



UNIVERSITY OF AGDER

Deep Reinforcement Learning using Capsules in Advanced Game Environments

PER-ARNE ANDERSEN

SUPERVISORS

Morten Goodwin
Ole-Christoffer Granmo

Master's Thesis
University of Agder, 2018
Faculty of Engineering and Science
Department of ICT

UiA

University of Agder
Master's thesis

Faculty of Engineering and Science
Department of ICT
© 2018 Per-Arne Andersen. All rights reserved

Abstract

Reinforcement Learning (RL) is a research area that has blossomed tremendously in recent years and has shown remarkable potential for artificial intelligence based opponents in computer games. This success is primarily due to vast capabilities of Convolutional Neural Networks (ConvNet), enabling algorithms to extract useful information from noisy environments. Capsule Network (CapsNet) is a recent introduction to the Deep Learning algorithm group and has only barely begun to be explored. The network is an architecture for image classification, with superior performance for classification of the MNIST dataset. CapsNets have not been explored beyond image classification.

This thesis introduces the use of CapsNet for Q-Learning based game algorithms. To successfully apply CapsNet in advanced game play, three main contributions follow. First, the introduction of four new game environments as frameworks for RL research with increasing complexity, namely Flash RL, Deep Line Wars, Deep RTS, and Deep Maze. These environments fill the gap between relatively simple and more complex game environments available for RL research and are in the thesis used to test and explore the CapsNet behavior.

Second, the thesis introduces a generative modeling approach to produce artificial training data for use in Deep Learning models including CapsNets. We empirically show that conditional generative modeling can successfully generate game data of sufficient quality to train a Deep Q-Network well.

Third, we show that CapsNet is a reliable architecture for Deep Q-Learning based algorithms for game AI. A capsule is a group of neurons that determine the presence of objects in the data and is in the literature shown to increase the robustness of training and predictions while lowering the amount training data needed. It should, therefore, be ideally suited for game plays. We conclusively show that capsules can be applied to Deep Q-Learning, and present experimental results of this method in the environments introduced. We further show that capsules do not scale as well as convolutions, indicating that CapsNet-based algorithms alone will not be able to play even more advanced games without improved scalability.

Table of Contents

Abstract	iii
Glossary	x
List of Figures	xii
List of Tables	xiii
List of Publications	xv
I Research Overview	1
1 Introduction	3
1.1 Motivation	3
1.2 Thesis definition	5
1.2.1 Thesis Goals	5
1.2.2 Hypotheses	5
1.2.3 Summary	5
1.3 Contributions	6
1.4 Thesis outline	7
2 Background	9
2.1 Artificial Neural Networks	10
2.1.1 Activation Functions	11
2.1.2 Optimization	11
2.1.3 Loss Functions	12
2.1.4 Hyper-parameters	13
2.2 Convolutional Neural Networks	14
2.2.1 Pooling	15
2.2.2 Summary	16
2.3 Generative Models	17
2.4 Markov Decision Process	18
2.5 Reinforcement Learning	19
2.5.1 Q-Learning	19
2.5.2 Deep Q-Learning	20

3 State-of-the-art	21
3.1 Deep Learning	22
3.2 Deep Reinforcement Learning	23
3.3 Generative Modeling	24
3.4 Capsule Networks	26
3.5 Game Learning Platforms	27
3.5.1 Summary	28
3.6 Reinforcement Learning in Games	29
 II Contributions	 31
4 Environments	33
4.1 FlashRL	35
4.2 Deep Line Wars	37
4.3 Deep RTS	40
4.4 Deep Maze	42
4.5 Flappy Bird	44
5 Proposed Solutions	45
5.1 Environments	47
5.2 Capsule Networks	49
5.3 Deep Q-Learning	52
5.4 Artificial Data Generator	54
 III Experiments and Results	 57
6 Conditional Convolution Deconvolution Network	59
6.1 Introduction	59
6.2 Deep Line Wars	61
6.3 Deep Maze	63
6.4 FlashRL: Multitask	65
6.5 Flappy Bird	67
6.6 Summary	69
7 Deep Q-Learning	71
7.1 Experiments	72
7.2 Deep Line Wars	80
7.3 Deep RTS	80
7.4 Deep Maze	81
7.5 FlashRL: Multitask	81
7.6 Flappy Bird	82
7.7 Summary	82
8 Conclusion and Future Work	85
8.1 Conclusion	86

8.2 Future Work	88
References	93
Appendices	95
A Hardware Specification	95
 IV Publications	 97
A Towards a Deep Reinforcement Learning Approach for Tower Line Wars	99
B FlashRL: A Reinforcement Learning Platform for Flash Games	115

Glossary

AGI Artificial General Intelligence. 28

AI Artificial Intelligence. 3, 7, 28, 31, 43

ANN Artificial Neural Network. 9–14, 19, 20, 22, 30, 41, 53, 74

CapsNet Capsule Network. iii, x, xi, 5–7, 27, 35, 49, 53, 55, 75, 77–85, 88–90

CCDN Conditional Convolution Deconvolution Network. ix–xi, 58–60, 63–65, 68, 70, 72, 74, 75, 87, 89–91

CGAN Conditional Generative Adversarial Network. 25

ConvNet Convolutional Neural Network. iii, x, xi, 4, 5, 14–16, 22, 25, 27, 53, 66, 75, 77–85, 87, 88

DDQN Double Deep Q-Learning. 23

DL Deep Learning. 9, 42

DNN Deep Neural Network. 10, 11

DQN Deep Q-Network. x, xi, 19, 20, 23, 48, 49, 56, 58, 64, 75, 77–85, 89, 90

DRL Deep Reinforcement Learning. 9, 12, 22, 28, 37, 38, 49, 87

FCN Fully-Connected Network. 15

GAN Generative Adversarial Networks. 8, 24–26

MDP Markov Decision Process. 9, 18

MLP Multilayer Perceptron. 10

MSE Mean Squared Error. 12, 13, 58, 64, 68, 70, 74

ReLU Rectified Linear Unit. 11

RL Reinforcement Learning. iii, 3–9, 18, 19, 21, 26, 28–31, 35, 37, 38, 41, 42, 44–49, 51–53, 55, 58, 72, 74, 75, 87–91

RTS Real Time Strategy Games. 3, 5–7, 29, 43, 44, 87–90

SDG Stochastic Gradient Decent. 12, 13, 58, 65, 75

List of Figures

2.1	Deep Neural network with two hidden layers	10
2.2	Single Perceptron	10
2.3	Loss functions	12
2.4	Convolutional Neural Network for classification	14
2.5	Fully-Connected Neural Network for classification	15
2.6	MAX and AVG Pooling operation	16
2.7	Overview: Generative Model	17
3.1	Illustration of Generative Adversarial Network Model	24
3.2	Illustration of Conditional Generative Adversarial Network Model	25
4.1	Environment field of focus	34
4.2	FlashRL: Architecture	35
4.3	FlashRL: Frame-buffer Access Methods	36
4.4	FlashRL: Available environments	36
4.5	Deep Line Wars: Graphical User Interface	37
4.6	Deep Line Wars: Game-state representation	38
4.7	Deep Line Wars: Game-state representation using heatmaps	38
4.8	Deep RTS: Graphical User Interface	40
4.9	Deep RTS: Architecture	41
4.10	Deep Maze: Graphical User Interface	42
4.11	Deep Maze: State-space complexity	43
4.12	Flappy Bird: Graphical User Interface	44
5.1	Proposed Deep Reinforcement Learning ecosystem	46
5.2	Architecture: gym-cair	47
5.3	Capsule Networks: Parameter count for different input sizes	50
5.4	Capsule Networks: Architecture	50
5.5	Architecture: CCDN	55
6.1	CCDN: Deep Line Wars: Training Performance	61
6.2	CCDN: Deep Maze: Training Performance	63
6.3	CCDN: FlashRL: Training Performance	65
6.4	CCDN: Flappy Bird: Training Performance	67
7.1	DQN-CapsNet: Deep Line Wars	72
7.2	DQN-ConvNet: Deep Line Wars	73
7.3	DQN-CapsNet: Deep RTS	74

7.4	DQN-ConvNet: Deep RTS	75
7.5	DQN-CapsNet: Deep Maze	76
7.6	DQN-ConvNet: Deep Maze	77
7.7	DQN-CapsNet: Flappy Bird	78
7.8	DQN-ConvNet: Flappy Bird	79
7.9	DQN-CapsNet: Agent building defensive due to low health in Deep Line Wars . . .	80
7.10	DQN-CapsNet: Agent attempting to find the shortest path in a 25×25 Deep Maze	81

List of Tables

2.1	Equations of activation function	11
3.1	Summary of researched platforms	28
4.1	Deep Line Wars: Representation modes	39
4.2	Deep RTS: Player Resources	41
5.1	Capsule Networks: Dimension Comparison	49
5.2	Deep Q-Learning architectures in testbed	52
5.3	Deep Q-Learning architectures	52
5.4	Deep Q-Learning hyper-parameters	53
5.5	Proposed prediction cycle for CCDN	55
6.1	CCDN: Deep Line Wars	62
6.2	CCDN: Deep Maze	64
6.3	CCDN: FlashRL: Multitask	66
6.4	CCDN: Flappy Bird	68
7.1	DQN: Hyper-parameters	71
7.2	Comparison of DQN-CapsNet, DQN-ConvNet, and Random accumulative reward (Higher is better)	83

List of Publications

A. Towards a Deep Reinforcement Learning Approach for Tower Line Wars	99
B. FlashRL: A Reinforcement Learning Platform for Flash Games	115

Part I

Research Overview

Chapter 1

Introduction

1.1 Motivation

Despite many advances in Artificial Intelligence (AI) for games, no universal Reinforcement Learning (RL) algorithm can be applied to advanced game environments without extensive data manipulation or customization. This includes traditional Real-Time Strategy (RTS) games such as Warcraft III, Starcraft II, and Age of Empires. RL has been applied to simpler games such as the Atari 2600 platform but is to the best of our knowledge not successfully applied to more advanced games. Further, existing game environments that target AI research are either overly simplistic such as Atari 2600 or complex such as Starcraft II.

RL has in recent years had tremendous progress in learning how to control agents from high-dimensional sensory inputs like images. In simple environments, this has been proven to work well [36], but are still an issue for advanced environments with large state and action spaces [34]. In environments where the objective is easily observable, there is a short distance between the action and the reward which fuels the learning [21]. This is because the consequence of any action is quickly observed, and then easily learned. When the objective is complicated, the game objectives still need to be mapped to a reward, but it becomes far less trivial [24]. For the Atari 2600 game Ms. Pac-Man this was solved through a hybrid reward architecture that transforms the objective to a low-dimensional representation [59]. Similarly, the OpenAI's bot is able to beat world's top professionals at one versus one in DotA 2. It uses an RL algorithm and trains this with self-play methods, learning how to predict the opponents next move.

Applying RL to advanced environments is challenging because the algorithm must be able to learn features from a high-dimensional input, in order act correctly within the environment [15]. This is solved by doing trial and error to gather knowledge about the mechanics of the environment. This process is slow and unstable [37]. Tree-Search algorithms have been successfully applied to board games such as Tic-Tac-Toe and Chess, but fall short for environments with large state-spaces [8]. This is a problem because the grand objective is to use these algorithms in real-world environments, that are often complex by nature. Convolutional Neural Networks (ConvNet) [28] solves complexity problems but faces several challenges when it comes to interpreting the environment data correctly.

The primary motivation of this thesis is to create a foundation for RL research in advanced environments, Using generative modeling to train artificial neural networks, and to use the Capsule Network architecture in RL algorithms.

1.2 Thesis definition

The primary objective of this thesis is to perform **Deep Reinforcement Learning using Capsules in Advanced Game Environments**. The research is split into six goals following the thesis hypotheses.

1.2.1 Thesis Goals

Goal 1: *Investigate the state-of-the-art research in the field of Deep Learning, and learn how Capsule Networks function internally.*

Goal 2: *Design and develop game environments that can be used for research into RL agents for the RTS game genre.*

Goal 3: *Research generative modeling and implement an experimental architecture for generating artificial training data for games.*

Goal 4: *Research the novel CapsNet architecture for MNIST classification and combine this with RL problems.*

Goal 5: *Combine Deep-Q Learning and CapsNet and perform experiments on environments from Achievement 2.*

Goal 6: *Combine the elements of Goal 3 and Goal 5. The goal is to train an RL agent with artificial training data successfully.*

1.2.2 Hypotheses

Hypothesis 1: *Generative modeling using deep learning is capable of generating artificial training data for games with a sufficient quality.*

Hypothesis 2: *CapsNet can be used in Deep Q-Learning with comparable performance to ConvNet based models.*

1.2.3 Summary

The first goal of this thesis is to create a learning platform for RTS game research. Second, to use generative modeling to produce artificial training data for RL algorithms. The third goal is to apply CapsNets to Deep Reinforcement Learning algorithms. The hypothesis is that its possible to produce artificial training data, and that CapsNets can be applied to Deep Q-Learning algorithms.

1.3 Contributions

This thesis introduces four new game environments, *Flash RL*¹, *Deep Line Wars*², *Deep RTS*, and *Deep Maze*. These environments integrates well with OpenAI GYM, creating a novel learning platform that targets *Deep Reinforcement Learning for Advanced Games*.

CapsNet is applied to RL algorithms and provides new insight on how CapsNet performs in problems beyond object recognition. This thesis presents a novel method that use generative modeling to train RL agents using artificial training data.

There is to the best of our knowledge no documented research on using CapsNet in RL problems, nor are there environments specifically targeted RTS AI research.

¹Proceedings of the 30th Norwegian Informatics Conference, Oslo, Norway 2017

²Proceedings of the 37th SGAI International Conference on Artificial Intelligence, Cambridge, UK, 2017

1.4 Thesis outline

Chapter 2 provides preliminary background research for Artificial Neural Networks (2.1, 2.2), Generative Models (2.3), Markov Decision Process (2.4), and Reinforcement Learning (2.5).

Chapter 3 investigates the current state-of-the-art in Deep Neural Networks (3.1), RL (3.2), GAN (3.3) and Game environments (3.5).

Chapter 4 outlines the technical specifications for the new game environments Flash RL (4.1), Deep Line Wars (4.2), Deep RTS (4.3), and Maze (4.4). In addition, a well established game environment (Section 4.5) is introduced to validate experiments conducted in this thesis.

Chapter 5 introduces the proposed solutions for the goals defined in Section 1.2. Section 5.1 outlines how the environments are presented as a learning platform. Section 5.2 introduces the proposal to use Capsules in RL. Section 5.3 describes the Deep Q-Learning algorithm and the implementations used for the experiments in this thesis. Finally, the artificial training data generator is outlined in Section 5.4.

