gets its lower bound in four epochs and the agent cannot explore the target quickly. Thus, it decreases the learning speed.

The intermediate reward in setup (d) was actually designed to hinder learning since it is en route to a longer path. In the no-enemy case, the results show that this



(a) No enemies

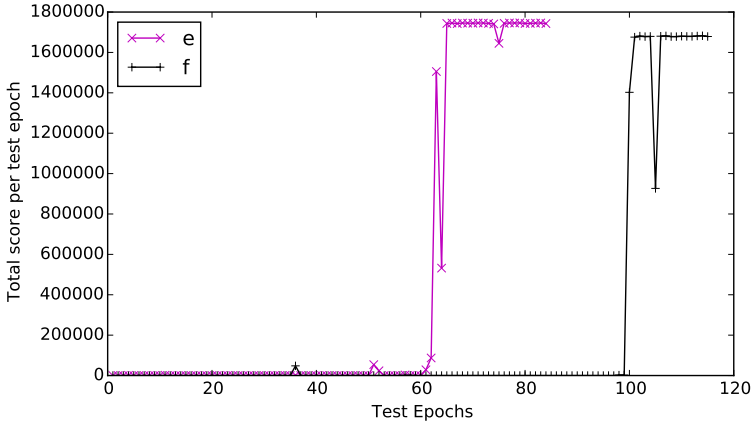


(b) Two enemies

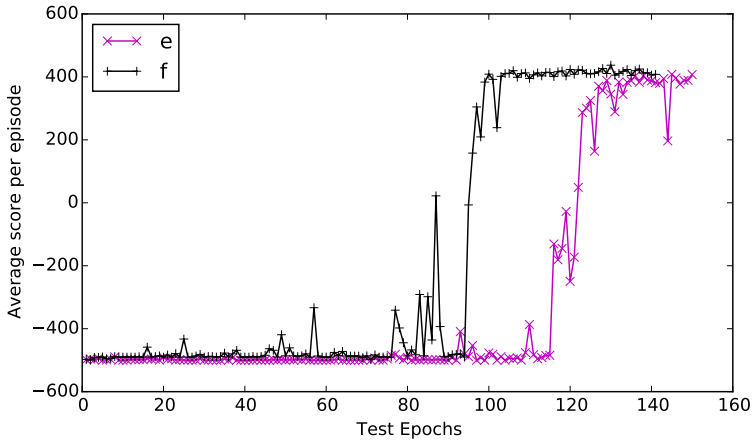
Figure 5.12. Comparison of DQN convergences on Pacman setup with harder left path (c), early intermediate reward (d) and hard intermediate reward (g).

reward actually hinders learning as can be seen in Figure 5.12. After the agent takes this reward, instead of choosing the leftmost path, it uses the rightmost path which is the closer one to the target. However, it results with even better outcomes than setup (c) when there are enemies. The agent in this case is able to learn to choose between left or right paths according to the closeness of the enemies to those paths. In setup (g), we move the intermediate reward to a further position on the left path and saw that it decreases learning speed in no-enemy setup, better than (d). This also helps learning when there are two enemies and provides worse results than (d), because the risk of death is higher in the path with the intermediate reward. Nevertheless, when the algorithm converges, the agent is able to wait for risk of the opponents to pass and

go back if it is necessary.



(a) No enemies



(b) Two enemies

Figure 5.13. Comparison of DQN convergences on Pacman setup with harder paths on both sides (e) and intermediate reward (f).

Comparison of setups (e) and (f) are given in Figure 5.13. Similar to the previous results, since the ϵ value gets to its minimum value quickly, the intermediate reward in this case (f) slows down the learning process. Slow down rate is extremely high because most of the paths are blocked, the agent cannot get the target . It helps when there are enemies, because enemies force the agent to explore.

6. CONCLUSIONS AND FUTURE WORK

6.1. Summary of Findings

Deep reinforcement learning is a recent research area that combines deep neural networks with reinforcement learning. The Deep Q-Network learns to play Atari games end-to-end. However, just like with other neural network applications, DQN is a black-box learner, meaning that we cannot interpret how and why the network learns. DQN learns some games better than others, some faster than others and in some, DQN is as good as a human player whereas in some it is not. Our assumption is that the complexity of a game depends on some factors and these factors affect DQN's learning speed and quality. This thesis is an attempt to find such factors as well as analyze their effect on the convergence of DQN.

We started by implementing a visual interface that shows the weights and activations of network layers. Using this interface, we have seen some abstract features that the network learns. However, these abstract features change between games, so it was not possible to capture a general pattern. Using a clustering algorithm, we find that the DQN convergence curves of 45 ATARI 2600 games correlate with some game characteristics and that games that are played similarly are learned similarly by DQN and are placed nearby in the clustering dendrogram.

We defined variants of a Maze task on which we defined a number of factors and tested their effect using DQN as it is, with no changes to the network architecture or learning algorithm. These factors include:

- the maze size that changes the average number of actions to reach the goal,
- separating walls that increases difficulty,
- enemy units that prevent player from reaching the goal,
- intermediate rewards that break down a long task into smaller tasks.

We see that larger mazes and the presence of enemy units generally delay convergence since the environment becomes more complex. Intermediate rewards, as expected, increase the learning speed since they provide a hint, dividing a long sequence into smaller sequences.

In the second set of experiments, we use a Pacman environment with similar factors with a total of 14 different setups. We find that the factors affect learning differently if there are enemies. Blocking a path with a wall usually decreases the learning speed, but has the opposite effect if it is blocking many of the possible paths thereby reducing possible actions.

In our experiments, we have used the same neural network structure with the same algorithm and hyper-parameters as defined in the original DQN paper. Even if we change the single parameter that controls the decay of exploration rate ϵ, the convergence speed of the algorithm is greatly affected. Since experiments in deep learning takes too much time to complete, we believe every shared experimental result is an asset and we hope this work will aid researchers who are working in this area.

The ultimate aim is to transfer what we learn from DQN's behavior on games to what deep reinforcement learning can do in real life. From these experiments, we would like to move a level up and define at a more abstract level, general tasks and general strategies to solve them, as well as how such strategies can be learned. Our work is one small step towards this aim.

6.2. Future Work

There are many possible research directions:

- One is to add more factors that affect the difficulty of the game. For example, in most games there is randomness. In our maze setup, the initial positions of units are randomly selected but the moves of agents are not random; it would

certainly affect the learning performance if any random event should occur during the game.

- We believe our maze setups are suitable for curriculum learning [52,53] and should result with better learning curves on setups with similar abstract factors.
- Transferring weights of a trained network [54] might increase the learning speed in our experiments.
- We trained our network using the same hyper-parameters with the original paper. However, we believe that since this huge network is adapted to play most of the Atari games that are more complicated, most of the weights are redundant for our much simpler maze problem. Using a smaller network should result with faster convergence rates and would allow more experiments.
- Conducting the same experiments with recent extensions to DQN such as Double DQN, A3C, etc. may accelerate learning.

REFERENCES

1. LeCun, Y., L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, Vol. 86, No. 11, pp. 2278–2324, 1998.

2. He, K., X. Zhang, S. Ren and J. Sun, "Deep residual learning for image recognition", *arXiv preprint arXiv:1512.03385*, 2015.

3. Szegedy, C., W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, "Going deeper with convolutions", *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, June 2015.

4. Collobert, R., J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu and P. Kuksa, "Natural Language Processing (Almost) from Scratch", *Journal of Machine Learning Research*, Vol. 12, pp. 2493–2537, 2011.

5. Socher, R., A. Perelygin, J. Y. Wu, J. Chuang, C. D. Manning, A. Y. Ng and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank", *Proceedings of the conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1631–1642, Association for Computational Linguistics, 2013.

6. Dahl, G. E., *Deep learning approaches to problems in speech recognition, computational chemistry, and natural language text processing*, Ph.D. Thesis, University of Toronto, 2015.

7. Deng, L., J. Li, J.-T. Huang, K. Yao, D. Yu, F. Seide, M. L. Seltzer, G. Zweig, X. He, J. D. Williams, Y. Gong and A. Acero, "Recent advances in deep learning for speech research at Microsoft.", *ICASSP*, pp. 8604–8608, IEEE, 2013.

8. Mahadevan, S. and J. Connell, "Automatic programming of behavior-based robots

using reinforcement learning", *Artificial Intelligence*, Vol. 55, No. 2, pp. 311 – 365, 1992.

9. Matarić, M. J., "Reinforcement Learning in the Multi-Robot Domain", *Autonomous Robots*, Vol. 4, No. 1, pp. 73–83, 1997.

10. Tesauro, G., "Temporal Difference Learning and TD-Gammon", *Commun. ACM*, Vol. 38, No. 3, pp. 58–68, 1995.

11. Thrun, S., "Learning to Play the Game of Chess", G. Tesauro, D. Touretzky and T. Leen (Editors), *Advances in Neural Information Processing Systems (NIPS) 7*, MIT Press, Cambridge, MA, 1995.

12. Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning", *Nature*, Vol. 518, No. 7540, pp. 529–533, 2015.

13. Yosinski, J., J. Clune, A. M. Nguyen, T. Fuchs and H. Lipson, "Understanding Neural Networks Through Deep Visualization", *arXiv preprint arXiv:1506.06579*, 2015.

14. Das, A., H. Agrawal, C. L. Zitnick, D. Parikh and D. Batra, "Human Attention in Visual Question Answering: Do Humans and Deep Networks Look at the Same Regions?", *arXiv preprint arXiv:1606.03556*, 2016.

15. Maaten, L. v. d. and G. Hinton, "Visualizing data using t-SNE", *Journal of Machine Learning Research*, Vol. 9, No. Nov, pp. 2579–2605, 2008.

16. Alpaydın, E., *Introduction to Machine Learning*, The MIT Press, 2nd edn., 2010.

17. Erhan, D., Y. Bengio, A. Courville and P. Vincent, *Visualizing Higher-Layer Features of a Deep Network*, Tech. Rep. 1341, University of Montreal, 2009, also

presented at the ICML 2009 Workshop on Learning Feature Hierarchies, Montréal, Canada.

18. Yosinski, J., J. Clune, A. Nguyen, T. Fuchs and H. Lipson, "Understanding Neural Networks Through Deep Visualization", *Deep Learning Workshop, International Conference on Machine Learning (ICML)*, 2015.

19. Olah, C., A. Mordvintsev and L. Schubert, "Feature Visualization", *Distill*, 2017.

20. Olah, C., A. Satyanarayan, I. Johnson, S. Carter, L. Schubert, K. Ye and A. Mordvintsev, "The Building Blocks of Interpretability", *Distill*, 2018.

21. Howard, R., *Dynamic Programming and Markov Processes*, MIT Press, 1960.

22. Sutton, R. S., "Learning to predict by the methods of temporal differences", *Machine Learning*, Vol. 3, No. 1, pp. 9–44, 1988.

23. Korjus, K., I. Kuzovkin, A. Tampuu and T. Pungas, *Artificial General Intelligence that plays Atari video games: How did DeepMind do it?*, 2014, `https://goo.gl/FH9YuE`, accessed at April 2018.

24. Lin, L.-J., "Programming Robots Using Reinforcement Learning and Teaching", *Proceedings of the Ninth National Conference on Artificial Intelligence - Volume 2*, AAAI'91, pp. 781–786, AAAI Press, 1991.

25. Ruder, S., "An overview of gradient descent optimization algorithms", *arXiv preprint arXiv:1609.04747*, 2016.

26. Duchi, J., E. Hazan and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", *Journal of Machine Learning Research*, Vol. 12, pp. 2121–2159, 2011.

27. Levine, S., C. Finn, T. Darrell and P. Abbeel, "End-to-end training of deep visuo-

motor policies", *Journal of Machine Learning Research*, Vol. 17, No. 39, pp. 1–40, 2016.

28. Silver, D., A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of Go with deep neural networks and tree search", *Nature*, Vol. 529, No. 7587, pp. 484–489, 2016.

29. Baudiš, P. and J.-l. Gailly, "Pachi: State of the art open source Go program", *Advances in computer games*, pp. 24–38, Springer, 2011.

30. Kocsis, L. and C. Szepesvári, "Bandit based monte-carlo planning", *European Conference on Machine Learning*, pp. 282–293, Springer, 2006.

31. Silver, D., J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel and D. Hassabis, "Mastering the game of Go without human knowledge", Vol. 550, pp. 354–359, October 2017.

32. Silver, D., T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm", *arXiv preprint arXiv:1712.01815*.

33. Peng, X. B., G. Berseth and M. van de Panne, "Terrain-adaptive locomotion skills using deep reinforcement learning", *ACM Transactions on Graphics*, Vol. 35, No. 4, p. 81, 2016.

34. Peng, X. B., G. Berseth and M. van de Panne, "Dynamic terrain traversal skills using reinforcement learning", *ACM Transactions on Graphics*, Vol. 34, No. 4, p. 80, 2015.

35. Lillicrap, T. P., J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver

and D. Wierstra, "Continuous control with deep reinforcement learning", *arXiv preprint arXiv:1509.02971*, 2015.

36. Silver, D., G. Lever, N. Heess, T. Degris, D. Wierstra and M. Riedmiller, "Deterministic Policy Gradient Algorithms", *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, ICML'14, pp. I–387–I–395, JMLR.org, 2014.

37. Hasselt, H. V., "Double Q-learning", *Advances in Neural Information Processing Systems*, pp. 2613–2621, 2010.

38. Hasselt, H. v., A. Guez and D. Silver, "Deep Reinforcement Learning with Double Q-Learning", *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pp. 2094–2100, AAAI Press, 2016.

39. Schaul, T., J. Quan, I. Antonoglou and D. Silver, "Prioritized experience replay", *arXiv preprint arXiv:1511.05952*, 2015.

40. Dean, J., G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks", *Advances in neural information processing systems*, pp. 1223–1231, 2012.

41. Nair, A., P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen *et al.*, "Massively parallel methods for deep reinforcement learning", *arXiv preprint arXiv:1507.04296*, 2015.

42. Mnih, V., A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning", *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*, pp. 1928–1937, 2016.

43. Sutton, R. S. and A. G. Barto, *Reinforcement learning: An introduction*, Vol. 1, MIT press Cambridge, 1998.

44. Wang, Z., N. de Freitas and M. Lanctot, "Dueling Network Architectures for Deep Reinforcement Learning", *arXiv preprint arXiv:1511.06581*, 2015.

45. Bellemare, M. G., W. Dabney and R. Munos, "A Distributional Perspective on Reinforcement Learning", *arXiv preprint arXiv:1707.06887*, 2017.

46. Hessel, M., J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. G. Azar and D. Silver, "Rainbow: Combining Improvements in Deep Reinforcement Learning", *arXiv preprint arXiv:1710.02298*, 2017.

47. Elias, G. S., R. Garfield and K. R. Gutschera, *Characteristics of Games*, The MIT Press, 2012.

48. Anderson, D., M. Stephenson, J. Togelius, C. Salge, J. E. Levine and J. Renz, "Deceptive Games", *Applications of Evolutionary Computation 21st International Conference, EvoApplications 2018 Proceedings*, pp. 376–391, Springer, 2018.

49. Yannakakis, G. N. and J. Togelius, *Artificial Intelligence and Games*, Springer, 2018, `http://gameaibook.org`.

50. Berndt, D. J. and J. Clifford, "Using Dynamic Time Warping to Find Patterns in Time Series", *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, AAAIWS'94, pp. 359–370, AAAI Press, 1994.

51. DeNero, J. and D. Klein, *UC Berkeley CS188 Intro to AI – Course Materials*, 2014, `http://ai.berkeley.edu/reinforcement.html`, accessed at May 2018.

52. Bengio, Y., J. Louradour, R. Collobert and J. Weston, "Curriculum Learning", *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pp. 41–48, ACM, New York, NY, USA, 2009.

53. Svetlik, M., M. Leonetti, J. Sinapov, R. Shah, N. Walker and P. Stone, "Automatic Curriculum Graph Generation for Reinforcement Learning Agents", *AAAI*, 2017.

54. Parisotto, E., J. Lei Ba and R. Salakhutdinov, "Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning", *arXiv preprint arXiv:1511.06342*, 2015.

55. Sprague, N., *Theano-based implementation of Deep Q-learning*, 2016, `https://github.com/spragunr/`, accessed at May 2018.

56. Emekligil, E., *Theano-based implementation of Deep Q-learning*, 2018, `https://github.com/erdememekligil/deep_q_rl/tree/dev`, accessed at May 2018.

57. Bellemare, M. G., Y. Naddaf, J. Veness and M. Bowling, "The Arcade Learning Environment: An Evaluation Platform for General Agents", *Journal of Artificial Intelligence Research*, Vol. 47, pp. 253–279, 2013.

APPENDIX A: DEEP NEURAL NETWORK VISUALIZATION

A visualization tool is implemented as part of this thesis to show the learned convolutional kernels and their forward-pass activations (outputs) at the run-time. The tool has two interfaces. The first one shown in Figure A.1, displays kernels for all 32 output feature maps of each layer separately. Each of these outputs are computed by convolving four input frames with four weights and summing them up with a bias.

Figure A.1. The visualization tool that shows kernels as well as their outputs. Weights that are close to zero are painted in red.

The second interface shows all of the activations on the same page. An example of this interface is given in Figure A.2.

With this tool, it is possible to see that in an Atari playing neural network, there are some redundancies. This means that the network architecture can be optimized for any particular game.

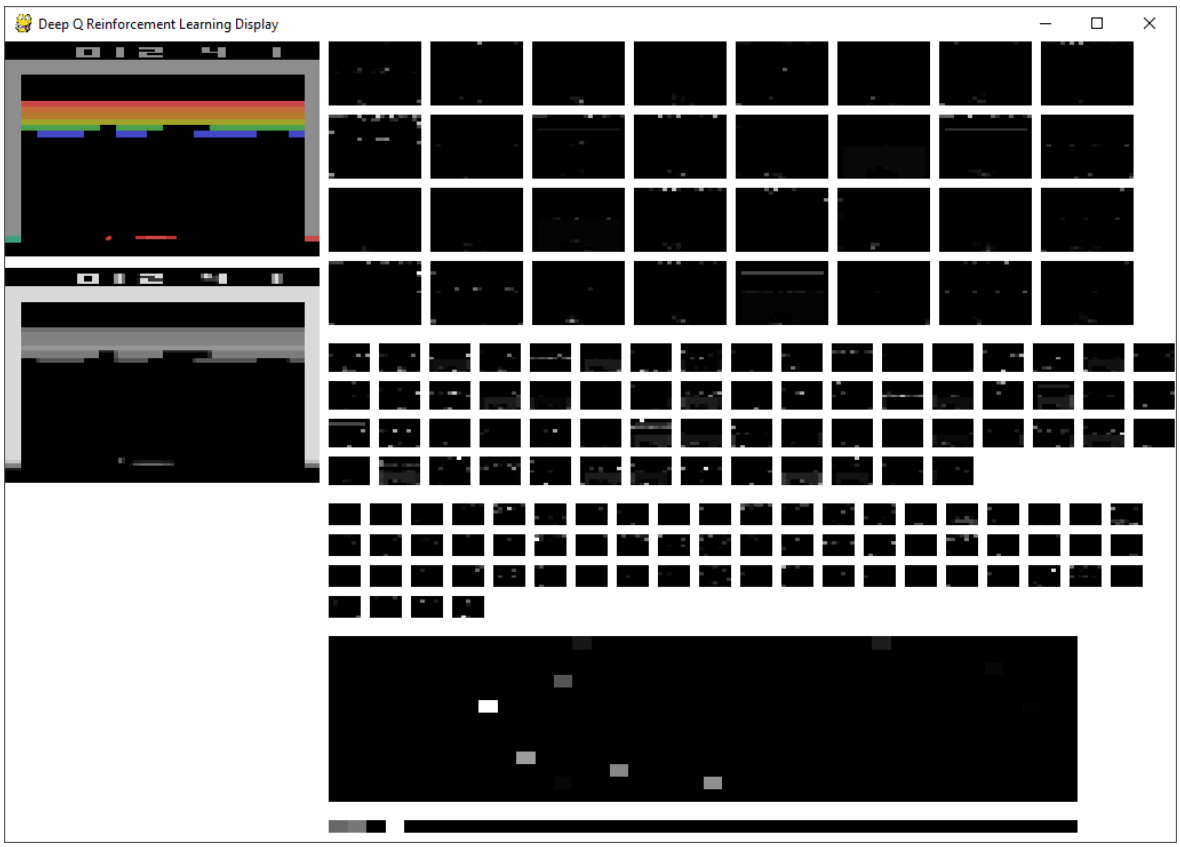


Figure A.2. The visualization tool that shows kernels as well as their outputs. Weights that are close to zero are painted in red.

APPENDIX B: FACTORS OF ATARI 2600 GAMES

To fit this format, our data is split into two as Table B.10 and B.11. The encodings are explained in Tables B.1-B.9.

Table B.1. Encoding of enemy attribute.

Label	Explanation
0	Multiple
1	Multiple enemies (Increases between stages)
2	Multiple enemies (Does not change over time)
3	Multiple enemies (Increases over time)
4	One enemy
5	No

Table B.2. Encoding of intermediate reward attribute.