Chapter 6 and 7 shows experimental results from the work presented in Chapter 5.

Chapter 8 concludes the thesis hypotheses and provides a summary of the work done in this thesis. Section 8.2 outlines the road-map for future research related to the thesis.

Chapter 2

Background

Deep Learning (DL) is a branch of machine learning algorithms that recently became popularized due to the exponential growth in available computing power. DL is unique in that it is designed to learn data representations, as opposed to task-specific algorithms. Methods from DL are frequently used in RL algorithms, creating a new branch called *Deep Reinforcement Learning* (DRL). Artificial Neural Networks (ANN) are used at its core, utilizing the most novel DL techniques to gain state-of-the-art capabilities.

This chapter outlines background theory for topics related to the research performed later in this thesis. Section 2.1 shows how Artificial Neural Networks work, moving onto computer vision with Convolutional Neural Networks in Section 2.2. Section 2.4 outlines the theory behind the Markov Decision Process (MDP) and how it is used in RL.

2.1 Artificial Neural Networks

An Artificial Neural Network (ANN) is a computing system that is inspired by how the biological nervous systems, such as the brain, function [19]. ANNs are composed of an interconnected network of neurons that pass data to its next layer when stimulated by an activation signal. When a network consists of several hidden layers, it is considered a *Deep Neural Network* (DNN). Figure 2.1 illustrates a Deep Multi-Layer Perceptron (MLP) with two hidden layers.

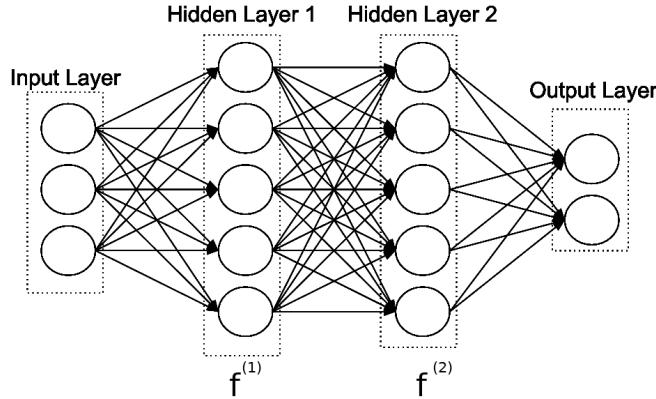


Figure 2.1: Deep Neural network with two hidden layers

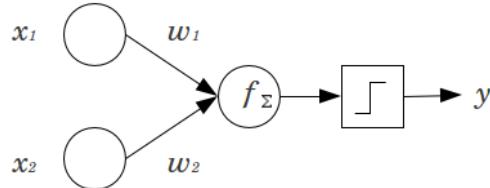


Figure 2.2: Single Perceptron

$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^n (w_i \cdot x_i) + b > 0 \\ 0 & \text{otherwise} \end{cases} \quad (2.1)$$

MLPs are considered a network because they are composed of many different functions. Each of these functions is represented as a *perceptron*. The combination of these functions gives us the ability to represent complex and high-dimensional functions [19]. Figure 2.2 illustrates a single perceptron from an MLP where $x_1, x_2 \dots x_n$ are inputs to the perceptron. Each of these inputs has a weight $w_1, w_2 \dots w_n$. Input x_n and weight w_n are multiplied into $z_n = x_n \cdot w_n$ and $z = \sum_{i=1}^n (z_n) + b$ where b is the bias value and z is the perceptron value. In Figure 2.2, the perceptron has a binary activation function (Equation 2.1), the neuron produce the value 1 for all z above 1, and 0 otherwise. There are several different activation functions that can be used in a perceptron network, see Section 2.1.1.

Name	Equation
TanH	$\tanh(z) = \frac{2}{1+e^{-2z}} - 1$
Softmax	$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^K e^{z_k}}$ for $j = 1 \dots K$
Sigmoid	$f(z) = \frac{1}{1+e^{-z}}$
Rectified Linear Unit (ReLU)	$f(z) = \begin{cases} 0 & \text{for } z < 0 \\ z & \text{otherwise} \end{cases}$
LeakyReLU	$f(z) = \begin{cases} z & \text{if } z > 0 \\ 0.01z & \text{otherwise} \end{cases}$
Binary	$f(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$

Table 2.1: Equations of activation function

2.1.1 Activation Functions

The purpose of an *Activation function* is often to introduce non-linearity into the network. It is proven that an DNN using only linear activations are equal to a single-layered network [42]. It is therefore natural to use non-linear activation functions in the hidden layers of an ANN if the goal is to predict non-linear functions. TanH and Rectified Linear Unit (ReLU) has proven to work well in ANNs [22,39,65], but there exist several other alternatives as illustrated in Table 2.1. Researchers do not understand to the full extent why an activation function works better for a particular problem and is why trial and error is used to find the best fit [33].

2.1.2 Optimization

Optimization in ANNs is the process of updating the weights of neurons in a network. In the optimization process, a *loss function* is defined. This function calculates the error/cost value of the network at the output layer. The error value describes the distance between the ground truth and the predicted value. For the network to improve, this error is *backpropagated* back through the network until each neuron has an error value that reflects its positive or negative contribution to the ground truth. Each neuron also calculates the gradient of its weights by multiplying output delta together with the input activation value. Weights are updated using *stochastic gradient descent* (SGD), which is a method of gradually descending the weight loss until reaching the optimal value.

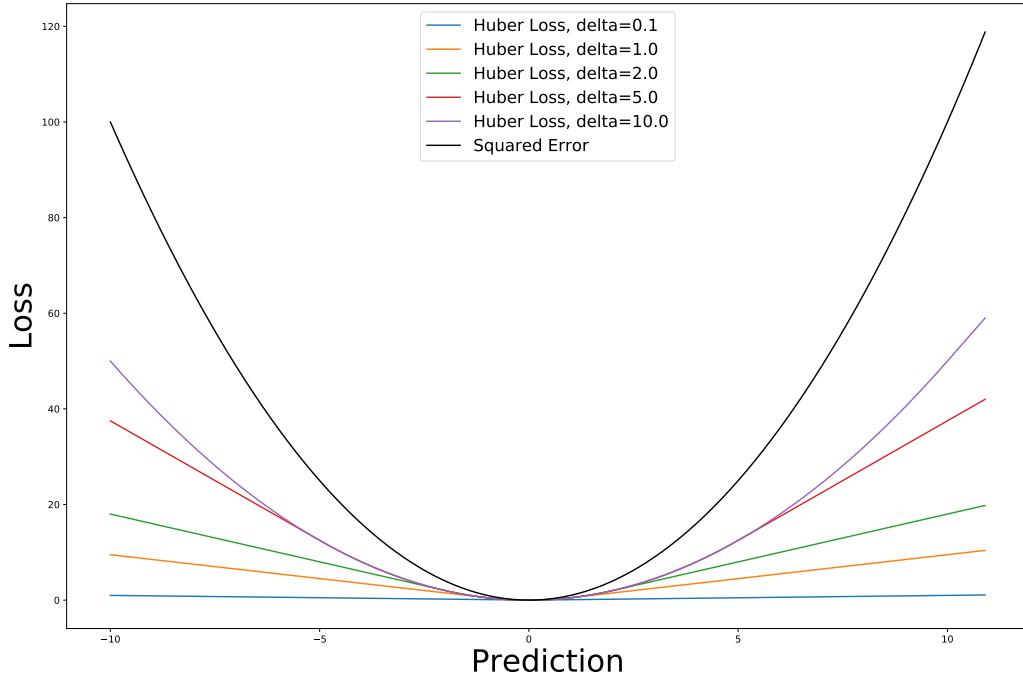


Figure 2.3: Loss functions

2.1.3 Loss Functions

To measure the inconsistency between the predicted value and the ground truth, a loss function is used in ANNs. The loss function calculates a positive number that is minimized throughout the optimization of the parameters¹ (Section 2.1.2). A loss function can be any mathematical formula, but there exist several well established functions. The performance varies on the classification task.

Mean Squared Error (MSE) is a quadratic loss function widely used in linear regression, and are also used in this thesis. Equation 2.2 is the standard form of MSE, where the goal is to minimize the residual squares ($y^{(i)} - \hat{y}^{(i)}$).

$$L = \frac{1}{n} \sum_{i=1}^n (y^{(i)} - \hat{y}^{(i)})^2 \quad (2.2)$$

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{for } |a| \leq \delta, \\ \delta(|a| - \frac{1}{2}\delta), & \text{otherwise} \end{cases} \quad (2.3)$$

Huber Loss is a loss function that is widely used in DRL. It is similar to MSE, but are less sensitive to data far apart from the ground truth. Equation 2.3 defines the function where a refers to the

¹Weights and Parameters are used interchangeably throughout the thesis

residuals and δ refers to its sensitivity. Figure 2.3 illustrates the difference between MSE and Huber Loss using different δ configurations.

2.1.4 Hyper-parameters

Hyper-parameters are tunable variables in ANNs. These parameters include learning rate, learning rate decay, loss function, and optimization algorithm like Adam, and SDG.

2.2 Convolutional Neural Networks

A Convolutional Neural Network is a novel ANN architecture that primarily reduces the compute power required to learn weights and biases for three-dimensional inputs. ConvNets are split into three layers:

1. Convolution layer
2. Activation layer
3. Pooling (Optional)

A Convolution layer has two primary components, *kernel* (parameters) and *stride*. The kernel consists of a weight matrix that is multiplied by the input values in its *receptive field*. The receptive field is the area of the input that the kernel is focused on. The kernel then slides over the input with a fixed stride. The stride value determines how fast this sliding happens. With a stride of 1, the receptive field move in the direction $x + 1$, and when at the end of the input x-axis, $y + 1$.

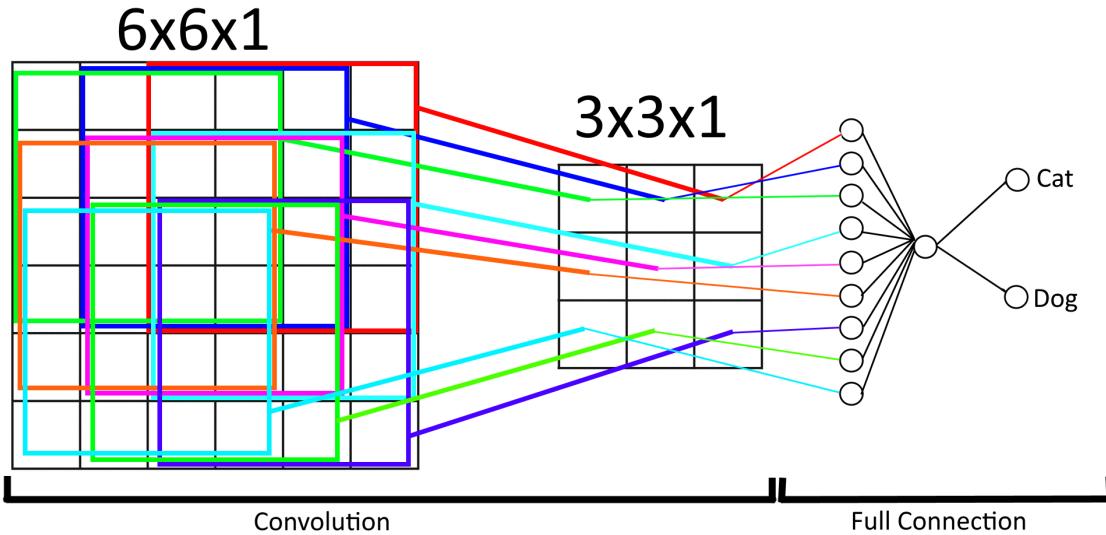


Figure 2.4: Convolutional Neural Network for classification

Consider a three-dimensional matrix representing an image of size $28 \times 28 \times 3$. In this example, the goal is to classify the image to be either a cat or dog. By using hyperparameters $\text{kernel} = 3 \times 3$ and $\text{stride} = 1 \times 1$, there are 32 *shared* parameters to be optimized. In contrast, a Fully-Connected network (FCN) with a single neuron layer, would have 2357 parameters to optimize. The reason why convolutions work is that it exploits what is called *feature locality*. ConvNets use *filters* that learn a specific feature of the input, for example, horizontal and vertical lines. For every convolutional layer added to the network, the information becomes more abstract, identifying objects and shapes. Figures 2.4 and 2.5 illustrate how a simple ConvNet is modeled compared to an FCN. The ConvNet

use a stride of 1×1 and a kernel size of 4×4 yielding a 3×3 output. This produces a total of 31 parameters to optimize, compared to 41 parameters in the FCN.

2.2.1 Pooling

Pooling is the operation of reducing the data resolution, often subsequent a convolution layer. This is beneficial because it reduces the number of parameters to optimize, hence decreasing the computational requirement. Pooling also controls overfitting by generalizing features. This makes the network capable of better handling spatial invariance [48].