Label	Explanation
0	Yes
1	Yes (Each round)

Table B.3. Encoding of time constraint attribute.

Label	Explanation
0	No
1	Yes

Table B.4. Encoding of reflexes attribute.

Label	Explanation
0	Does not depend on reflexes
1	Require some reflexes
2	Does depend on reflexes

Table B.5. Encoding of death attribute.

Label	Explanation
0	No
1	Yes, ends game

Table B.6. Encoding of randomness attribute.

Label	Explanation
0	Stochastic game
1	Deterministic game

Table B.7. Encoding of changes attribute.

Label	Explanation
0	Nothing changes
1	Colors change
2	Shapes change
3	Color and shapes change

Table B.8. Encoding of speedup attribute.

Label	Explanation
0	No speed-up
1	Speed-up over time

Table B.9. Encoding of reward difficulty attribute.

Label	Explanation
0	Easy
1	Medium
2	Hard

Table B.10. Factors for ATARI games.

Game	Enemy	Intermediate reward	Time constraint	Reflexes
Alien	0	0	0	1
Assault	1	0	0	1
Asterix	0	0	0	2
Atlantis	0	0	0	2
Battle Zone	0	0	0	1
Bowling	5	1	0	0
Boxing	4	0	1	2
Breakout	5	0	0	2
Centipede	1	0	1	1
Chopper Command	1	0	0	1
Crazy Climber	0	0	1	1
Demon Attack	1	0	0	2
Double Dunk	0	1	1	0
Enduro	0	0	0	2
Fishing Derby	5	0	1	1
Freeway	0	0	0	2
Frostbite	1	1	1	0
Gopher	4	0	0	1
Ice Hockey	2	0	1	0
Kangaroo	0	0	0	1
Krull	0	0	0	2
Kung-Fu Master	3	0	1	2
Ms. Pacman	0	0	0	0
Name This Game	2	0	1	2
Pong	4	0	0	2
Private Eye	2	0	1	1
Q*bert	3	0	0	0
River Raid	3	0	1	1
Road Runner	3	0	0	2
Robotank	3	0	0	0
Seaquest	3	0	1	2
Space Invaders	2	0	1	2
Star Gunner	3	0	0	2
Tennis	5	0	0	1
Time Pilot	0	0	0	2
Tutankham	0	0	1	1
Up and Down	0	0	0	2
Venture	0	0	0	1
Video pinball	5	0	0	2
Wizard of Wor	3	0	0	1
Zaxxon	3	0	0	1

Table B.11. More factors for ATARI games.

Game	Death	Randomness	Changes	Speedup	Reward difficulty
Alien	1	0	3	0	0
Assault	1	0	3	0	0
Asterix	1	0	1	1	0
Atlantis	1	0	0	1	0
Battle Zone	1	0	0	0	1
Bowling	0	1	0	0	1
Boxing	0	0	0	0	0
Breakout	1	1	0	1	0
Centipede	1	1	1	1	1
Chopper Command	1	0	3	0	0
Crazy Climber	1	0	3	0	0
Demon Attack	1	0	3	1	0
Double Dunk	0	1	0	0	2
Enduro	0	0	1	0	0
Fishing Derby	0	0	0	0	2
Freeway	0	0	0	0	2
Frostbite	1	1	3	0	1
Gopher	1	0	0	0	1
Ice Hockey	0	1	0	0	2
Kangaroo	1	1	2	0	0
Krull	1	0	3	0	0
Kung-Fu Master	1	0	2	0	0
Ms. Pacman	1	0	3	0	0
Name This Game	1	1	0	1	0
Pong	1	1	0	1	1
Private Eye	0	1	3	0	2
Q*bert	1	0	3	0	0
River Raid	1	0	3	0	0
Road Runner	1	0	3	0	0
Robotank	1	0	1	0	0
Seaquest	1	0	1	1	0
Space Invaders	1	0	0	1	0
Star Gunner	1	0	2	1	1
Tennis	0	1	0	0	2
Time Pilot	1	0	3	1	1
Tutankham	1	1	3	0	1
Up and Down	1	0	3	0	1
Venture	1	0	3	0	2
Video pinball	1	1	0	0	0
Wizard of Wor	1	0	1	1	1
Zaxxon	1	0	3	0	0

APPENDIX C: CLUSTERING OF GAMES

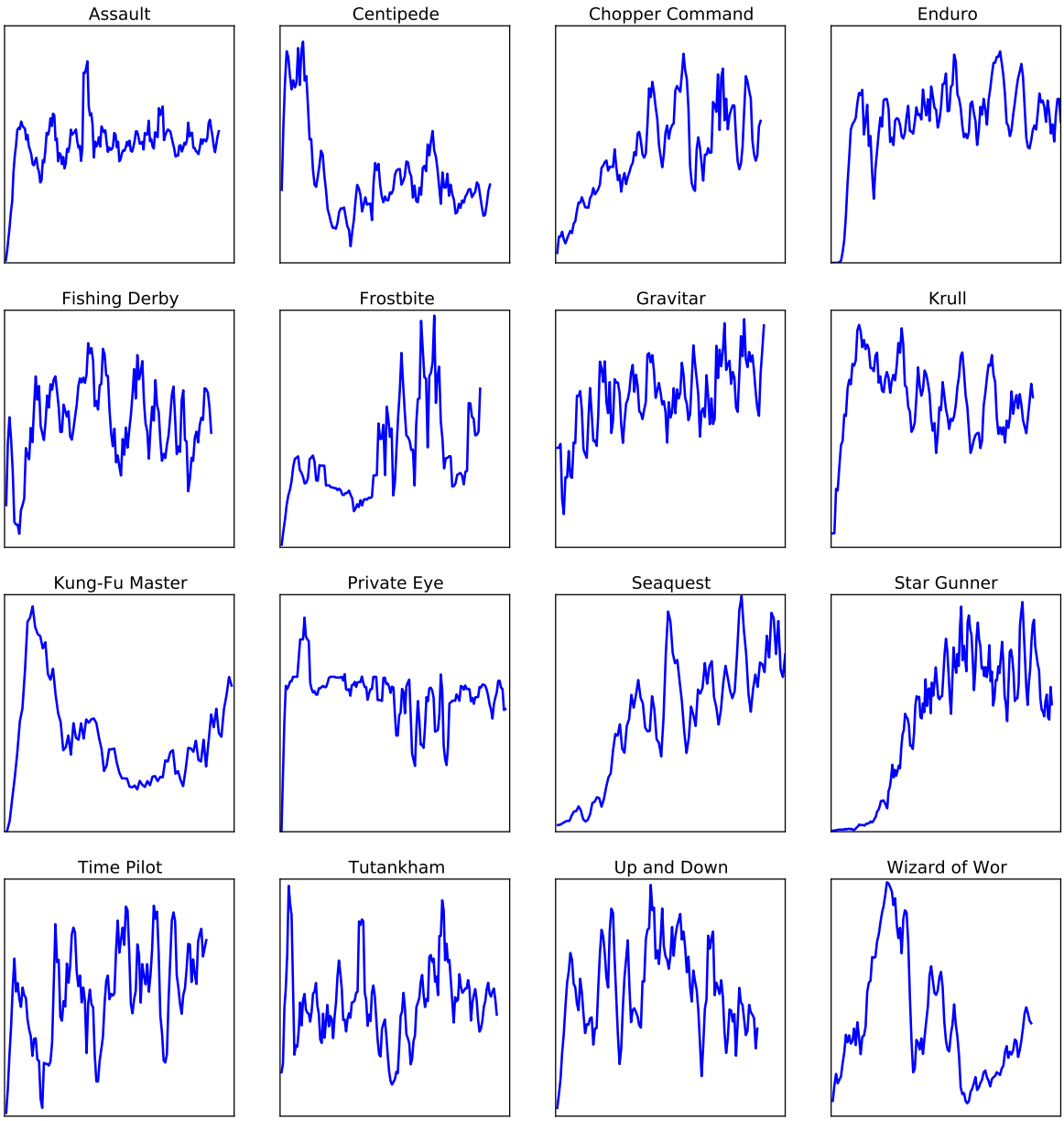


Figure C.1. The convergence plots of DQN on games in the first cluster.

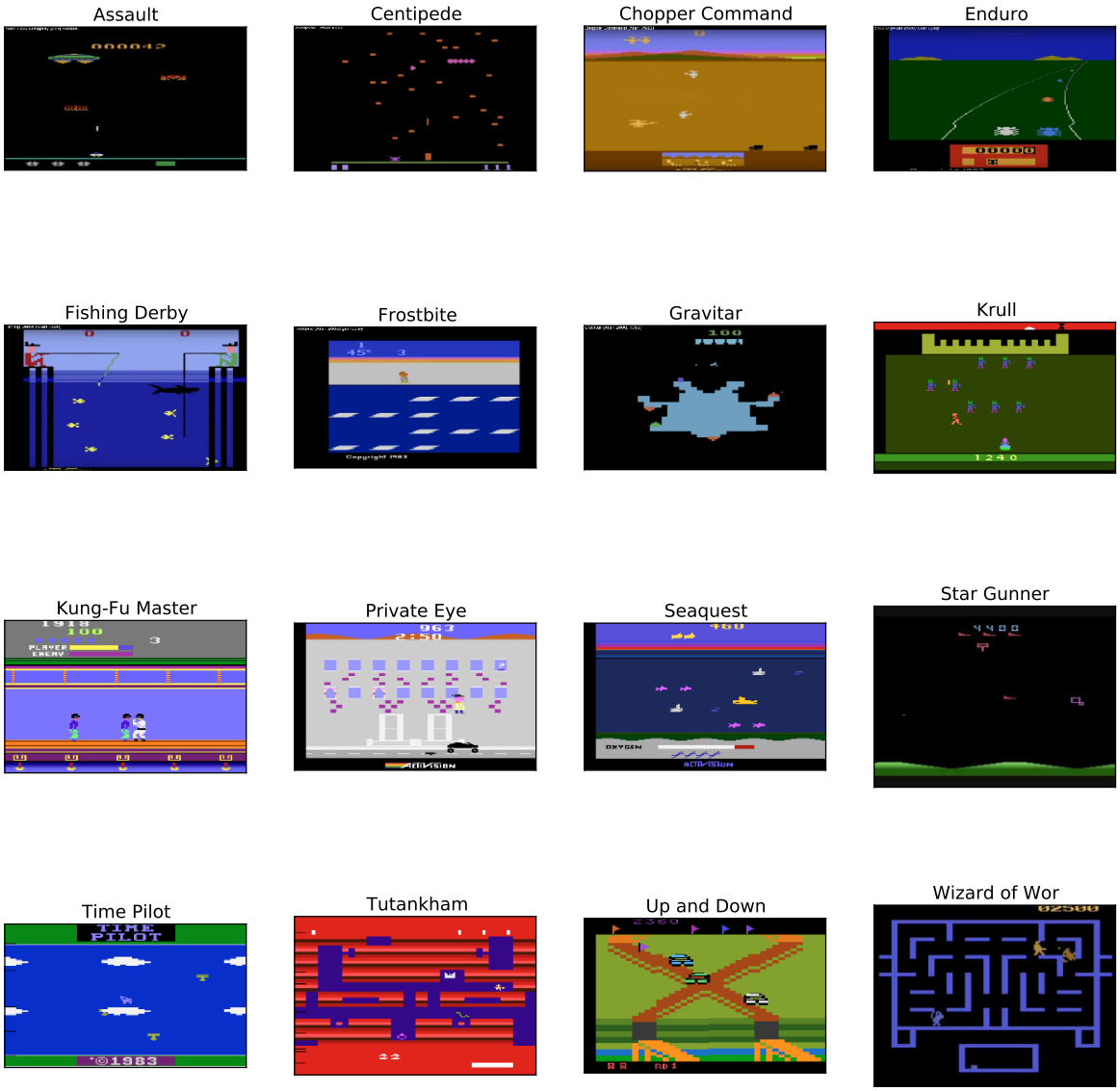


Figure C.2. Sample screenshots of DQN on games in the first cluster.

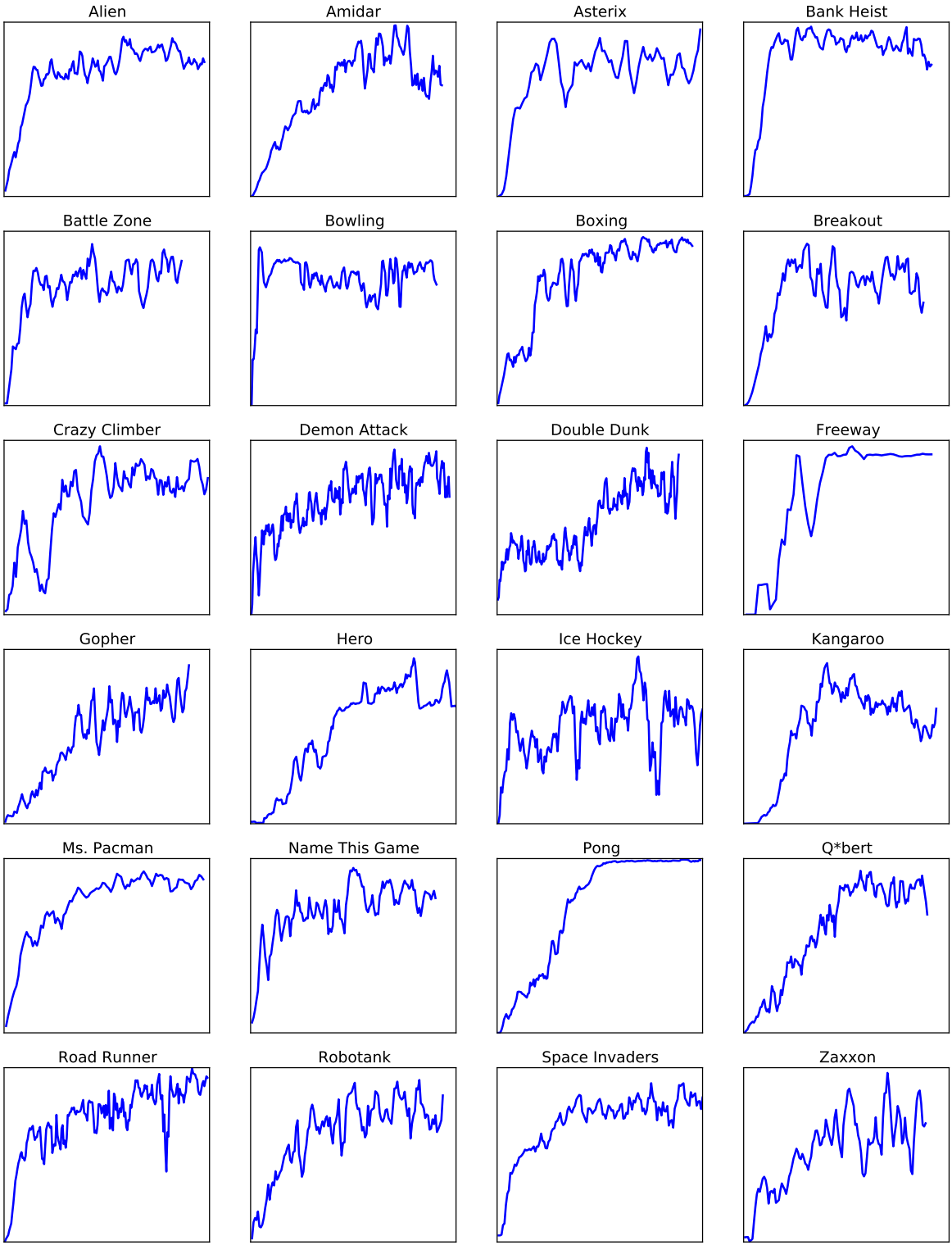


Figure C.3. The convergence plots of DQN on games in the second cluster.

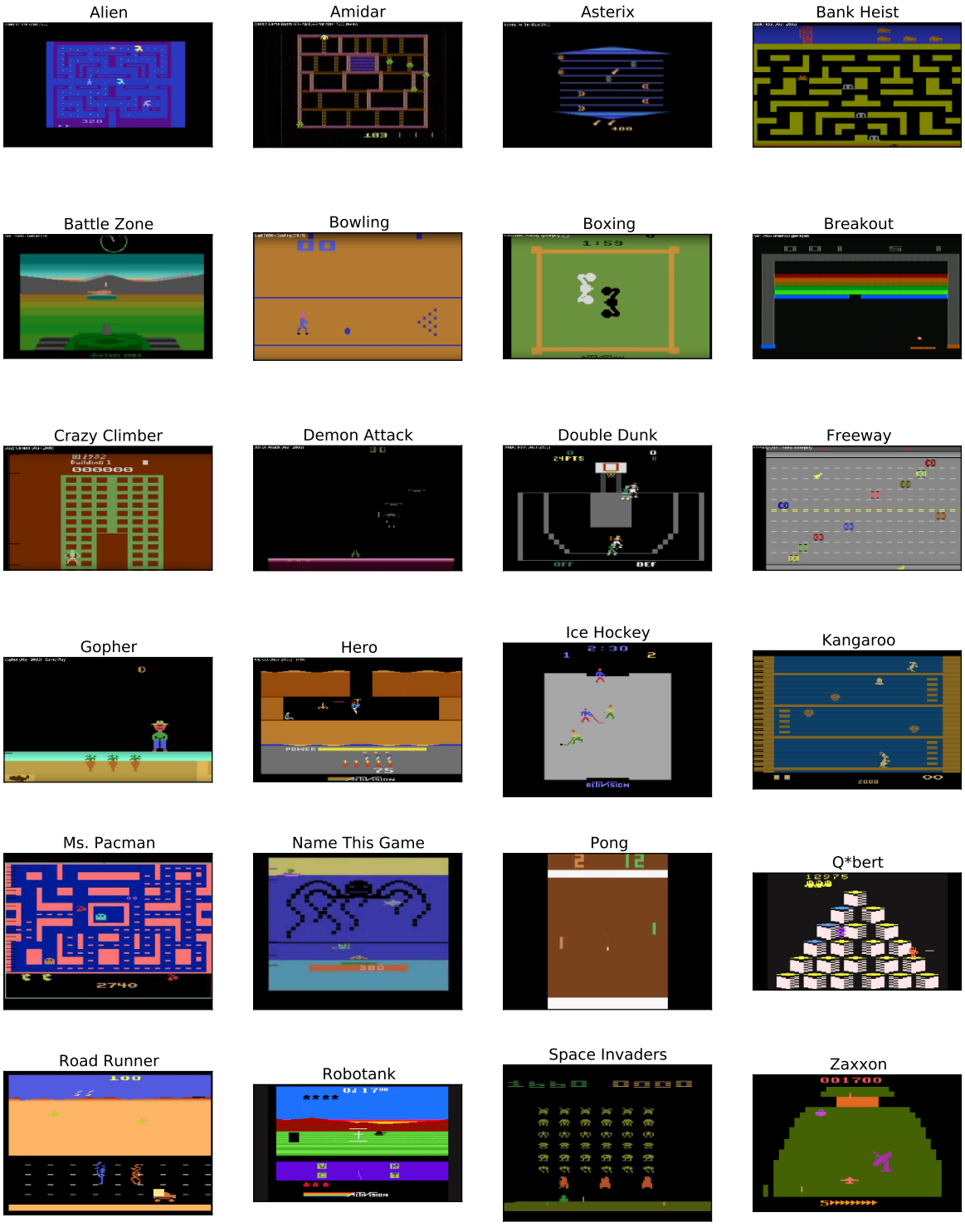


Figure C.4. Sample screenshots of DQN on games in the second cluster.

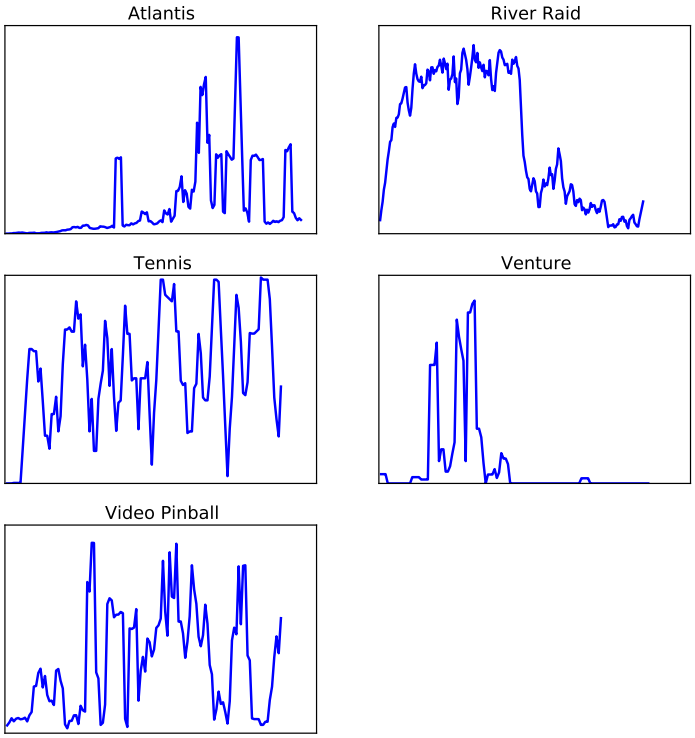


Figure C.5. The convergence plots of DQN on games in the third cluster.