There are several ways to perform pooling. *Max* and *Average* pooling are considered the most stable methods in whereas Max pooling is most used in state-of-the-art research [29]. Figure 2.6 illustrates the pooling process using Max and Average pooling on a $4 \times 4 \times X^2$ input volume. The hyperparameters for the pooling operation is *kernel* = 2×2 and *stride* = 2×2 applied to the input vector yields the resulting $2 \times 2 \times X$ output volume. This operation performed independently for

² X = Depth of the input volume

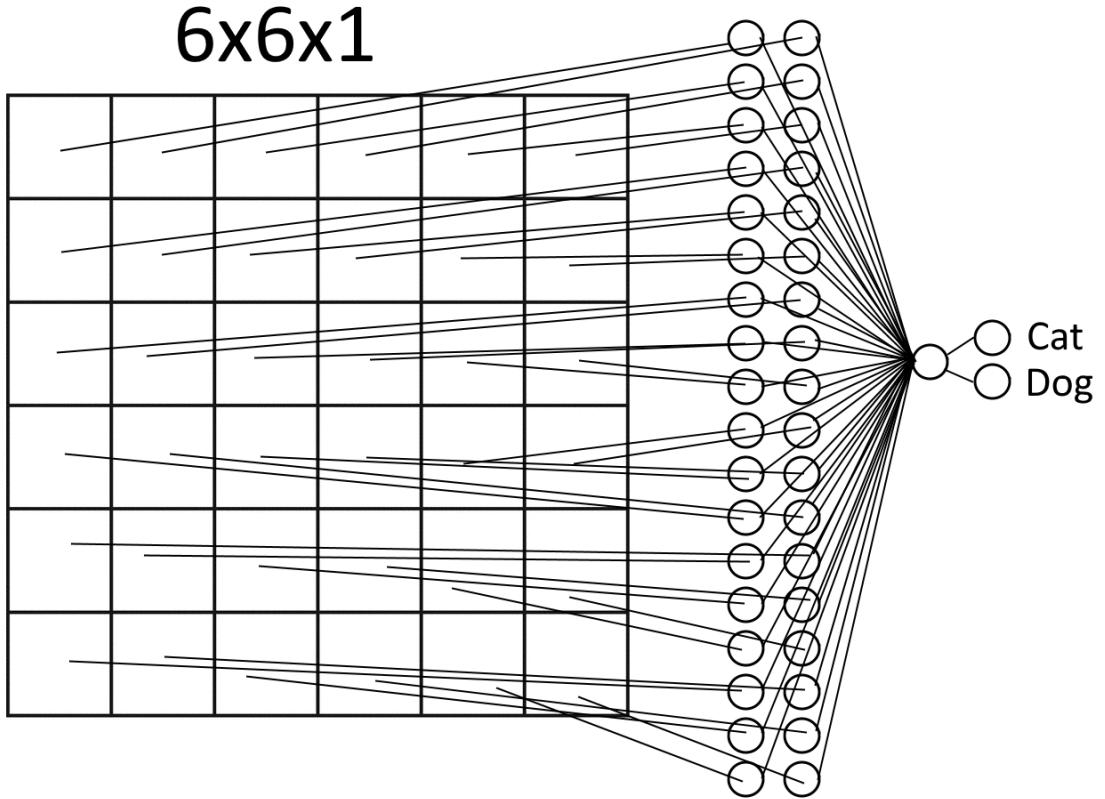


Figure 2.5: Fully-Connected Neural Network for classification

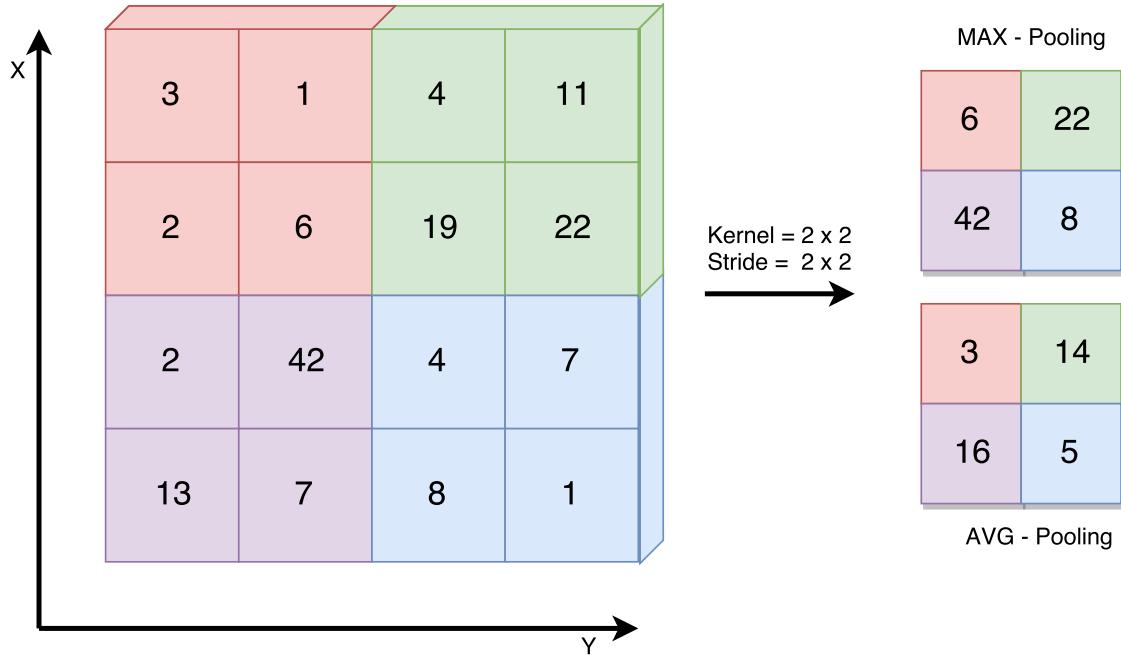


Figure 2.6: MAX and AVG Pooling operation

each depth slice of the input volume.

2.2.2 Summary

Historically, ConvNets drastically improved the performance of image recognition because it successfully reduced the number of parameters required, and at the same time preserving important features in the image. There are however several challenges, most notably that they are not rotation invariant. ConvNets are much more complicated than covered in this section, but this beyond the scope of this thesis. For an in-depth survey of the ConvNet architecture, refer to Recent Advances in Convolutional Neural Networks [12].

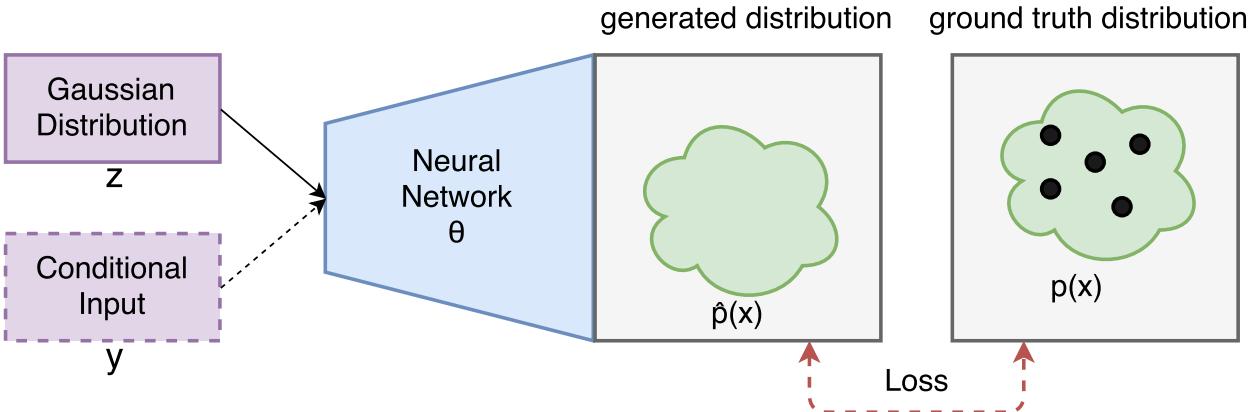


Figure 2.7: Overview: Generative Model

2.3 Generative Models

Generative Models are a series of algorithms trying to generate an artificial output based on some input, often randomized. Generative Adversarial Networks and Variational Autoencoder is two methods that have shown excellent results in this task. These methods have primarily been used in generating realistic images from various datasets like MNIST and CIFAR-10. This section will outline the theory in understanding the underlying architecture of generative models.

The objective of most Generative Models is to generate a distribution of data, that is close to the ground-truth distribution (the dataset). The Generative Model takes a Gaussian distribution z , as input, and outputs $\hat{p}(x)$ as illustrated in Figure 2.7. The goal is to find parameters θ that best matches the ground truth distribution with the generated distribution. Convolutional Neural Networks are often used in Generative Modeling, typically for models using noise as input. The model has several hidden parameters θ that is tuned via backpropagation methods like stochastic gradient descent. If the model reaches optimal parameters, $\hat{p}(x) = p(x)$ is considered true.

2.4 Markov Decision Process

MDP is a mathematical method of modeling decision-making within an *environment*. An environment defines a real or virtual world, with a set of rules. This thesis focuses on virtual environments, specifically, games with the corresponding game mechanic limitations. The core problem of MDPs is to find an optimal *policy* function for the decision maker (hereby referred to as an agent).

$$\underbrace{a}_{\text{Action}} = \underbrace{\pi(s)}_{\text{Policy } \pi \text{ for state } s} \quad (2.4)$$

Equation 2.4 illustrates how a decision/action is made using observed knowledge of the environmental state. The goal of the policy function is to find the decision that yields the best cumulative reward from the environment. MDP behaves like a Markov chain, hence gaining the *Markov Property*. The Markov property describes a system where future states only depend on the present and not the past. This enables MDP based algorithms to do iterative learning [54]. MDP is the foundation of how RL algorithms operate to learn the optimal behavior in an environment.

2.5 Reinforcement Learning

Reinforcement learning is a process where an agent performs actions in an environment, trying to maximize some cumulative reward [53] (see Section 2.4). RL differs from supervised learning because the ground truth is never presented directly. In RL there are *model-free* and *model-based* algorithms. In model-free RL, the algorithm must learn the environmental properties (the model) without guidance. In contrast, model-based RL is defined manually describing the features of an environment [10]. For model-free algorithms, the learning only happens in present time and the future must be *explored* before knowledge about the environment can be learned [11, 26].

This thesis focuses on *Q-Learning* algorithms, a model-free RL technique that may potentially solve difficult game environments. This section investigates the background theory of Q-Learning and extends this method to Deep Q-Learning (DQN), a novel algorithm that combines RL and ANN.

2.5.1 Q-Learning

Q-Learning is a model-free algorithm. This means that the MDP stays hidden throughout the learning process. The objective is to learn the optimal policy by estimating the action-value function $Q^*(s, a)$, yielding maximum expected reward in state s performing action a in an environment. The optimal policy can then be found by

$$\pi(s) = \text{argmax}_a Q^*(s, a) \quad (2.5)$$

Equation 2.5 is derived from finding the optimal utility of a state $U(s) = \max_a Q(s, a)$. Since the utility is the maximum value, the argmax of that same value qualifies as the optimal policy. The update rule for Q-Learning is based on value iteration:

$$Q(s, a) \leftarrow Q(s, a) + \underbrace{\alpha}_{\text{LR}} \left(\underbrace{R(s)}_{\text{Reward}} + \underbrace{\gamma}_{\text{Discount}} \underbrace{\max_{a'} Q(s', a') - Q(s, a)}_{\text{New Q - Old Q}} \right) \quad (2.6)$$

Equation 2.6 shows the iterative process of propagating back the estimated Q-value for each discrete time-step in the environment. α is the learning rate of the algorithm, usually low number between 0.001 and 0.00001. The reward function $R(s) \in \mathbb{R}$, and is often between $-1 < x < 1$ to increase learning stability. γ is the discount factor, discounting the importance of future states. The "old Q" is the estimated Q-Value of the starting state while the "new Q" estimates the future state. Equation 2.6 is guaranteed to converge towards the optimal action-value function, $Q_i \rightarrow Q^*$ as $i \rightarrow \infty$ [36, 53].

2.5.2 Deep Q-Learning

At the most basic level, Q-Learning utilizes a table for storing (s, a, r, s') pairs. Instead, a non-linear function approximation can be used to approximate $Q(s, a; \theta)$. This is called **Deep-Q Learning**. θ describes tunable parameters (weights) for the approximation. ANNs are used as an approximation method for retrieving values from the Q-Table but at the cost of stability. Using ANN is much like compression found in JPEG images. The compression is *lossy*, and information is lost at compression time. This makes DQN unstable, since values may be wrongfully encoded under training. In addition to value iteration, a loss function must be defined for the backpropagation process of updating the parameters.

$$L(\theta_i) = E[(r + \gamma \max_{a'} Q(s', a'; \theta_i) - Q(s, a; \theta_i))^2] \quad (2.7)$$

Equation 2.7 illustrates the loss function proposed by Minh et al [37]. It uses Bellmans equation to calculate the loss in gradient descent. To increase training stability, *Experience Replay* is used. This is a memory module that store memories from already explored parts of the state space. Experiences are often selected at random and then replayed to the neural network as training data. [36].

Chapter 3

State-of-the-art

This thesis focus on topics that are in active research, meaning that the state-of-the-art methods quickly advances. There have been many achievements in Deep Learning, primarily related to Computer Vision topics. This chapter investigates recent advancements in Deep Learning (3.1), Deep Reinforcement Learning (3.2), Generative Modeling (3.3), Capsule Networks (3.4) and Game Learning Platforms (3.5). In the success of Deep Learning, there have been several breakthroughs in popular game environments. Section 3.6 outlines the state-of-the-art of applying RL algorithms to game environments.

3.1 Deep Learning

Deep Learning has a long history, dating back to late 1980's. One of the first relevant papers on the area is *Learning representations by backpropagating errors* from Rumelhart et al. [44] In this paper, they illustrated that a deep neural network could be trained using backpropagation. The deep architecture proved that a neural network could successfully learn non-linear functions.

Yann LeCun started in the early 1990's research into Convolutional Neural Networks (ConvNet), with handwritten zip code classification as the primary goal [27]. He created the famous MNIST dataset, which is still widely used in the literature [28]. After ten years of research, LeCun et al. achieved state-of-the-art results on the MNIST dataset using ConvNets similar to those found in literature today [28]. But due to scaling issues with Deep ANNs, they were outperformed by classifiers like Support Vector Machines. It was not until 2006 with the paper *A fast learning algorithm for deep belief nets* by Hinton et al. that Deep Learning would appear again [17]. This paper showed how effectively train a deep neural network, by training one layer at a time. This was the beginning of Deep Neural Networks as they are known today.

For this thesis, Computer Vision is the most interesting architecture. There have been many advances in computer vision in the last couple of years. AlexNet [25], VGGNet [40] and ResNet [63] are models achieving state-of-the-art results in the ImageNet competition. These models are complex, but do a good job in image recognition. For DRL, there is to best of our knowledge no abstract model, that works for all environments. Therefore the model must be adapted to fit the environment at hand best.

3.2 Deep Reinforcement Learning

The earliest work found related to Deep Reinforcement Learning is *Reinforcement Learning for Robots Using Neural Networks*. This PhD thesis illustrated how an ANN could be used in RL to perform actions in an environment with delayed reward signals successfully. [31]

With several breakthroughs in computer vision in early 2010's, researchers started work on integrating ConvNets into RL algorithms. Q-Learning together with Deep Learning was a game-changing moment, and has had tremendous success in many single agent environments on the Atari 2600 platform. Deep Q-Learning (DQN) as proposed by Mnih et al. used ConvNets to predict the Q function. This architecture outperformed human expertise in over half of the games. [36]

Hasselt et al. proposed *Double DQN* (DDQN), which reduced the overestimation of action values in the Deep Q-Network. This led to improvements in some of the games on the Atari platform. [7]

Wang et al. then proposed a dueling architecture of DQN which introduced estimation of the value function and advantage function. These two functions were then combined to obtain the Q-Value. Dueling DQN were implemented with the previous work of van Hasselt et al. [43].