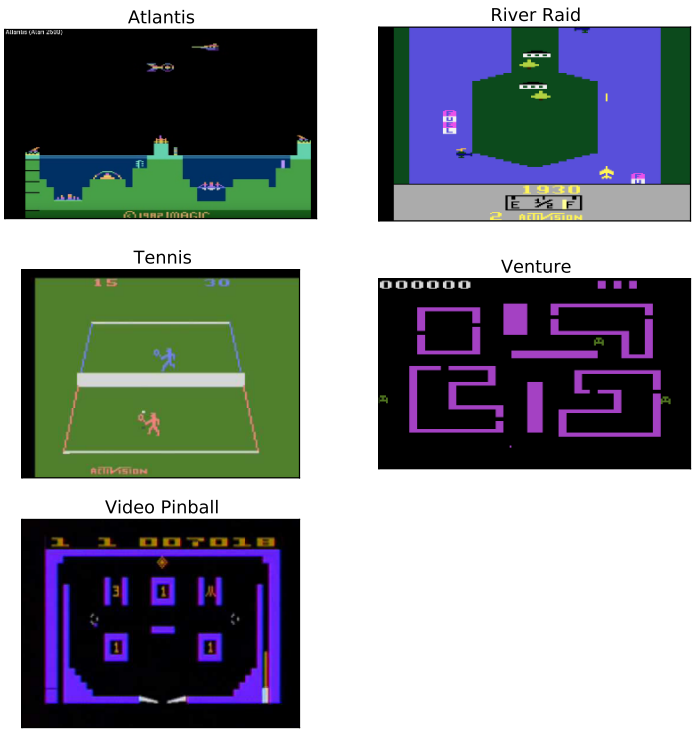
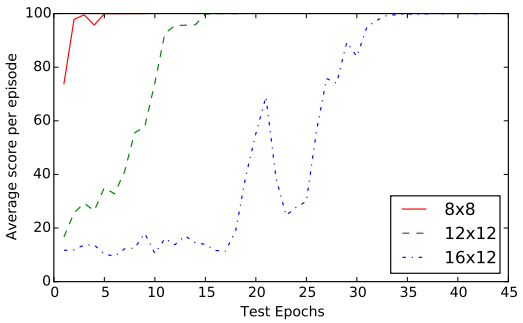


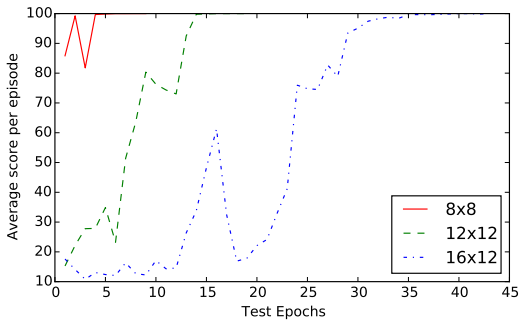
Figure C.6. Sample screenshots of DQN on games in the third cluster.

APPENDIX D: MAZE ENVIRONMENT RESULTS

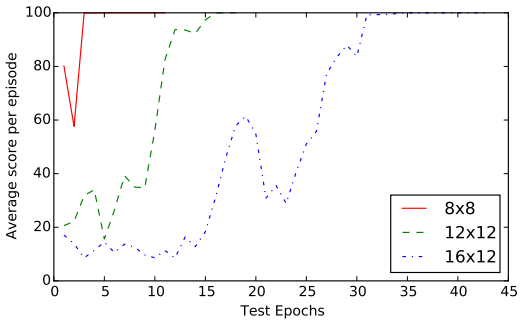
Five experiments with different random seeds are conducted for each maze setup. Only the plot that best represents the average behavior is given throughout the main chapters since the plots that show all of the results on the same figure are hard to understand. Here we show simple maze results for all five runs and pacman results for all ten runs.



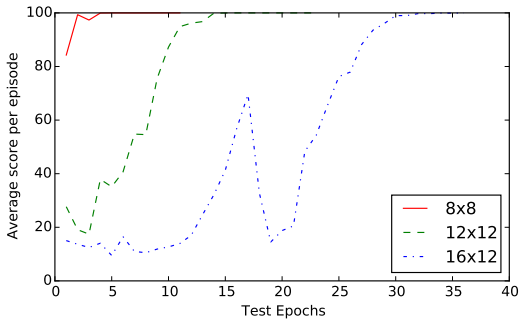
(a) Seed 1



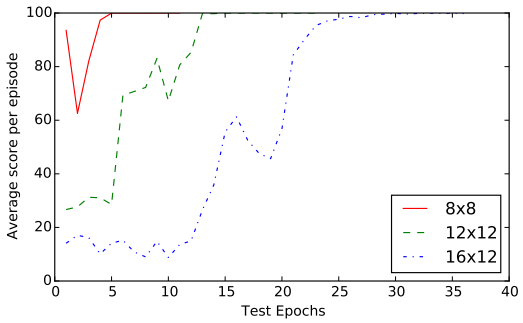
(b) Seed 2



(c) Seed 3

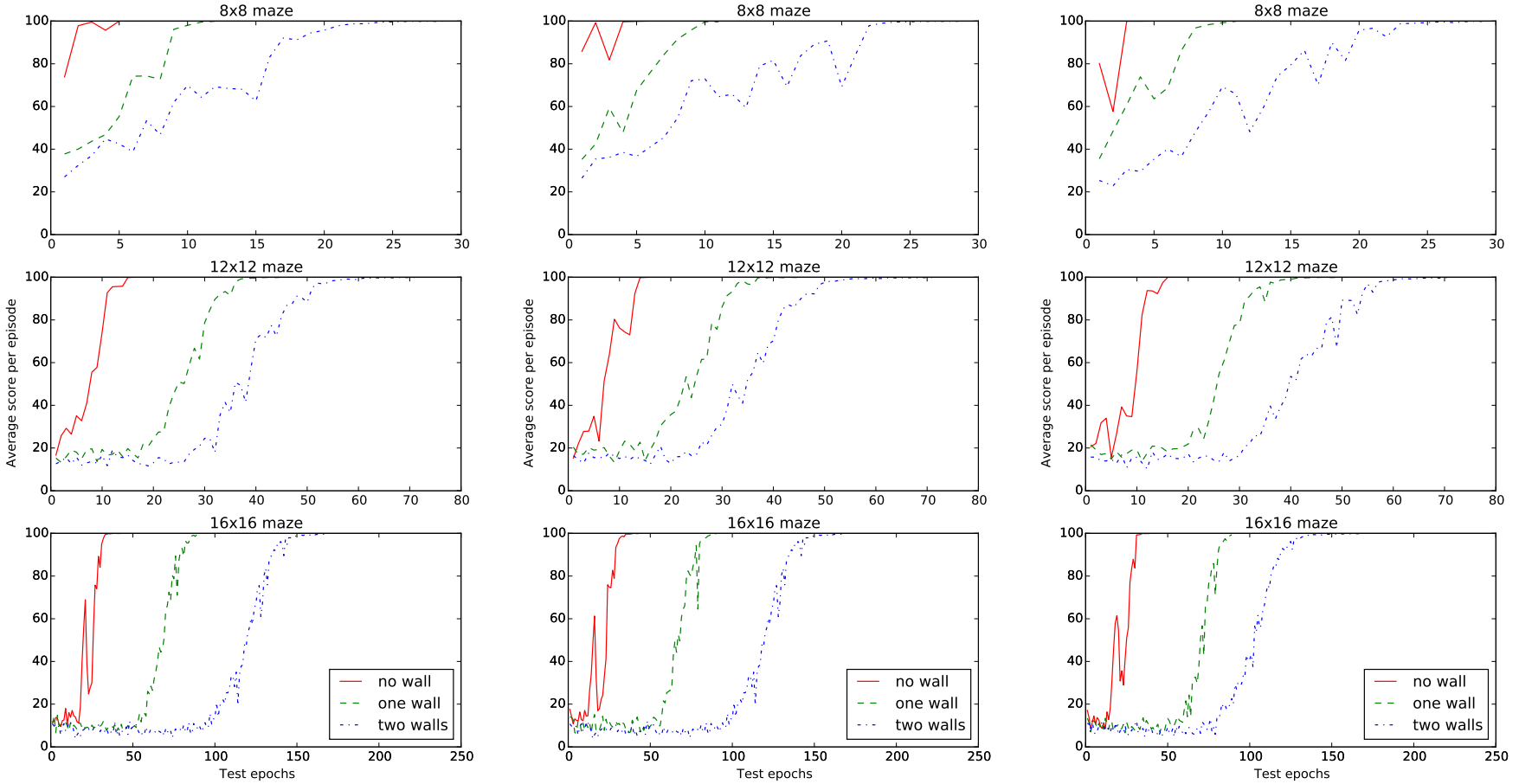


(d) Seed 4



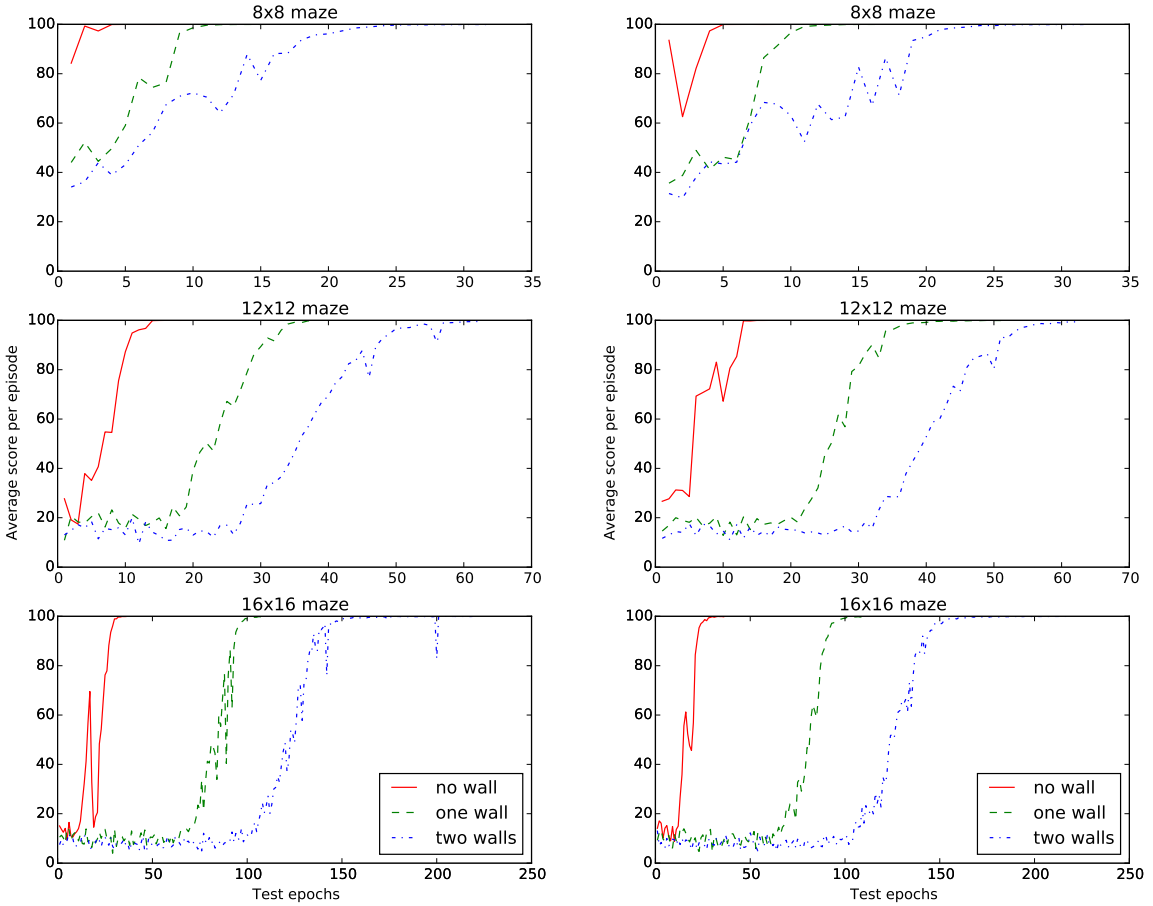
(e) Seed 5

Figure D.1. Convergence of DQN as a function of maze sizes for five seeds.



(a) Seed 1 (b) Seed 2 (c) Seed 3

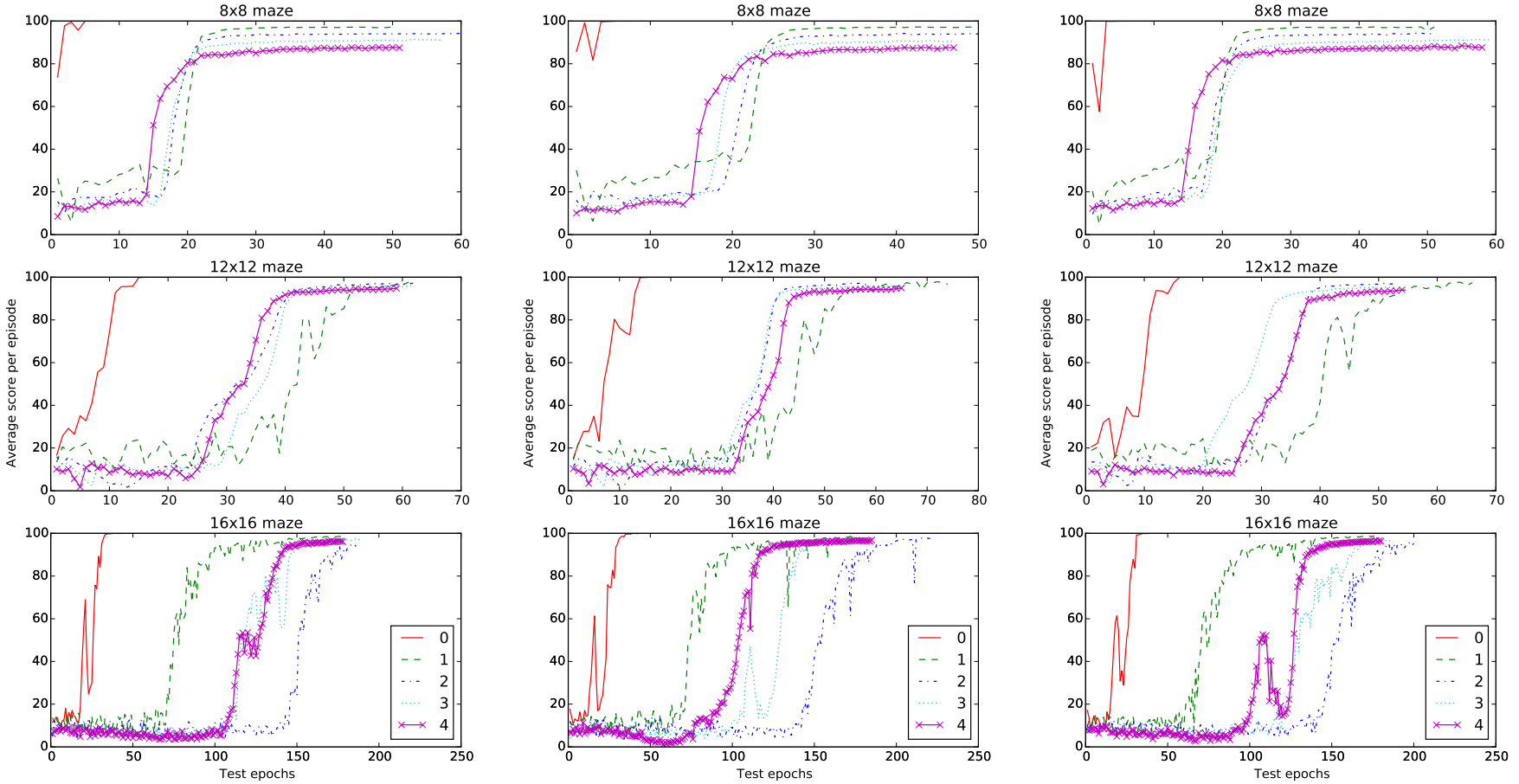
Figure D.2. Convergence of DQN as a function of maze sizes and wall structures; the first three seeds.



(a) Seed 4

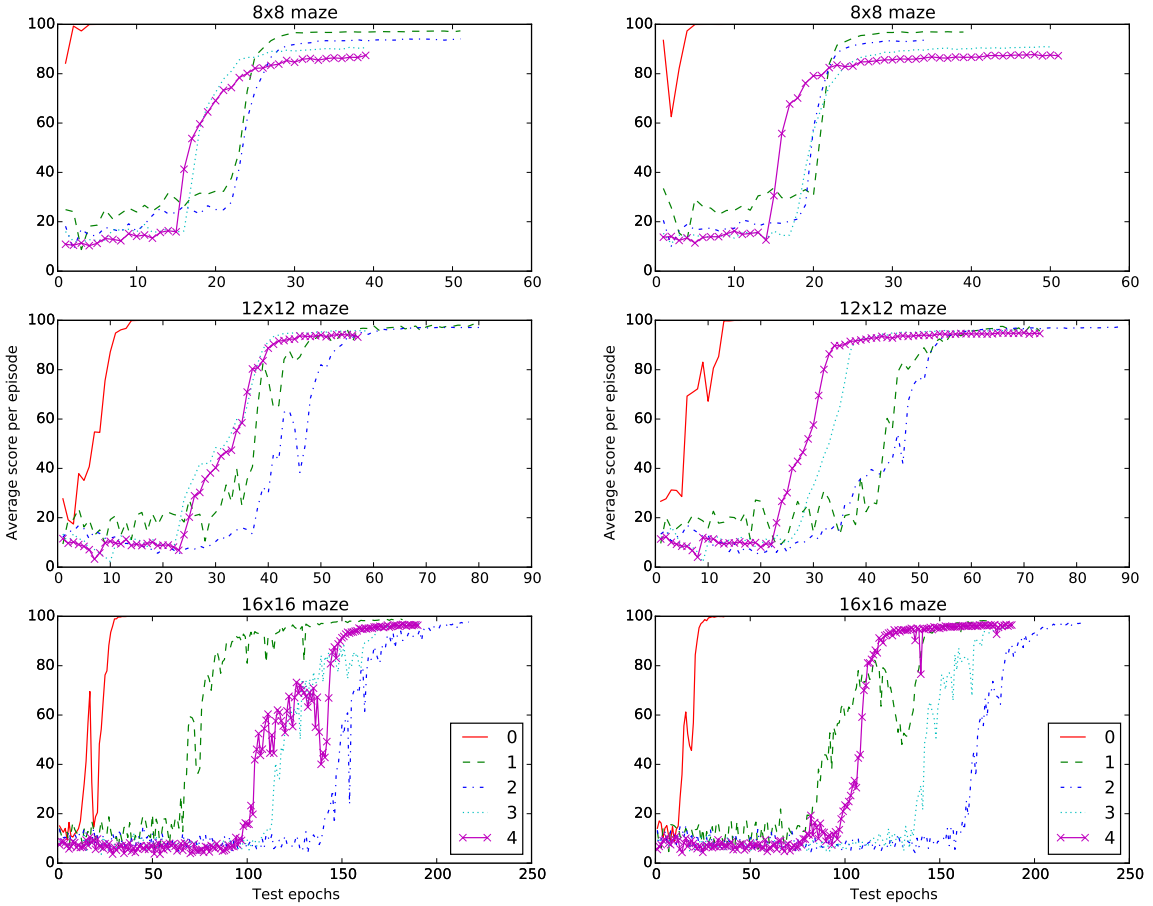
(b) Seed 5

Figure D.3. Convergence of DQN as a function of maze sizes and wall structures; the other two seeds.