Harm van Seijen et al. recently published an algorithm called Hybrid Reward Architecture (HRA) which is a divide and conquer method where several agents estimate a reward and a Q-value for each state. The algorithm performed above human expertise in Ms. Pac-Man, which is considered one of the hardest games in the Atari 2600 collection and is currently state-of-the-art in the reinforcement learning domain. The drawback of this algorithm is that generalization of Minh et al. approach is lost due to a huge number of separate agents that have domain-specific sensory input. [59]

There have been few attempts at using Deep Q-Learning on advanced simulators made explicitly for machine-learning. It is probable that this is because there are very few environments created for this purpose.

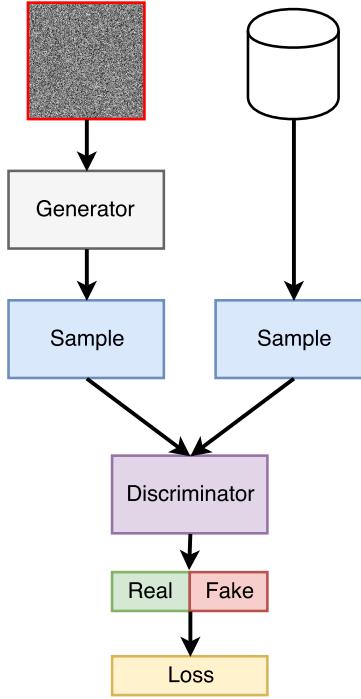


Figure 3.1: Illustration of Generative Adversarial Network Model

3.3 Generative Modeling

There are primarily three Generative models that are actively used in recent literature, GAN, Variational Autoencoders and Autoregressive Modeling. GAN show far better results than any other generative model and is the primary field of research for this thesis.

GAN show great potential when it comes to generating artificial images from real samples. The first occurrence of GAN was introduced in the paper *Generative Adversarial Networks* from Ian J. Goodfellow et al. [23]. This paper proposed a framework using a generator and discriminator neural network. The general idea of the framework is a two-player game where the generator generates synthetic images from noise and tries to fool the discriminator by learning to create authentic images, see Figure 3.1.

In future work, it was specified that the proposed framework could be extended from $p(x) \rightarrow p(x | c)$. This was later proposed in the paper *Conditional Generative Adversarial Nets* (CGAN) by Mirza et al. [35]. GAN is extended to a conditional model by demanding additional information y as input for the generator and discriminator. This enabled to condition the generated images on information like labels illustrated in Figure 3.2.

Radford et al. [33] proposed *Deep Convolutional Generative Adversarial Networks* (DCGAN) in *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. This paper improved on using ConvNets in unsupervised settings. Several architectural constraints were set to make training of DCGAN stable in most scenarios. This paper illustrated many great

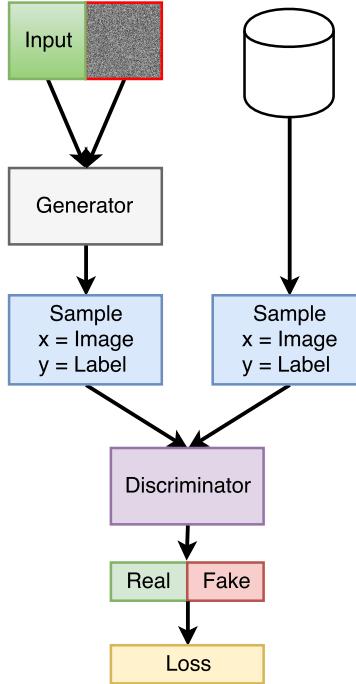


Figure 3.2: Illustration of Conditional Generative Adversarial Network Model

examples of images generated with DCGAN, for instance, state-of-the-art bedroom images.

In summer 2016, Salimans et al. (Goodfellow) presented *Improved Techniques for Training GANs* achieving state-of-the-art results in the classification of MNIST, CIFAR-10, and SVHN [46]. This paper introduced minibatch discrimination, historical averaging, one-sided label smoothing and virtual batch normalization.

There have been many advances in GAN between and after these papers. Throughout the research process of GANs, the most prominent architecture for our problem is Conditional GANs which enables us to condition the input variable x on variable y . The most recent paper on this topic is *Towards Diverse and Natural Image Descriptions via a Conditional GAN* from Dai et al. [9]. This paper focuses on captioning images using Conditional GANs. It produced captions that were of similar quality to human-made captions. In RL terms it is successfully able to learn a good policy for the dataset.

3.4 Capsule Networks

Capsule Neural Networks (CapsNet) is a novel deep learning architecture that attempts to improve the performance of image and object recognition. CapsNet is theorized to be far better at detecting rotated objects and requires less training data than traditional ConvNet. Instead of creating deep networks like for example ResNet-50, a Capsule layer is created, containing several sub-layers in depth. Each of these capsules has a group of neurons, where the objective is to learn a specific object or part of an object. When an image is inserted into the Capsule Layer, an iterative process of identifying objects begins. The higher dimension layers receive a signal from the lower dimensions. The higher dimension layer then determines which signal is the strongest and a connection is made between the winning signal (betting). This method is called *dynamic routing*. This routing-by-agreement ensures that features are mapped to the output, and preserves all input information at the same time.

Pooling in ConvNet is also a primitive form of routing, but information about the input is lost in the process. This makes pooling much more vulnerable to attacks compared to dynamic routing. In current state-of-the-art, CapsNet is explained as inverse graphics, where a capsule tries to learn an activity vector describing the probability that an object exists.

Capsule Networks are still only in infancy, and there is not well-documented research on this topic yet apart from state-of-the-art paper *Dynamic Routing Between Capsules* by Sabour et al. [45].

3.5 Game Learning Platforms

There exists several exciting game learning platform used to research state-of-the-art AI algorithms. The goal of these platforms is generally to provide the necessary platform for studying *Artificial General Intelligence* (AGI). AGI is a term used for AI algorithms that can perform well across several environments without training. DRL is currently the most promising branch of algorithms to solve AGI.

Bellemare et al. provided in 2012 a learning platform *Arcade Learning Environment* (ALE) that enabled scientists to conduct edge research in general deep learning [4]. The package provided hundreds of Atari 2600 environments that in 2013 allowed Minh et al. to do a breakthrough with Deep Q-Learning and A3C. The platform has been a critical component in several advances in RL research. [32, 36, 37]

The Malmo project is a platform built atop of the popular game *Minecraft*. This game is set in a 3D environment where the object is to survive in a world of dangers. The paper *The Malmo Platform for Artificial Intelligence Experimentation* by Johnson et al. claims that the platform had all characteristics qualifying it to be a platform for AGI research. [20]

ViZDoom is a platform for research in Visual Reinforcement Learning. With the paper *ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning* Kempka et al. illustrated that an RL agent could successfully learn to play the game *Doom*, a first-person shooter game, with behavior similar to humans. [41]

With the paper *DeepMind Lab*, Beattie et al. released a platform for 3D navigation and puzzle solving tasks. The primary purpose of Deepmind Lab is to act as a platform for DRL research. [3]

In 2016, Brockman et al. from OpenAI released GYM which they referred to as "*a toolkit for developing and comparing reinforcement learning algorithms*". GYM provides various types of environments from following technologies: Algorithmic tasks, Atari 2600, Board games, Box2d physics engine, MuJoCo physics engine, and Text-based environments. OpenAI also hosts a website where researchers can submit their performance for comparison between algorithms. GYM is open-source and encourages researchers to add support for their environments. [5]

OpenAI recently released a new learning platform called *Universe*. This environment further adds support for environments running inside VNC. It also supports running Flash games and browser applications. However, despite OpenAI's open-source policy, they do not allow researchers to add new environments to the repository. This limits the possibilities of running any environment. Universe is, however, a significant learning platform as it also has support for desktop games like Grand Theft Auto IV, which allow for research in autonomous driving [30].

Very recently Extensive Lightweight Flexible (ELF) research platform was released with the NIPS paper *ELF: An Extensive, Lightweight and Flexible Research Platform for Real-time Strategy Games*. This paper focuses on RTS game research and is the first platform officially targeting these types of games. [58]

Platform	Diversity	AGI	Advanced Environment(s)
ALE	Yes	Yes	No
Malmo Platform	No	No	Yes
ViZDoom	No	Yes	Yes
DeepMind Lab	No	No	Yes
OpenAI Gym	Yes	Yes	No
OpenAI Universe	Yes	Yes	Partially
ELF	No	No	Yes
(GYM-CAIR)	Yes	Yes	Yes

Table 3.1: Summary of researched platforms

3.5.1 Summary

Multiple interesting observations about current state-of-the-art in learning platforms for RL algorithms were found during our research. Table 3.1 describes the capabilities of each of the learning platform in the interest of fulfilling the requirements of this thesis. GYM-CAIR is included in this comparison and is further described in Chapter 4 and 5.

3.6 Reinforcement Learning in Games

Reinforcement Learning for games is a well-established field of research and is frequently used to measure how well an algorithm can perform within an environment. This section presents some of the most important achievements in Reinforcement Learning.

TD-Gammon is an algorithm capable of reaching an expert level of play in the board game *Backgammon* [56,57]. The algorithm was developed by Gerald Tesauro in 1992 at IBM's Thomas J. Watson Research Center. TD-Gammon consists of a three-layer ANN and is trained using an RL technique called *TD-Lambda*. TD-Lambda is a temporal difference learning algorithm invented by Richard S. Sutton [52]. The ANN iterates over all possible moves the player can perform and estimates the reward for that particular move. The action that yields the highest reward is then selected. TD-Gammon is one of the first algorithms to utilize self-play methods to improve the ANN parameters.

In late 2015, *AlphaGO* became the first algorithm to win against a human professional Go player. AlphaGO is an RL framework that uses Monte Carlo Tree search and two Deep Neural Networks for value and policy estimation [49]. Value refers to the expected future reward from a state assuming that the agent plays perfectly. The policy network attempts to learn which action is best in any given board configuration. The earliest versions of AlphaGO used training data from games played by human professionals. In the most recent version, *AlphaGO Zero*, only self-play is used to train the AI [51] In a recent update, AlphaGO was generalized to work for Chess and Shogi (Japanese Chess) only using 24 hours to reach superhuman level of play [50]

DOTA 2 is an advanced player versus player game where the player is controlling a hero unit. The game objective is to defeat the enemy heroes and destroy their base. In August 2017, OpenAI invented an RL based AI that defeated professional players in one versus one game. Training was done only using self-play, and the algorithm learned how to exploit game mechanics to perform well.

DeepStack is an algorithm that can perform an expert level play in Texas Hold'em poker. This algorithm uses tree-search in conjunction with neural networks to perform sensible actions in the game [38]. DeepStack is a general-purpose algorithm that aims to solve problems with imperfect information.

There have been several other significant achievements in AI, but these are not directly related to the use of RL algorithms. These include Deep Blue¹ and Watson from IBM.

¹Deep Blue is not AI

Part II

Contributions

Chapter 4

Environments

Simulated environments are a popular research method to conduct experiments on algorithms in computer science. These simulated environments are often tailored to the problem, and quickly proves, or disproves the capability of an algorithm. This chapter proposes four new game environments for deep learning research: *FlashRL*, *Deep Line Wars*, *Deep RTS*, and *Deep Maze*. The game *Flappy Bird* is introduced as a validation environment for experiments conducted in Chapter 7. Figure 4.1 illustrates that each of these environments has different goals, and the agent placed in these environments are challenged in several topics, for instance, multitasking, deep and shallow state interpretation and planning. This chapter creates a foundation for research into CapsNet based RL-algorithms in advanced game environments.

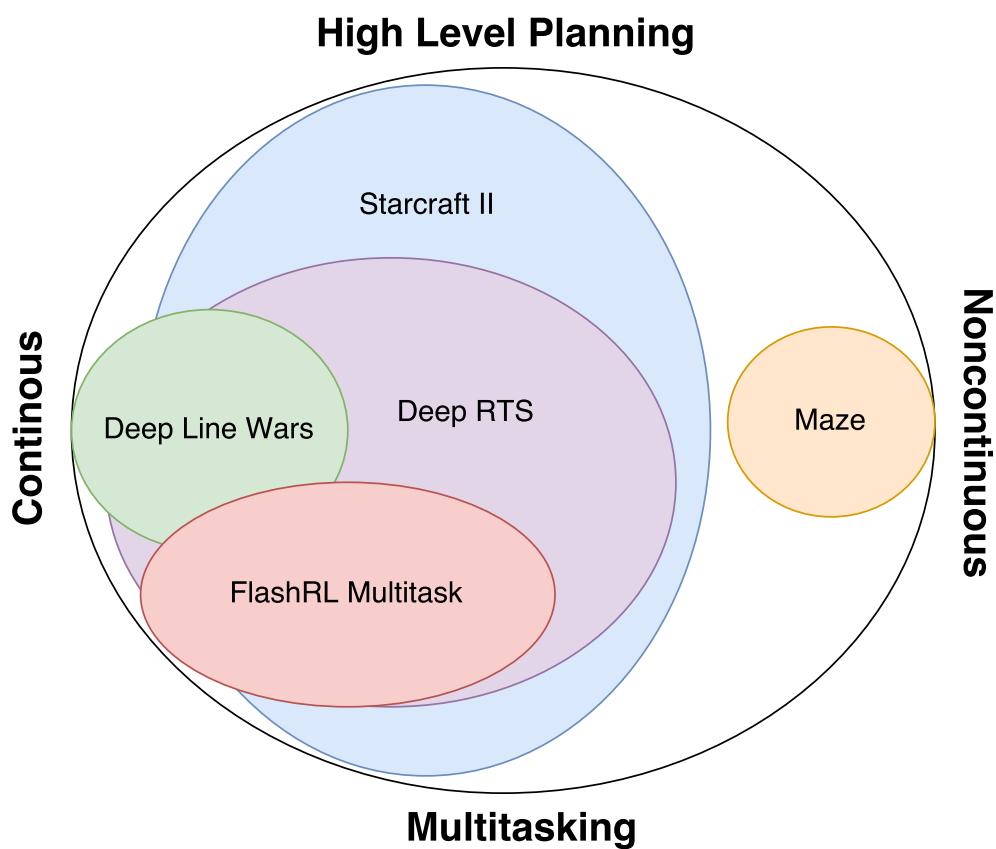


Figure 4.1: Environment field of focus

4.1 FlashRL

Adobe Flash is a multimedia software platform used for the production of applications and animations. The Flash run-time was recently declared deprecated by Adobe, and by 2020, no longer supported. Flash is still frequently used in web applications, and there are countless games created for this platform. Several web browsers have removed the support for the Flash runtime, making it difficult to access the mentioned game environments. Flash games are an excellent resource for machine learning benchmarking, due to size and diversity of its game repository. It is therefore essential to preserve the Flash run-time as a platform for RL.