(a) Seed 1 (b) Seed 2 (c) Seed 3

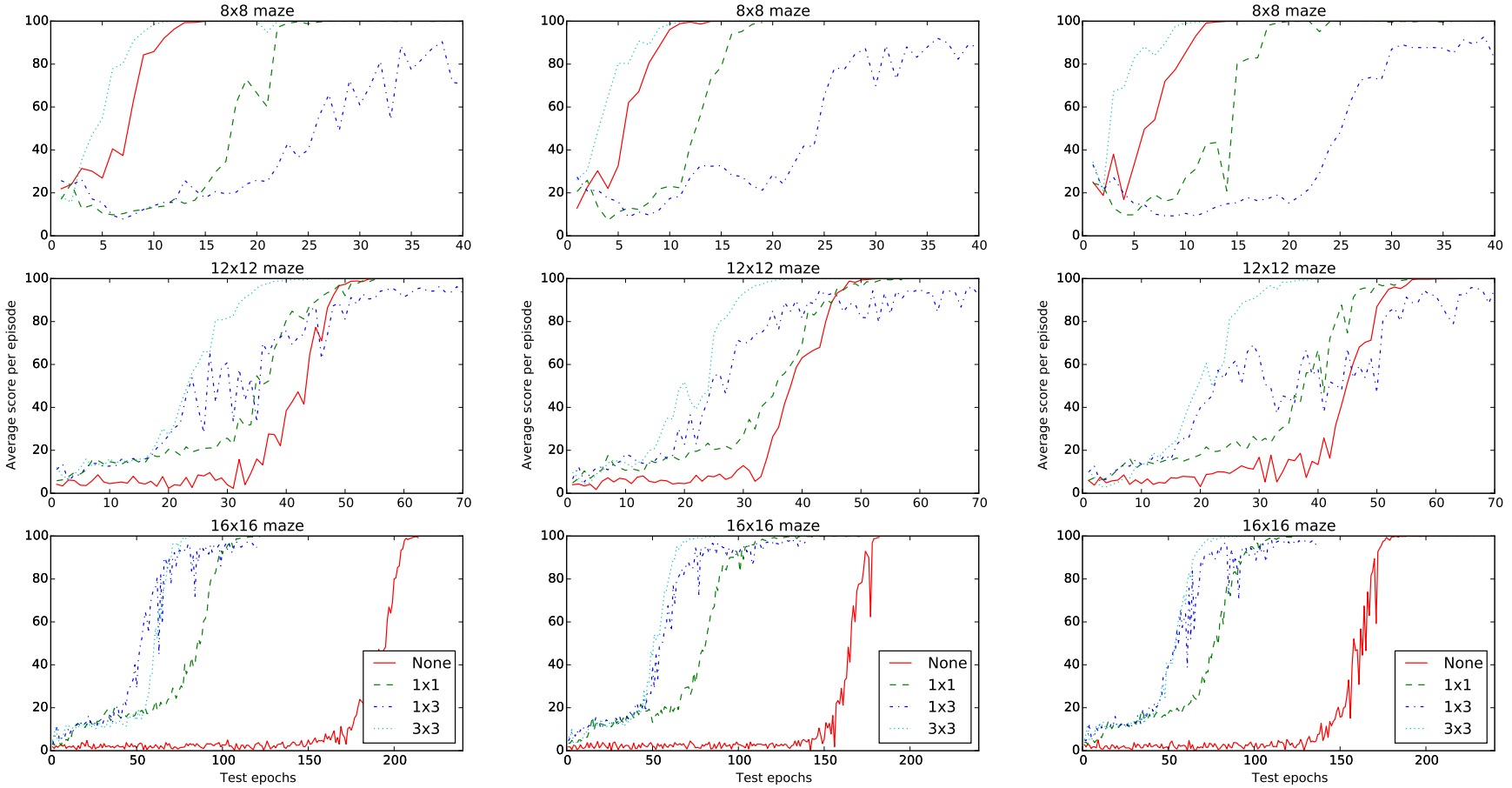
Figure D.4. Convergence of DQN as a function of maze size and the number of enemies; the first three seeds.



(a) Seed 4

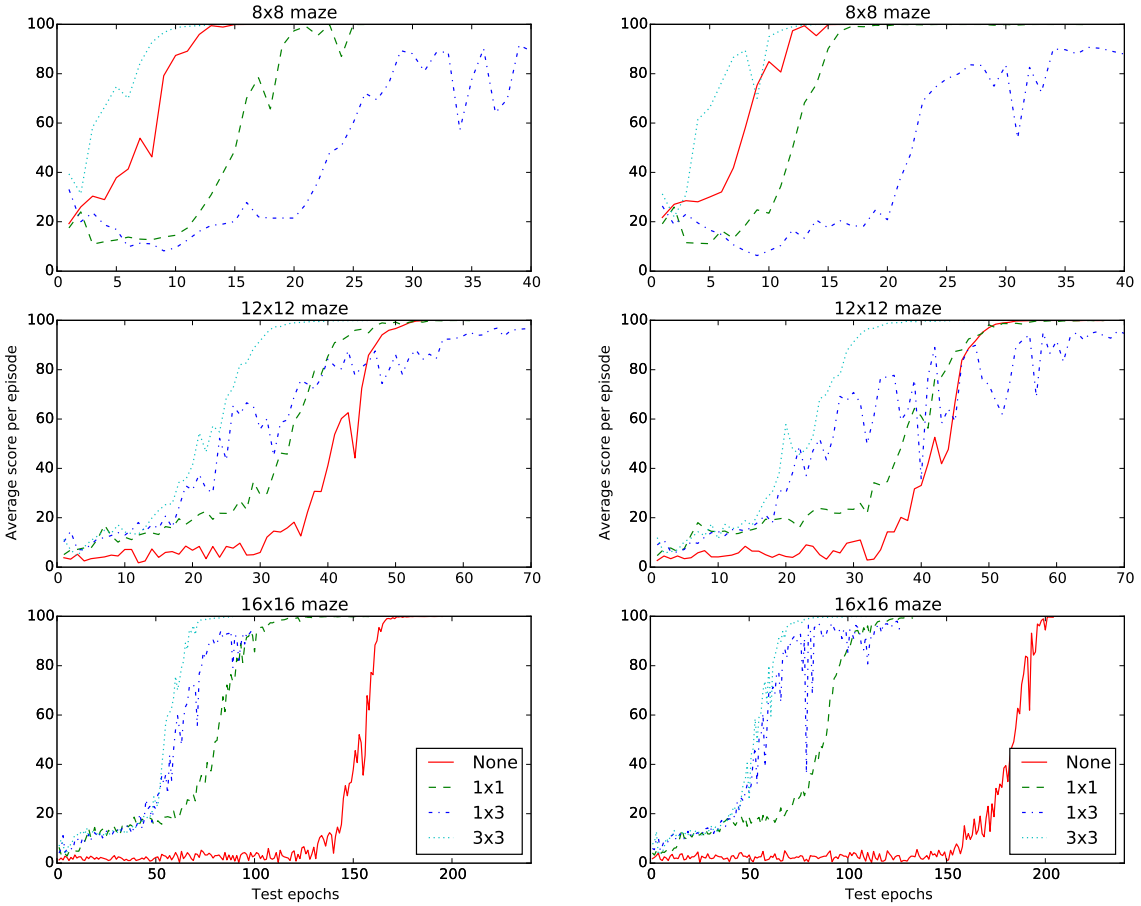
(b) Seed 5

Figure D.5. Convergence of DQN as a function of maze size and the number of enemies; the other two seeds.



(a) Seed 1 (b) Seed 2 (c) Seed 3

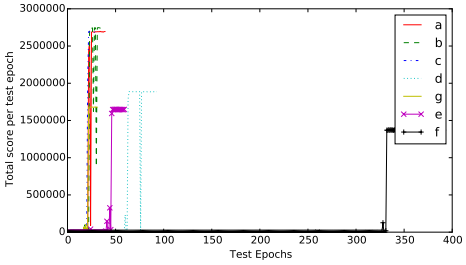
Figure D.6. Convergence of DQN as a function of maze size and the intermediate reward area; the first three seeds.



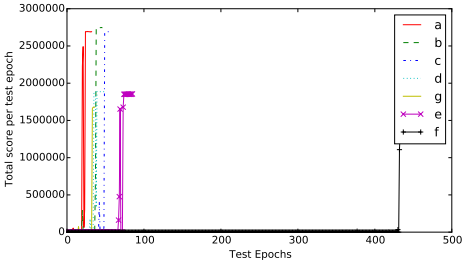
(a) Seed 4

(b) Seed 5

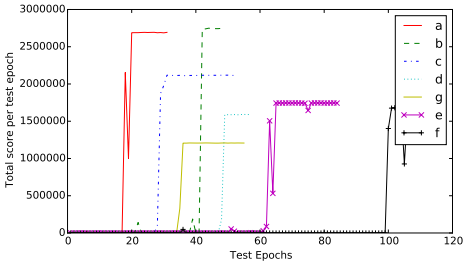
Figure D.7. Convergence of DQN as a function of maze size and the intermediate reward area; the other two seeds.



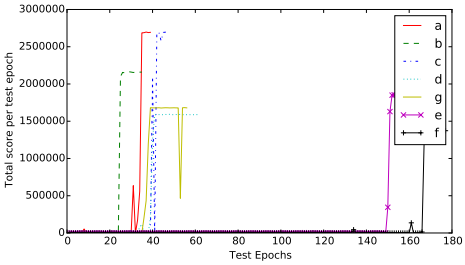
(a) Seed 1



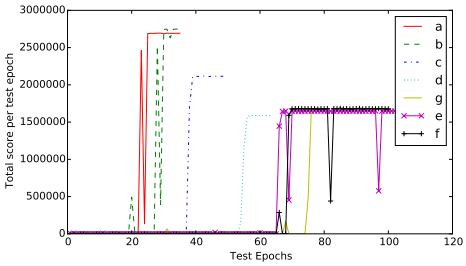
(b) Seed 2



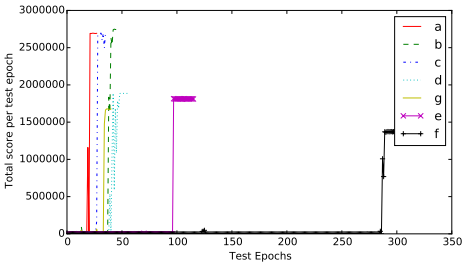
(c) Seed 3



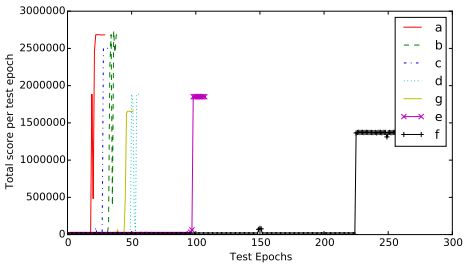
(d) Seed 4



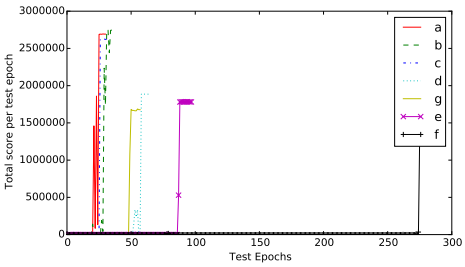
(e) Seed 5



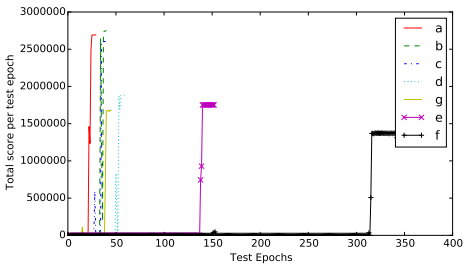
(f) Seed 6



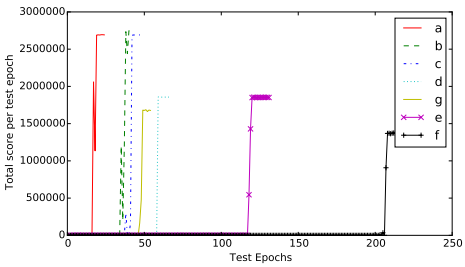
(g) Seed 7



(h) Seed 8

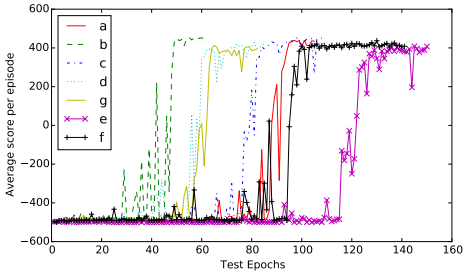


(i) Seed 9

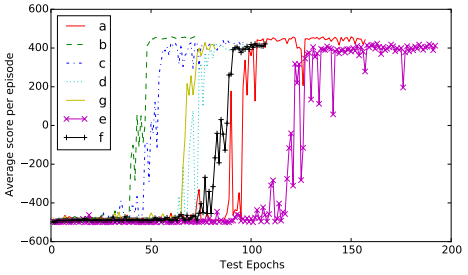


(j) Seed 10

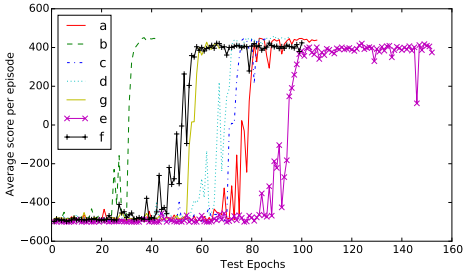
Figure D.8. Convergence of DQN on Pacman with no enemies.