Flash Reinforcement Learning (FlashRL) is a novel platform that acts as an input/output interface between Flash games and DRL algorithms. FlashRL enables researchers to interface against almost any Flash-based game environment efficiently.

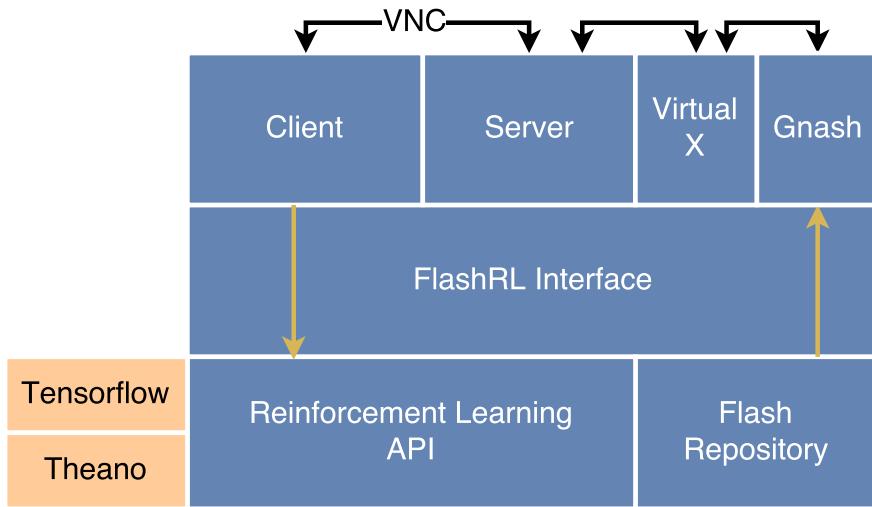


Figure 4.2: FlashRL: Architecture

The learning platform is developed primarily for Linux based operating systems but is likely to run on Cygwin with few modifications. There are several key components that FlashRL uses to operate adequately, see Figure 4.2. FlashRL uses XVFB to create a virtual frame-buffer. The frame-buffer acts like a regular desktop environment, found in Linux desktop distributions [18]. Inside the frame-buffer, a Flash game chosen by the researcher is executed by a third-party flash player, for example, *Gnash*. A VNC server serves the frame-buffer and enable FlashRL to access display, mouse and keyboard via the VNC protocol. The VNC Client *pyVLC* was specially made for this FlashRL. The code base originates from python-vnc-viewer [55]. The last component of FlashRL is the Reinforcement Learning API that allows the developer to access the input/output of the pyVLC. This makes it easy to develop sequenced algorithms by using API callbacks or invoke commands manually with threading.

Figure 4.3 illustrates two methods of accessing the frame-buffer from the Flash environment. Both

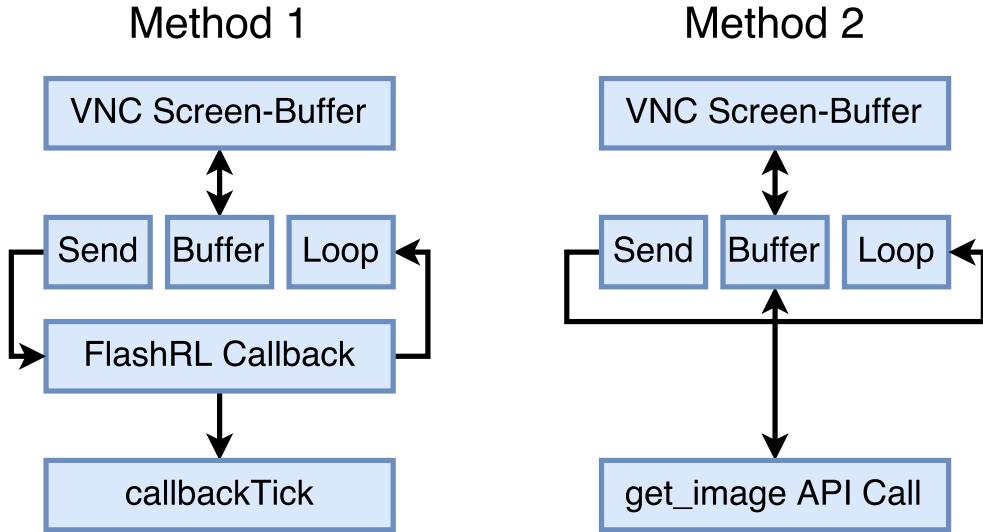


Figure 4.3: FlashRL: Frame-buffer Access Methods

approaches are sufficient to perform RL, but each has its strengths and weaknesses. Method 1 sends frames at a fixed rate, for example at 60 frames per second. The second method does not set any restrictions of how fast the frame-buffer can be captured. This is preferable for developers that do not require images from fixed time-steps because it demands less processing power per frame. The framework was developed with deep learning in mind and is proven to work well with Keras and Tensorflow [1].

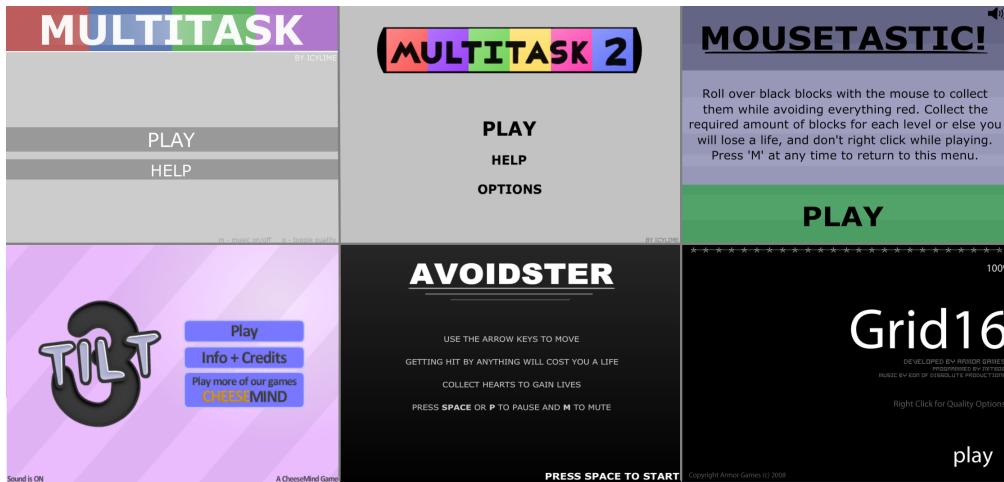


Figure 4.4: FlashRL: Available environments

There are close to a thousand game environments available for the first version of FlashRL. These game environments were gathered from different sources on the world wide web. FlashRL has a relatively small code-base and to preserve this size, the Flash repository is hosted at a remote site. Because of the large repository, not all games have been tested thoroughly. The game quality may therefore vary. Figure 4.4 illustrates tested games that yield a great value for DRL research.



Figure 4.5: Deep Line Wars: Graphical User Interface

4.2 Deep Line Wars

The game objective of Deep Line Wars is to invade the opposing player (hereby enemy) with mercenary units until all health points are depleted (see Figure 4.5). For every friendly unit that enters the red area on the map, the enemy health pool is reduced by one. When a player purchases a mercenary unit, it spawns at a random location inside the red area of the owners base. Mercenary units automatically move towards the enemy base. To protect the base, players can construct towers that shoot projectiles at the opponents mercenaries. When a mercenary dies, a fair percentage of its gold value is awarded to the opponent. When a player sends a unit, the income is increased by a percentage of the units gold value. As a part of the income system, players gain gold at fixed intervals. [2]

To successfully master game mechanics of Deep Line Wars, the player (agent) must learn

- offensive strategies of spawning units,
- defending against the opposing player’s invasions, and
- maintain a healthy balance between offensive and defensive to maximize income

The game is designed so that if the player performs better than the opponent in these mechanics, he is guaranteed to win over the opponent.

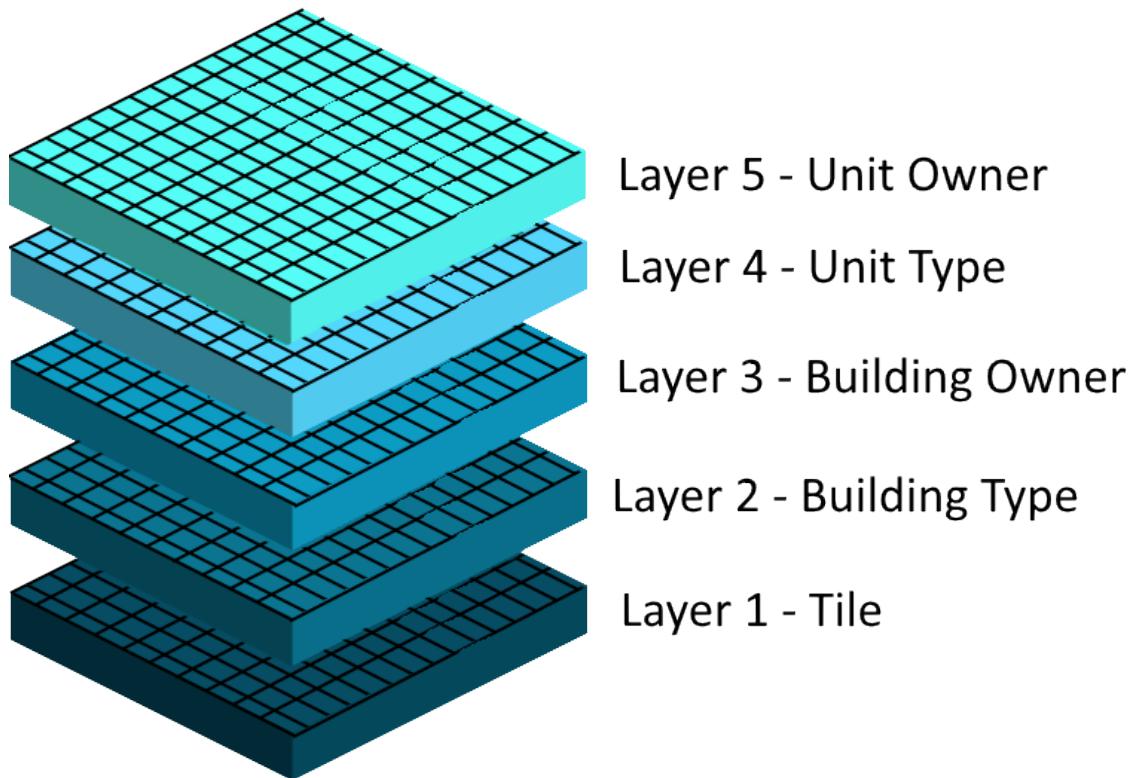


Figure 4.6: Deep Line Wars: Game-state representation



Figure 4.7: Deep Line Wars: Game-state representation using heatmaps

Representation	Matrix Size	Data Size
Image	$800 \cdot 600 \cdot 3$	1440000
Matrix	$10 \cdot 15 \cdot 5$	750
Heatmap RGB	$10 \cdot 15 \cdot 3$	450
Heatmap Grayscale	$10 \cdot 15 \cdot 1$	150

Table 4.1: Deep Line Wars: Representation modes

Because the game is specifically targeted towards RL research, the game-state is defined as a multi-dimensional matrix. This way, it is trivial to input the game-state directly into ANN models. Figure 4.6 illustrates how the game state is constructed. This state is later translated into graphics, seen in Figure 4.5. It is beneficial to directly access this information because it requires less data preprocessing compared to using raw game images. Deep Line Wars also features abstract state representation using heat-maps, seen in Figure 4.7. By using heatmaps, the state-space is reduced by a magnitude, compared to raw images. Heatmaps can better represent the true objective of the game, enabling faster learning for RL algorithms [47].

In Deep Line Wars, there are primarily four representation modes available for RL. Table 4.1 shows that there is considerably lower data size for grayscale heatmaps. Effectively, the state-space can be reduced by 9600%, when no data preprocessing is done. Heatmaps seen in 4.7 define

- red pixels as friendly buildings,
- green pixels as enemy units, and
- teal pixels as the mouse cursor.

When using grayscale heatmaps, RGB values are squashed into a one-dimensional matrix with values ranging between 0 and 1. Economy drastically increases the complexity of Deep Line Wars, and it is challenging to present only using images correctly. Therefore a secondary data structure is available featuring health, gold, lumber, and income. This data can then be feed into a hybrid DL model as an auxiliary input [61].

4.3 Deep RTS

RTS games are considered to be the most challenging games for AI algorithms to master [60]. With colossal state and action-spaces, in a continuous setting, it is nearly impossible to estimate the computational complexity of games such as Starcraft II.

The game objective of Deep RTS is to build a base consisting of a Town-Hall and then expand the base to gain the military power to defeat the opponents. Each of the players starts with a worker. Workers can construct buildings and gather resources to gain an economic advantage.



Figure 4.8: Deep RTS: Graphical User Interface

The game mechanics consist of two main terminologies, *Micro* and *Macro* management. The player with the best ability to manage their resources, military, and defensive is likely to win the game. There is a considerable leap from mastering Deep Line Wars to Deep RTS, much because Deep RTS features more than two players.

Player Resources					
Property:	Lumber	Gold	Oil	Food	Units
Value Range:	0 - 10^6	0 - 10^6	0 - 10^6	0 - 200	0 - 200

Table 4.2: Deep RTS: Player Resources

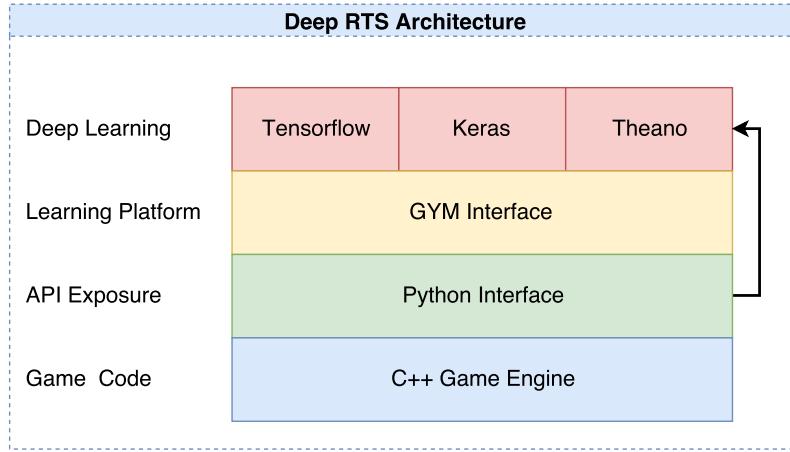


Figure 4.9: Deep RTS: Architecture

The game interface displays relevant statistics meanwhile a game session is running. These statistics show the *action distribution*, *player resources*, *player scoreboard* and a *live performance graph*. The action distribution keeps track of which actions a player has performed in the game session. These statistics are stored to the hard-drive after a game has reached the terminal state. Player Resources (Table 4.2), are shown at the top bar of the game. Player Scoreboard indicates the overall performance of each of the players, measured by kills, defensive points, offensive points and resource count. Deep RTS features several hotkeys for moderating the game-settings like game-speed and state representation. The hotkey menu is accessed by pressing the G-hotkey.

Deep RTS is an environment developed as an intermediate step between Atari 2600 and the famous game Starcraft II. It is designed to measure the performance in RL algorithms, while also preserving the game goal. Deep RTS is developed for high-performance simulation of RTS scenarios. The game engine is developed in C++ for performance but has an API wrapper for Python, seen in Figure 4.9. It has a flexible configuration to enable different AI approaches, for instance online and offline RL. Deep RTS can represent the state as raw game images (C++) and as a matrix, which is compatible with both C++ and Python.

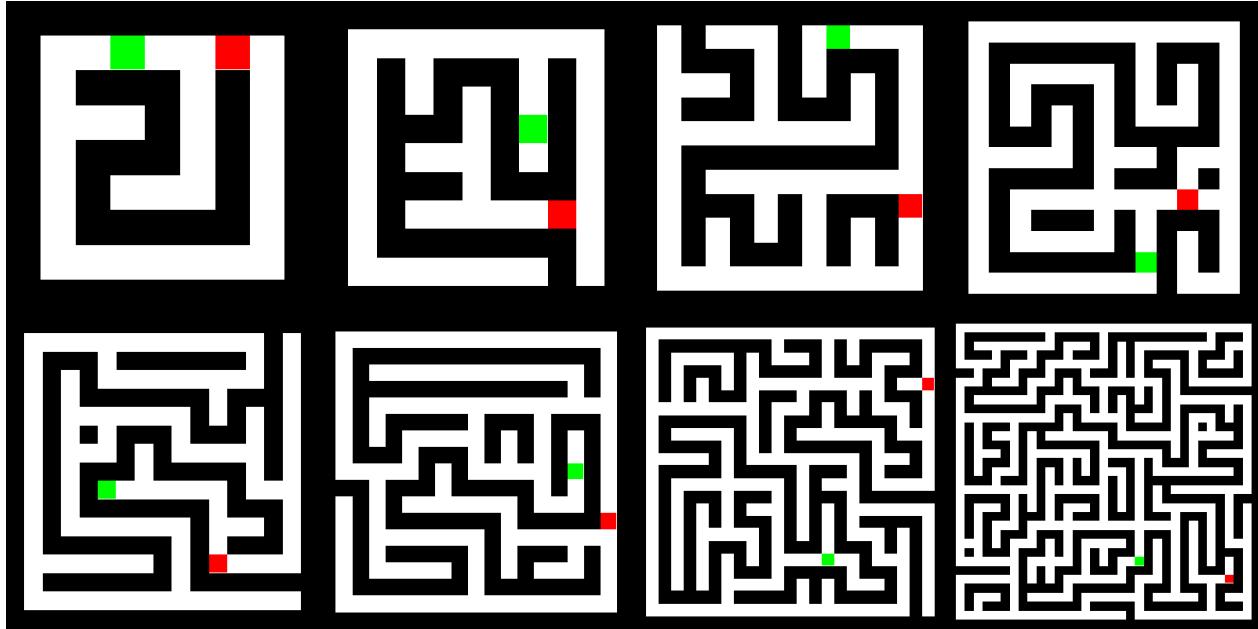


Figure 4.10: Deep Maze: Graphical User Interface

4.4 Deep Maze

Deep Maze is a game environment designed to challenge RL agents in the *shortest path problem*. Deep Maze defines the problem as follows:

- How can the agent optimally navigate through any fully-observable maze?

The environment is simple, but becomes drastically more complicated when the objective is to find the optimal policy $\pi^*(s)$ where s = state for all the maze configurations.

There are multiple difficulty levels for Deep Maze in two separate modes; deterministic and stochastic. In the deterministic mode, the maze structure is never changed from game to game. Stochastic mode randomizes the maze structure for every game played. There are multiple size configurations, ranging from 7×7 to 55×55 in width and height, seen in Figure 4.10.

Figure 4.11 illustrates how the theoretical maximum state-space set S of Deep Maze increase with maze size. This is calculated by performing following binomial: $S = \binom{\text{width} \times \text{height}}{\text{player} + \text{goal}} = \binom{w \times h}{2}$. This is however reduced depending on the maze composition, where dense maze structures are generally less complex to solve theoretically.

The simulation is designed for performance so that each discrete time step is calculated with fewest possible CPU cycles. The simulation is estimated to run at 3 000 000 ticks per second with modern hardware. This allows for fast training of RL algorithms.

From an RL point of view, Deep Maze challenges an agent in state-interpretation and navigation,

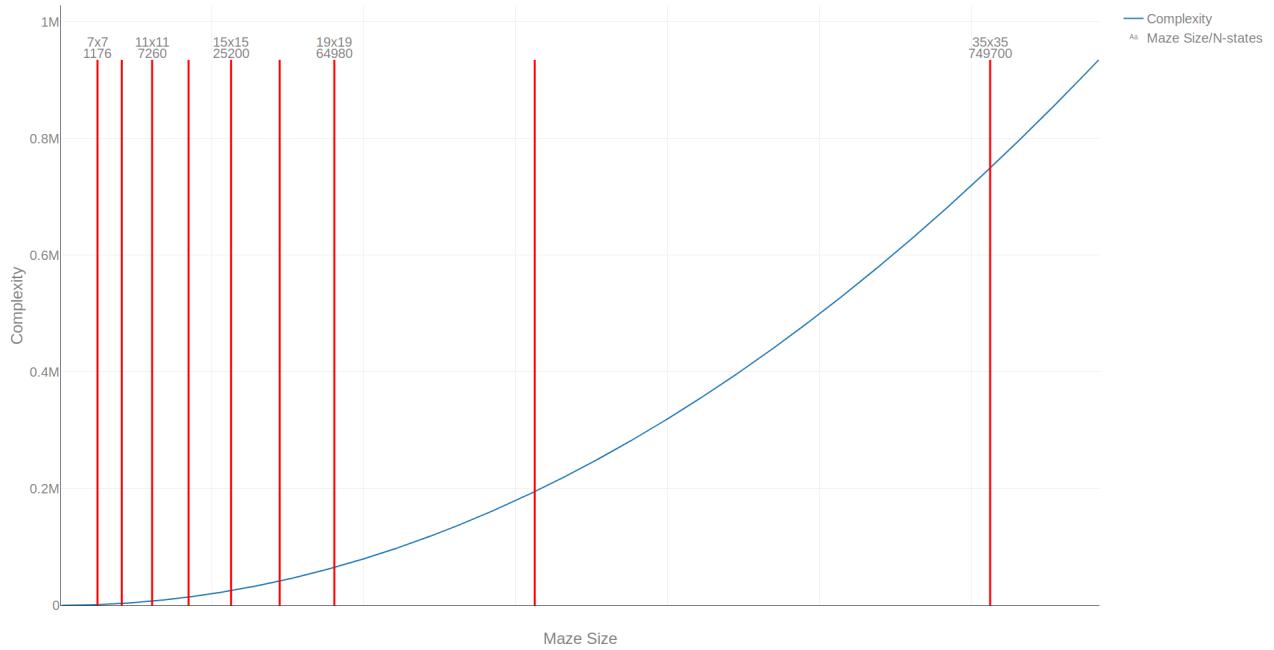


Figure 4.11: Deep Maze: State-space complexity

where the goal is to reach the terminal state in fewest possible time steps. It's a flexible environment that enables research in a single environment setting, as well as multiple scenarios played in sequence.

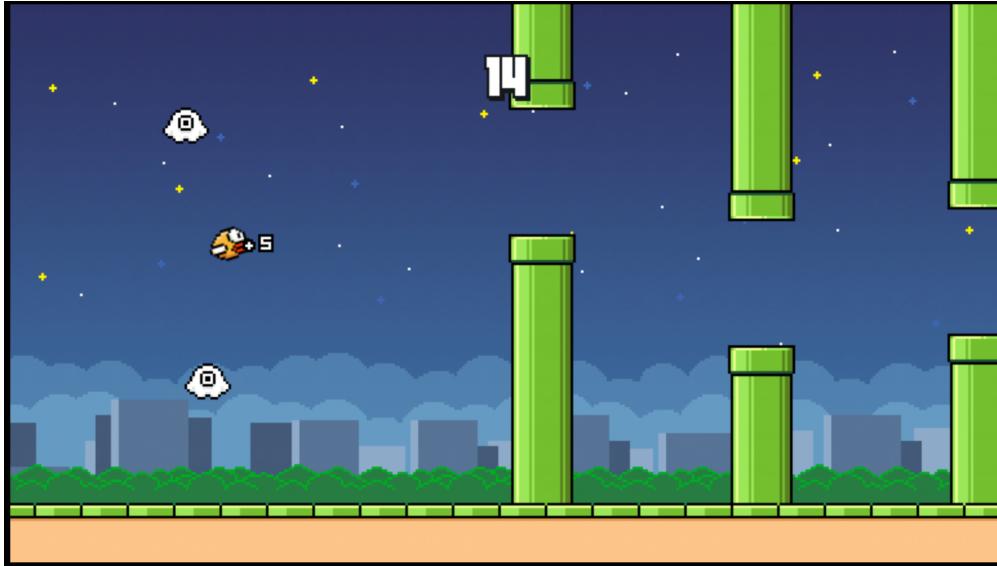


Figure 4.12: Flappy Bird: Graphical User Interface

4.5 Flappy Bird

Flappy Bird is a popular mobile phone game developed by Dong Nguyen in May 2013. The game objective is to control a bird by "flapping" its wings to pass pipes, see Figure 4.12. The player is awarded one point for each pipe passed.

Flappy Bird is widely used in RL research and was first introduced in *Deep Reinforcement Learning for Flappy Bird* [6]. This report shows superhuman agent performance in the game using regular DQN methods¹.

OpenAI's gym platform implements Flappy Bird through PyGame Learning Environment² (PLE). It supports both visual and non-visual state representation. The visual representation is an RGB image while the non-visual information includes vectorized data of the birds position, velocity, upcoming pipe distance, and position.

Figure 4.12 illustrates the visual state representation of Flappy Bird. It is represented by an RGB Image with the dimension of 512×288 . It is recommended that raw images are preprocessed to gray-scale and downsampled to 80×80 . Flappy Bird is an excellent environment for RL and provides adequate validation of new game environments introduced in this thesis.

¹Source code: <https://github.com/yenchenlin/DeepLearningFlappyBird>

²Available at: <https://github.com/ntasfi/PyGame-Learning-Environment>

Chapter 5

Proposed Solutions

Three key solutions are presented in this thesis. First is an architecture that provides a generic communication interface between the environments and the DRL agents. Second is to apply Capsule Layers to DQN, enabling the research into CapsNet based RL algorithms. The third is a novel technique for generating artificial training data for DQN models. These components propose a DRL ecosystem that is suited for research purposes, see Figure 5.1.

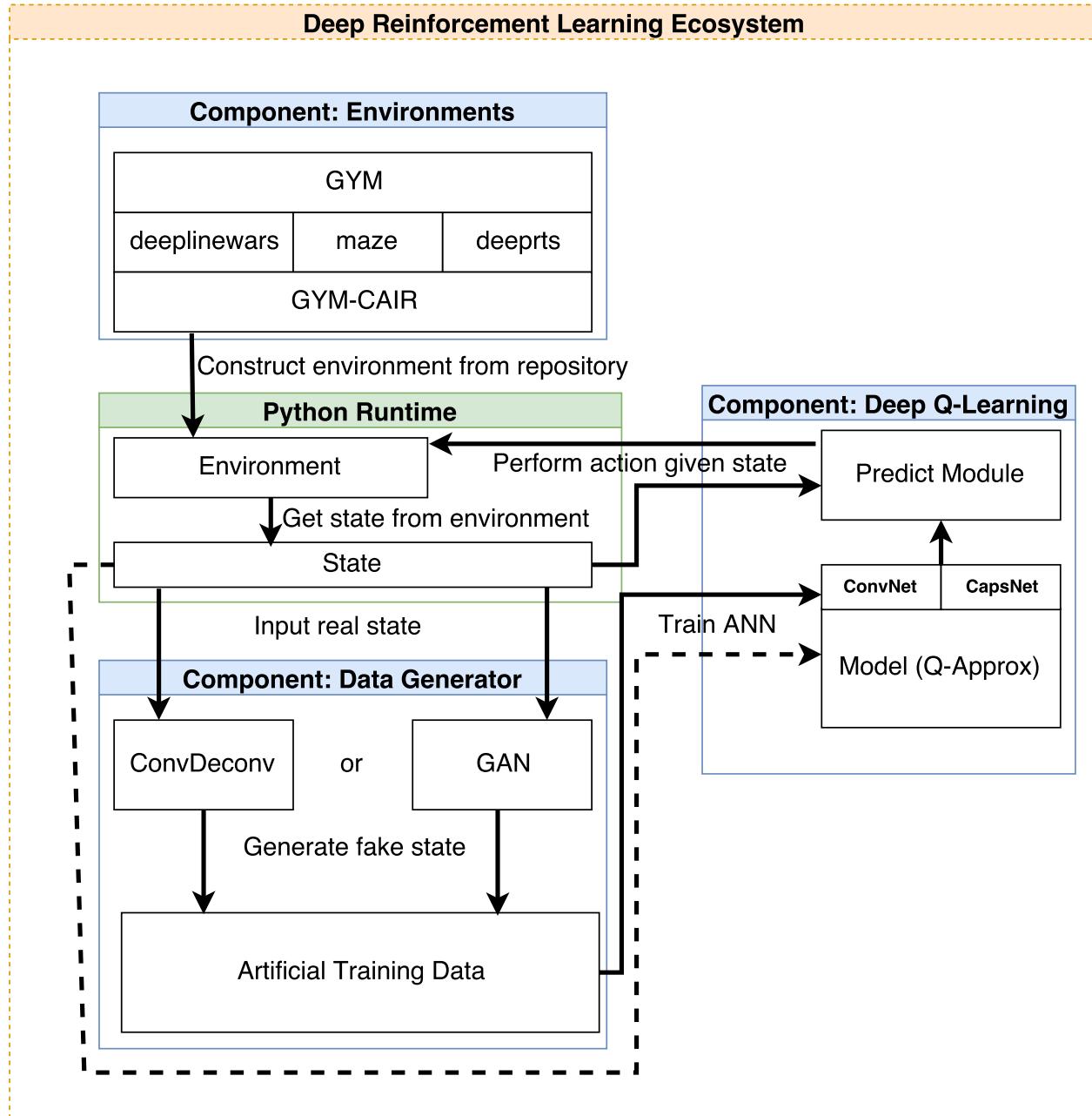


Figure 5.1: Proposed Deep Reinforcement Learning ecosystem

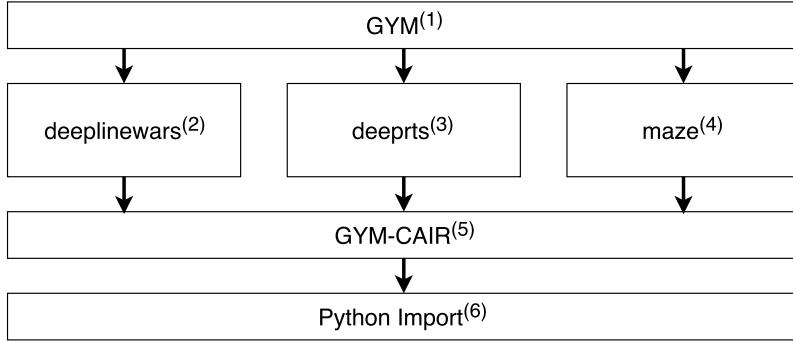


Figure 5.2: Architecture: gym-cair

5.1 Environments

OpenAI GYM is an open-source learning platform, exposing several game environments to the AI research community. There are many existing games available, but these are too simple because they have too easy game objectives. A game environment is *registered* to the GYM platform by defining a *scenario*. This scenario predefines the environment settings that determines the complexity. This type of registration is beneficial because it enables to construct multiple scenarios per game environment. An example of this would be the Maze environment, which contains scenarios for *deterministic* and *stochastic* gameplay for the different maze sizes.