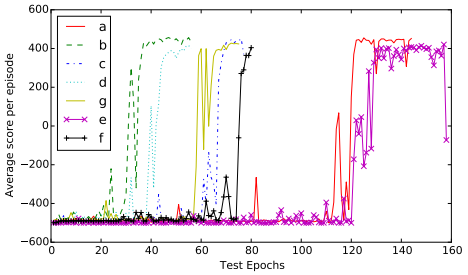
(a) Seed 1



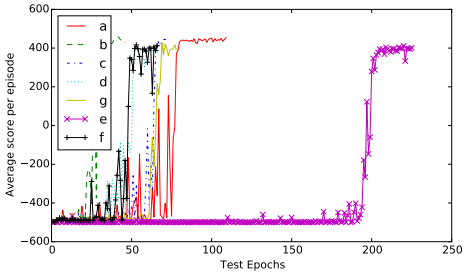
(b) Seed 2



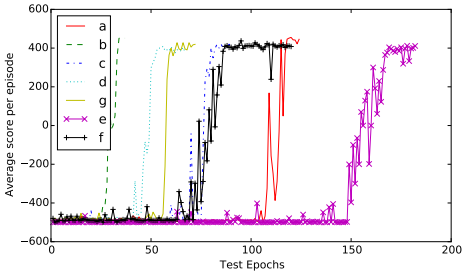
(c) Seed 3



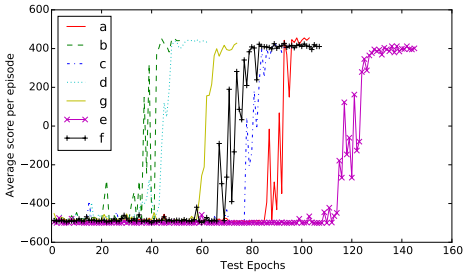
(d) Seed 4



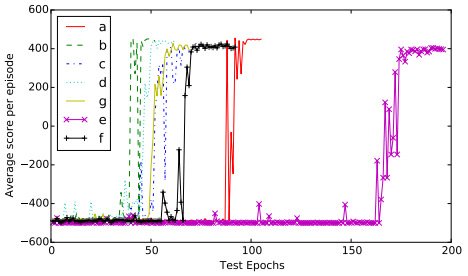
(e) Seed 5



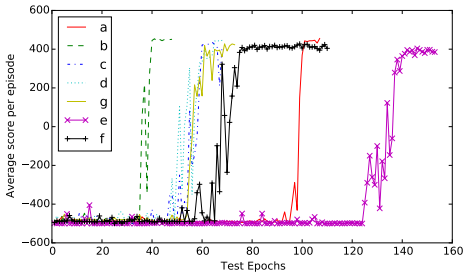
(f) Seed 6



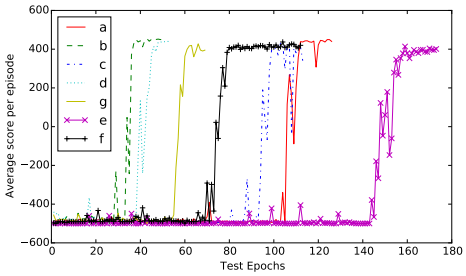
(g) Seed 7



(h) Seed 8



(i) Seed 9



(j) Seed 10

Figure D.9. Convergence of DQN on Pacman with two enemies.

APPENDIX E: IMPLEMENTATION DETAILS

We expand and use Nathan Sprague's replication [55] of Deep Q-learning in our experiments. This replication of the paper is written in the Python language and uses Theano, Lasagne and CuDNN frameworks for training the deep learning networks. The expansion of the replication is based on maze generation algorithms as well as the visualization of layers, our project is made publicly available [56].

We have not changed the original convolutional neural network architecture in our experiments. The weights and the input/output dimensions of this architecture are given in Table E.1. There is no subsampling or max pooling layer used between convolutional layers; convolution operations are calculated with a stride instead (stride is the weight shift size between two convolution operations).

Table E.1. DQN architecture layers.

Layer	**Input**	**Weights**	**Outputs**
Conv 1	84×84×4	4×8×8×32 (4 stride)	20×20×32
Conv 2	20×20×32	32×4×4×64 (2 stride)	9×9×64
Conv 3	9×9×64	64×3×3×64 (1 stride)	7×7×64
Fully Connected	7×7×64	3136×512	512
Fully Connected	512	512×18	18

The algorithm hyper-parameters are given in Table E.2. Only the *epsilon_decay* parameter is changed between experiments. It is selected as one million when running the ATARI 2600 games. In our maze experiments, *epsilon_decay* is adapted to each experiment. However, in Pacman maze experiments, it is set to 2.5 million and is not changed.

Table E.2. DQN hyper-parameters.

Parameter	**Explanation**	**Value**
epsilon_start	Epsilon greedy probability	1.0
epsilon_min	Epsilon cannot decrease below this value	0.1
epsilon_decay	Number of steps for epsilon to hit its minimum	changes
learning_rate	Global learning rate of RMSProp	0.00025
discount	Discount rate	0.99
rms_decay	Decay rate for RMSProp	0.95
rms_epsilon	Denominator epsilon for RMSProp	0.01
steps_per_epoch	Number of steps per epoch	250,000
steps_per_test	Number of steps per epoch when testing	125,000
update_frequency	Number of steps before each update	4
batch_size	Minibatch size	32
replay_memory_size	Number of experiences stored in replay	1,000,000
replay_start_size	Number of experiences to be gathered before training starts	50,000
resize	Input resize size and method	84×84 scale

APPENDIX F: ATARI 2600 ENVIRONMENT

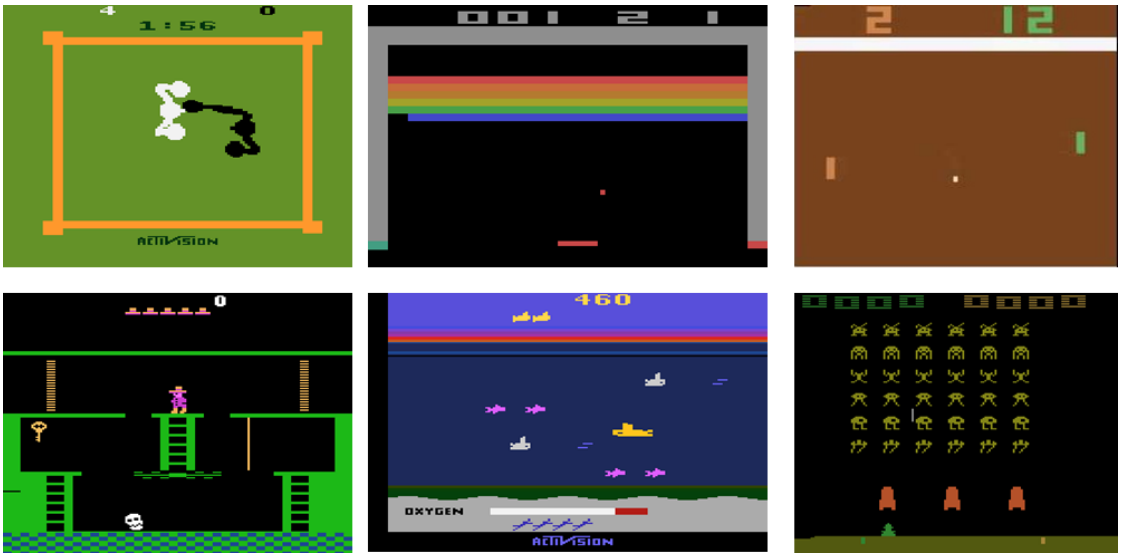


Figure F.1. Atari 2600 games.

The Arcade Learning Environment (ALE) is a simple object-oriented framework that allows researchers to develop AI agents for Atari 2600 games [57]. It provides game scores (rewards) as well as the RAM information and 160×210 display of the device at a certain time. It supports over 50 Atari 2600 games, screenshots from some are shown in Figure F.1. This environment has the ability to communicate with other programming languages. Atari 2600 has a controller with one 8-way joystick and one fire button. All possible combinations of the joystick and the fire button are supported by ALE and their integer representations are given in Table F.1.

Table F.1. All possible Atari 2600 actions.

noop (0)	down (5)	up-fire (10)	up-left-fire (15)
fire (1)	up-right (6)	right-fire (11)	down-right-fire (16)
up (2)	up-left (7)	left-fire (12)	down-left-fire (17)
right (3)	down-right (8)	down-fire (13)	
left (4)	down-left (9)	up-right-fire (14)	

It is possible to train an agent to play this game by using some hand-selected features from the RAM state of Atari 2600. The data extracted from the RAM can be normalized within some values to reduce the complexity of the problem. As we can see from Table F.2, there are $5 \times 34 \times 3 \times 7 \times 34 = 121,380$ Q values per action with processed features for Pong game, whereas there are originally $155 \times 165 \times 3 \times 7 \times 165 = 88,617,375$ Q values in case of no normalization. These numbers denote the complexity of the Pong game.

Table F.2. Hand-crafted RAM features for Pong game.

	ball x pos	ball y pos	ball y velocity	ball x velocity	player y pos
Raw data (hex)	32 - CD	26 - CB	09 - 0B	07 - 0D	26 - CB
Processed (dec)	0 - 4	0 - 33	0 - 2	0 - 6	0 - 33