Figure 5.2 illustrates how the environment ecosystem is designed using OpenAI GYM. Environments are registered to the $\text{GYM}_{(1)}$ platform.

Deep Line Wars₍₂₎, Deep RTS₍₃₎ and Maze₍₄₎ are then added to a common repository, called *gym-cair*₍₅₎. This repository links together all environments, which can be imported via Python₍₆₎.

Algorithm 1 Generic GYM RL routine

```

1: statex = gym.reset
2: terminal = False
3: while not terminal do
4:   env.render
5:   a = env.action_space.sample
6:   statex+1, rx+1, terminal, info = env.step(a)
7:   statex = statex+1
8: end while
  
```

The benefit of using GYM is that all environments are constrained to a generic RL interface, seen in Algorithm 1. The environment is initially reset by running *gym.reset* function (Line 1). It is assumed that the environment does not start in a terminal state (Line 2). While the environment is not in a terminal state, the agent can perform actions (Line 5 and 6). This procedure is repeated until the environment reaches the terminal state.

By using this setup, it is far more trivial to perform experiments in the proposed environments. It also enables better comparison, because GYM ensures that the environment configuration is not

altered while conducting the experiments.

Layer Name	Output	Params	Output	Params
Input	$28 \times 28 \times 1$	0	$84 \times 84 \times 1$	0
Conv Layer	$20 \times 20 \times 256$	20 992	$76 \times 76 \times 256$	20 992
Primary Caps	$6 \times 6 \times 256$	5 308 672	$34 \times 34 \times 256$	5 308 672
Capsule Layer	16×16	2 359 296	16×16	75 759 616
Output	16	0	16	0
Parameters	7 688 960		81 089 280	

Table 5.1: Capsule Networks: Dimension Comparison

5.2 Capsule Networks

Capsule Networks recently illustrated that a shallow ANN could successfully classify the MNIST dataset of digits, with state-of-the-art results, using considerably fewer parameters than in regular ConvNets. The idea behind CapsNet is to interpret the input by identifying *parts of the whole*, namely the objects of the input. [45] The objects are identified using Capsules that have the responsibility of finding specific objects in the whole. A capsule becomes active when the object it searches for exist.

It becomes significantly harder to use CapsNet in RL. The objective of RL is to identify actions that are sensible to do in any given state. This means that actions become *parts*, and the *whole* becomes the state. Instead of classifying objects, the capsules now estimate a vector of the likelihood that an action is sensible to do in the current state.

Several issues need to be solved for CapsNet to work properly in the environments outlined in Chapter 4. The first problem is the input size. The MNIST dataset of digits contains images of $28 \times 28 \times 1$ pixels, in contrast, game environments usually range between $64 \times 64 \times 1$ and $128 \times 128 \times 3$ pixels.

Table 5.1 illustrates the consequence of increasing the input size beyond the specified $28 \times 28 \times 1$. By increasing the input size by a magnitude of 3 (84×84), the model gains over $10\times$ parameters. Figure 5.3 illustrates how parameters increase exponentially with the input size. In attempts to solve the scalability issue, several Convolutional Layers can be put in front of the CapsNet. This enables the algorithm to extract feature maps from the original input, but it is crucial to not utilize any form of pooling prior the Capsule Layer. The whole reason to use Capsules is that it solves several problems with invariance that max-pooling possess.

Figure 5.4 illustrates how a standard CapsNet is structured, using a single Convolutional Layer. When a neural network is used, a question is defined to instruct the neural network to predict an answer. For a simple image classification task, the question is: *what do you see in the image*. The neural network then tries to answer, by using its current knowledge: *I see a bird*. The answer is then revealed to the neural network, which allows it to tune its response if it answered incorrectly. The same analogy can be used in an RL problem.

The hope is that despite having several scalability issues, it is possible to accurately encode states into the correct capsules for each possible action in the environment. There are several methods to

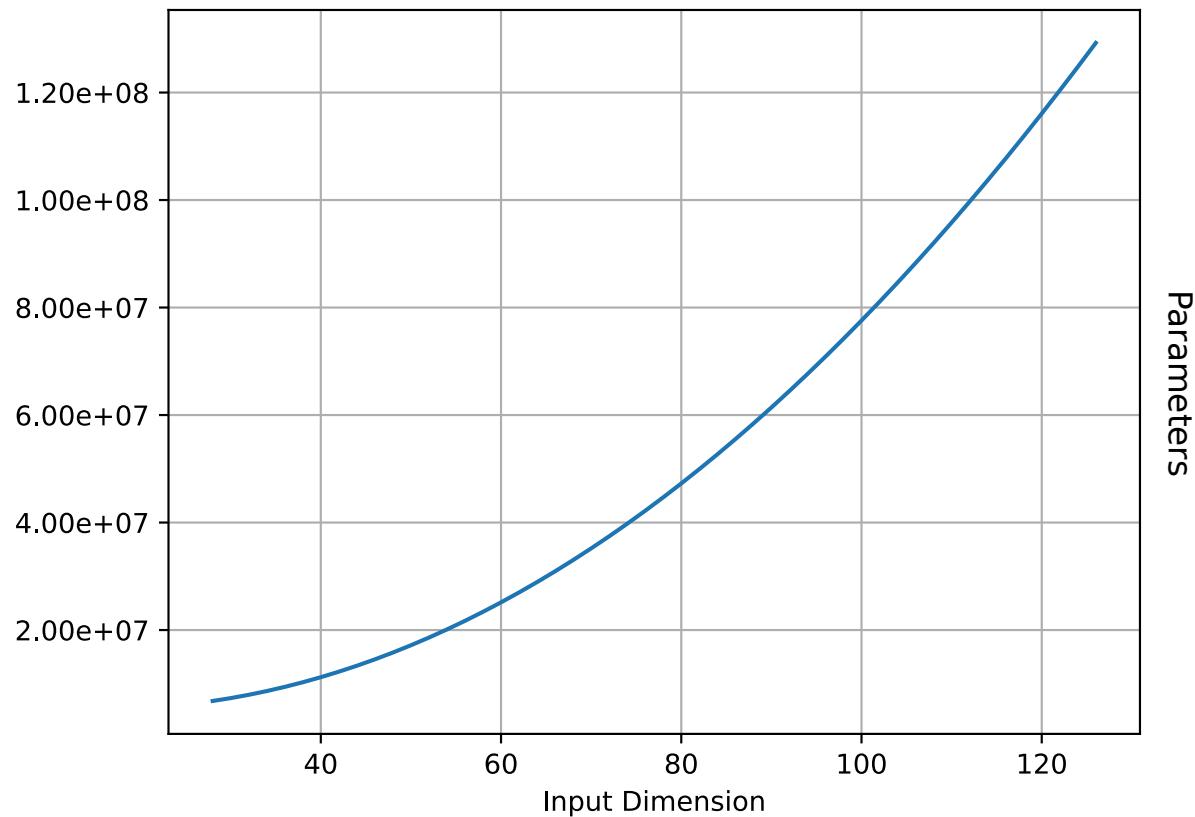


Figure 5.3: Capsule Networks: Parameter count for different input sizes

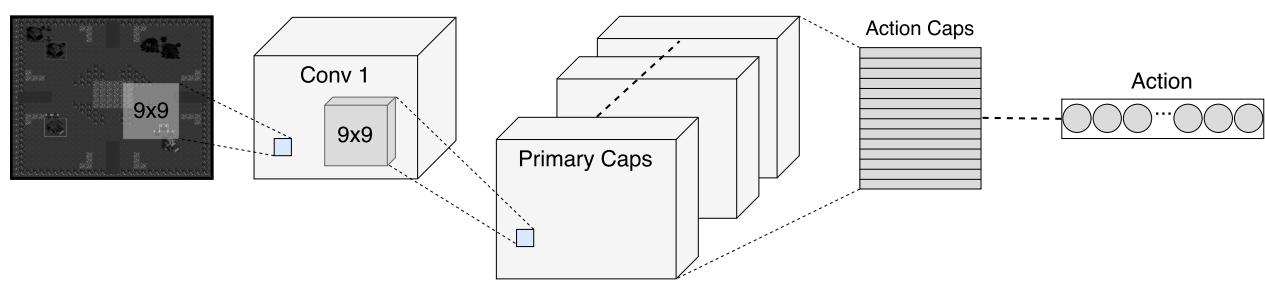


Figure 5.4: Capsule Networks: Architecture

improve the training, but for this thesis, only primitive Q-Learning strategies will be used.

Model	Paper	Year
1 Vanilla DQN	Mnih et al. [36, 37]	2013/2015
2 Deep Recurrent Q-Network	Hausknecht et al. [16]	2015
3 Double DQN	Hasselt et al. [7]	2015
4 Dueling DQN	Wang et al. [43]	2015
5 Continuous DQN	Gu et al. [14]	2016
6 Deep Capsule Q-Network		
7 Recurrent Capsule Q-Network		

Table 5.2: Deep Q-Learning architectures in testbed

Deep Q-Learning Models <i>(It is assumed that all models have a preceding input layer)</i>					
	Model	Layer 1	Layer 2	Layer 3	Layer 4
1	DQN	Conv	Conv	Conv	Dense
		Relu	Relu	Relu	Output
2	DRQN	Conv	Conv	Conv	LSTM
		Relu	Relu	Relu	
3	DDQN	Conv	Conv	Conv	Dense
		Relu	Relu	Relu	Output
4	DuDQN				Relu
					Linear
5	CDQN				Dense
					Output
6	DCQN	Conv	Conv	Conv	Capsule
		Relu	Relu	Relu	OutCaps
7	RCQN ²	Conv	Conv	Conv	Capsule
		Relu	Relu	Relu	OutCaps

Table 5.3: Deep Q-Learning architectures

5.3 Deep Q-Learning¹

There are many different Deep Q-Learning algorithms available consisting of different hyper-parameters, network depth, experience replay strategies and learning rates. The primary problem of DQN is learning stability, and this is shown with the countless configurations found in the literature [7, 14, 16, 36, 37, 43]. Refer to Section 2.5.2 for how the algorithm performs learning of the Q function.

Models 1-4 (Figure 5.2) are the most commonly used DQN architectures found in literature. Model 5 shows great potential in continuous environments, comparable to environments from Chapter 4. Models 6 and 7 are two novel approaches using Capsule Layers in conjunction with Convolution layers [45, 64].

¹General knowledge of ANN, DQN, and CapsNet from Chapter 2 is required.

²Time Distributed / Recurrent

Deep Q-Learning Hyperparameters		
Parameter	Value Range	Default
Learning Rate	0.0-1.0	1e-04
Discount Factor	0.0-1.0	0.99
Loss Function	[Huber, MSE]	Huber
Optimizer	[SGD, Adam, RMSProp]	Adam
Batch Size	$1 \rightarrow \infty$	32
Memory Size	$1 \rightarrow \infty$	1 000 000
ϵ_{min}	$0.0 \rightarrow 1.0$	0.10
ϵ_{max}	$0.0 \rightarrow 1.0$ and $> \epsilon_{min}$	1.0
ϵ_{start}	$\epsilon_{start} \in \{\epsilon_{min}, \epsilon_{max}\}$	1.0

Table 5.4: Deep Q-Learning hyper-parameters

Models 1-7 are implemented in the Keras/Tensorflow framework according to the definitions found in the illustrated papers. Table 5.3 shows the architecture of the DQN models found in Table 5.2. Filter and stride count is intentionally left out because these are considered as hyper-parameters. Hyperparameters are manually tuned by trial and error. Table 5.4 outlines the parameters that are tuned individually for each of the architectures.

5.4 Artificial Data Generator

The Artificial Data Generator component from Figure 5.1 is an attempt to shorten the exploration phase in RL. By generating artificial training data, the hope is that DQN models can learn features that were never experienced within the environment. The proposed algorithm could be able to predict these future states, s_{i+1} given s_i conditioned on action a in the generator function $s_{i+1} = G(s_i|a)$ [35]. The initial plan was to utilize adversarial generative networks but was not able to generate conditioned states successfully. Instead, an architecture called Conditional Convolution Deconvolution Network was developed that use SDG to update parameters (Section 5.4).

Conditional Convolution Deconvolution Network (CCDN) is an architecture that tries to predict the consequence of a condition applied to an image. A state is conditioned on a action to predict future game states.

Figure 5.5 illustrates the general idea of the model. The model is designed using two input streams, image and condition steam. The image stream is a series of convolutional layers following a fully-connected layer. The conditional stream contains fully-connected layers where the last layer matches the number neurons in the last layer of the image steam. These streams are then multiplied following a fully-connected layer that encodes the conditioned states. The conditioned state is then reconstructed using deconvolutions. The output layer is the final reconstructed image of the predicted next state given condition.

The process of training this model is supervised as it consumes data from the experience replay buffer gathered by RL agents. The model is trained by fetching a memory from the experience replay memory (s_i, a, s_{i+1}) where s_i is the current state, a is the action, and s_{i+1} is the transition $T(s_i, a)$. CCDN generates an artificial state \hat{s}_{i+1} by using the generator model $G(s_i|a, \theta)$. The parameters θ is optimized using SDG, and the loss is calculated using MSE.

$$MSE = \frac{1}{n} \sum_{i=1}^n (S_{i+1} - \hat{S}_{i+1})^2 \quad (5.1)$$

Equation 5.1 (Equation 2.2; the predicted value y is now denoted S) is simple in that the loss is decreased when the value of the predicted state \hat{S}_{i+1} gets closer to S_{i+1} .

Table 5.5 illustrates how states are generated using CCDN. It is assumed that it is possible to transition between states given an action, to create training data. When sufficient training data is collected, the recorded state data is used to estimate future states. In this example, there is a 2×2 grid where the agent is a red square with the actions, *up*, *down*, *left*, and *right*. This yields a theoretical maximum state-space of 4 states with 256 possible transitions (4 actions per cell = 4^4 possible state and action combinations). When a portion of the state-space is explored trough random-play the CCDN algorithm can train by comparing the predictions against real data. The goal is for the model to converge towards learning the transition function of the environment, to continue generating future states without any interaction with the environment. It is likely that the model are able to converge towards the optimal solution for more than a single time-step in the future.

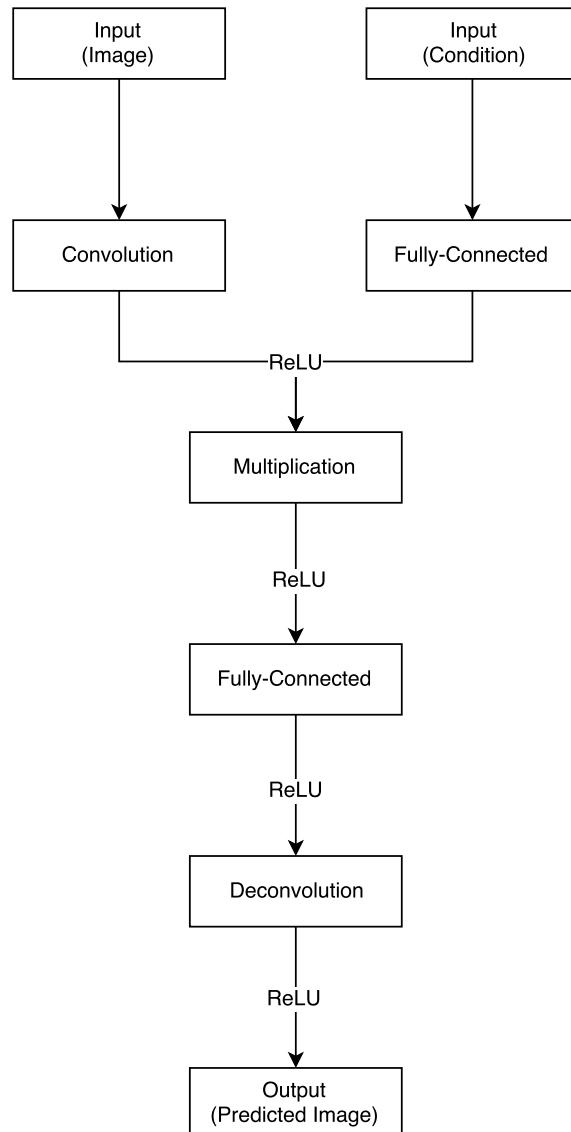


Figure 5.5: Architecture: CCDN

Real States	s_0	$T(s_0, A_{right})$	s_1	$T(s_1, A_{down})$	s_2
<hr/>					
Generated States	N/A	$G(s_0, A_{right})$	\hat{s}_1	$G(\hat{s}_1, A_{down})$	\hat{s}_2

Table 5.5: Proposed prediction cycle for CCDN

Part III

Experiments and Results

Chapter 6

Conditional Convolution Deconvolution Network

This chapter presents Conditional Convolution Deconvolution Network (CCDN). The purpose of the CCDN algorithm is to generate artificial training data for Deep Reinforcement Learning algorithms. The data is generated from the game environments introduced in Chapter 4. The goal is to generate high-quality training data that can be used to train algorithms without self-play. The algorithm is used on the following game environments:

- FlashRL: Multitask (Section 4.1),
- Deep Line Wars (Section 4.2),
- Deep Maze (Section 4.4), and
- Flappy Bird (Section 4.5).

Deep RTS (Section 4.3) is excluded from these tests because it does not support image state-representation¹

A dataset consisting of 100 000 unique state transitions is collected for all environments using random-play strategies. A training set is created, consisting of 60 000 transitions (60%), and the remaining 40% as a test. The training for CCDN took approximately 160 hours per game environment when using hardware listed in Appendix A.

6.1 Introduction

CCDN predicts the future states by conditioning current state on a action. Figure 5.5 illustrates the architecture used in these experiments. To calculate the loss, MSE (Equation 5.1) was used during

¹Deep RTS image state-representation is planned for future work

training. The model is tested using 32, 64, 128, 256, and 512 neurons in the fully-connected layer. Depending on the neuron count, the model has approximately 13 000 000 to 67 000 000 parameters in total.

It is beneficial to have a significant amount of parameters because it allows the model to encode more data. The drawback of this is that the model uses more memory, and takes longer to train. The aim is to train the model for 10 000 epoch at a maximum of 168 hours. For this reason, the algorithm used 32 neurons in the hidden layers which gave reasonably good results for some environments.

Conditioned actions are not shown in the generated images from these experiments. This is because the precision is still too coarse, and the generated future states are yet too far from the ground truth. These results are impressive for some environments, and there is a possibility that the generated samples can be used in conjunction with real samples to train DQN models.

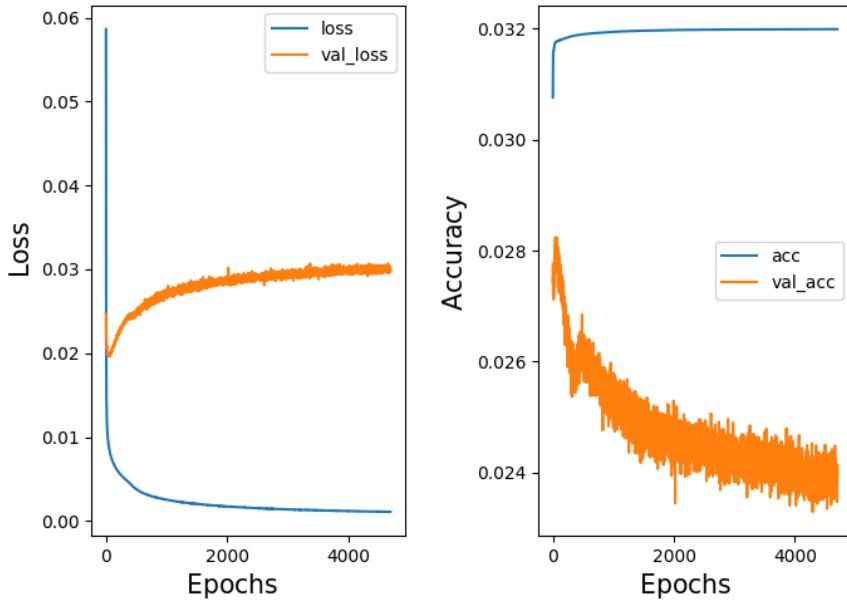


Figure 6.1: CCDN: Deep Line Wars: Training Performance

6.2 Deep Line Wars

Deep Line Wars show excellent results when generating data using CCDN to generate future states $\hat{s} = G(s|a)$. Table 6.1 illustrates the transition from real states (Left side) to generated future states by training CCDN using SDG.

CCDN was not able to converge, but it is possible that this is due to our low neuron count of 32 in the fully-connected layer. Figure 6.1 shows that the loss inclined gradually while the accuracy declined. Loss and accuracy do not reflect the generated images seen in Table 6.1. The observed transitions at Day 5-6.5 illustrate realistic transition behavior between states. Observations indicate that CCDN learns input features like:

- Background intensity (Represents health points)
- Possible mouse position (White square)
- Possible unit positions
- Building positions

The model is still not able to correctly predict the movement of units. This could potentially be solved by stacking several state transitions before predicting future states [6]. This could be done using ConvNets or the use of recurrence in neural networks (RNN).

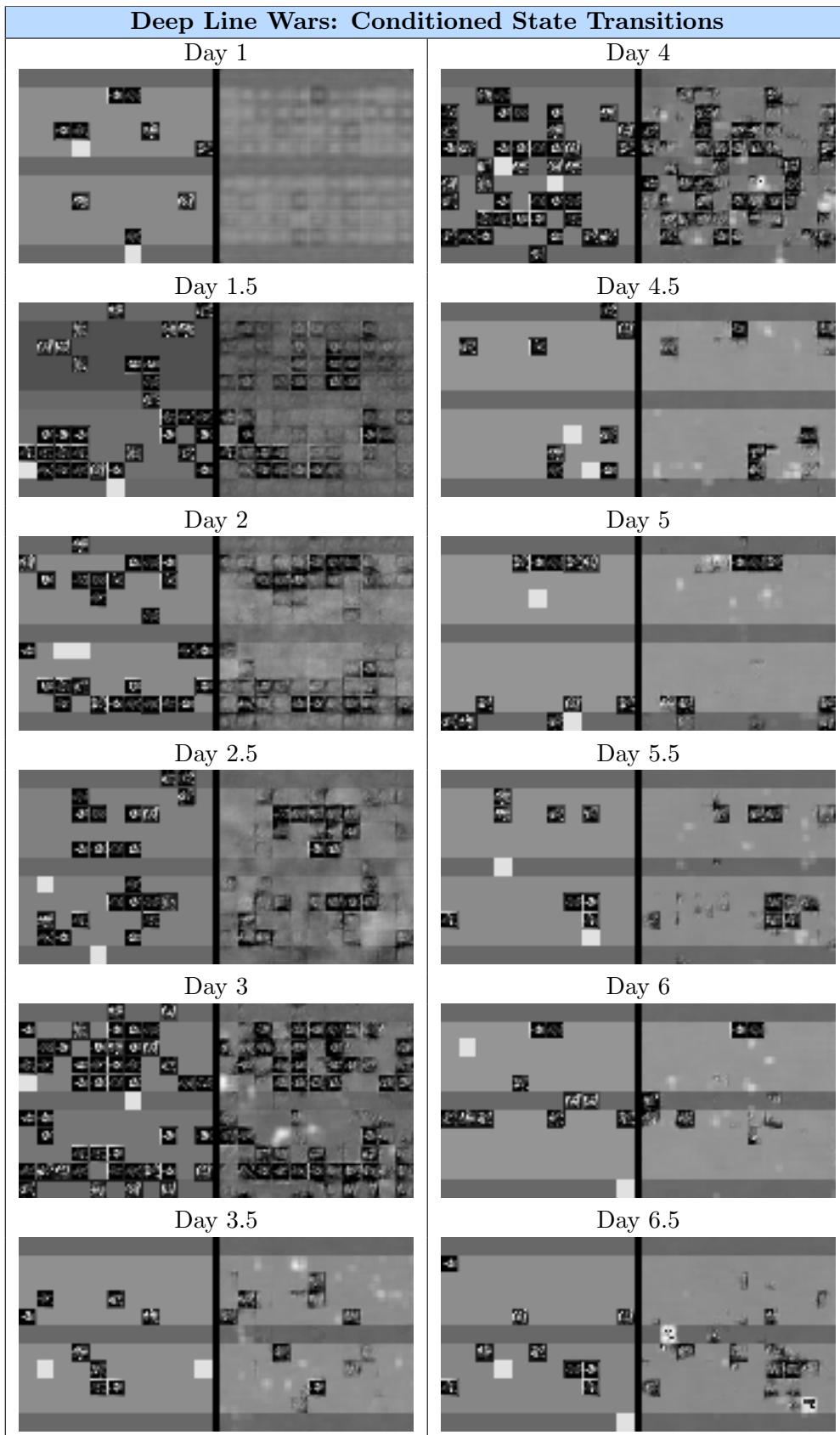


Table 6.1: CCDN2 Deep Line Wars

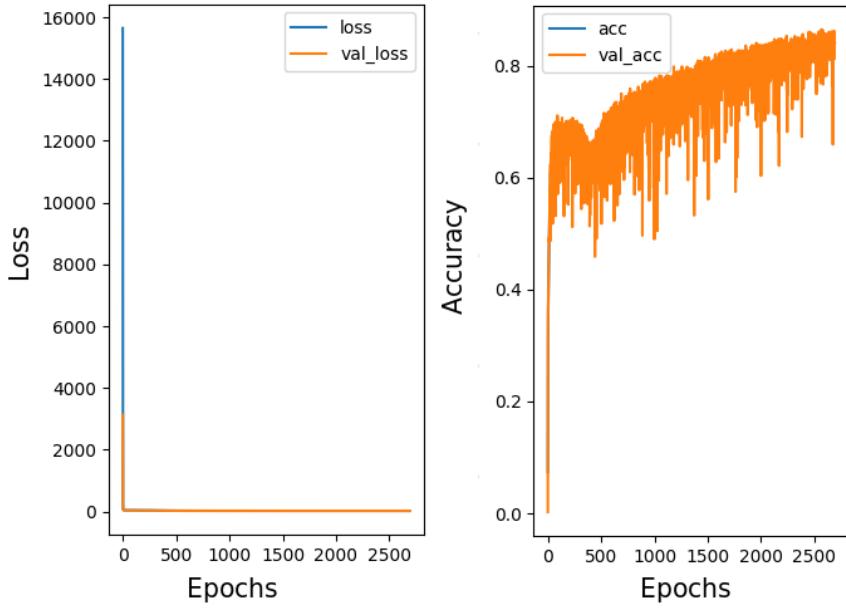


Figure 6.2: CCDN: Deep Maze: Training Performance

6.3 Deep Maze

Deep Maze should be considered as one of the more straightforward environments to generate high-quality training data because it has the simplest state-space. From Table 6.2 it is clear that CCDN recognized features like the maze structure early in the training process. Figure 6.2 illustrates that CCDN converged quickly, having a loss near 0 at the 5th epoch of training. High accuracy was reported during training when using MSE as the loss function. By inspecting the produced images manually, it was clear that CCDN did not learn how to predict the position of the player inside the maze. Hallways inside the maze did not show any sensible information about the actual location of the player. Instead, the maze hallways were generated as random noise. There are however some parts of the maze that presents less noise, indicating that the player did not visit these locations as frequently.

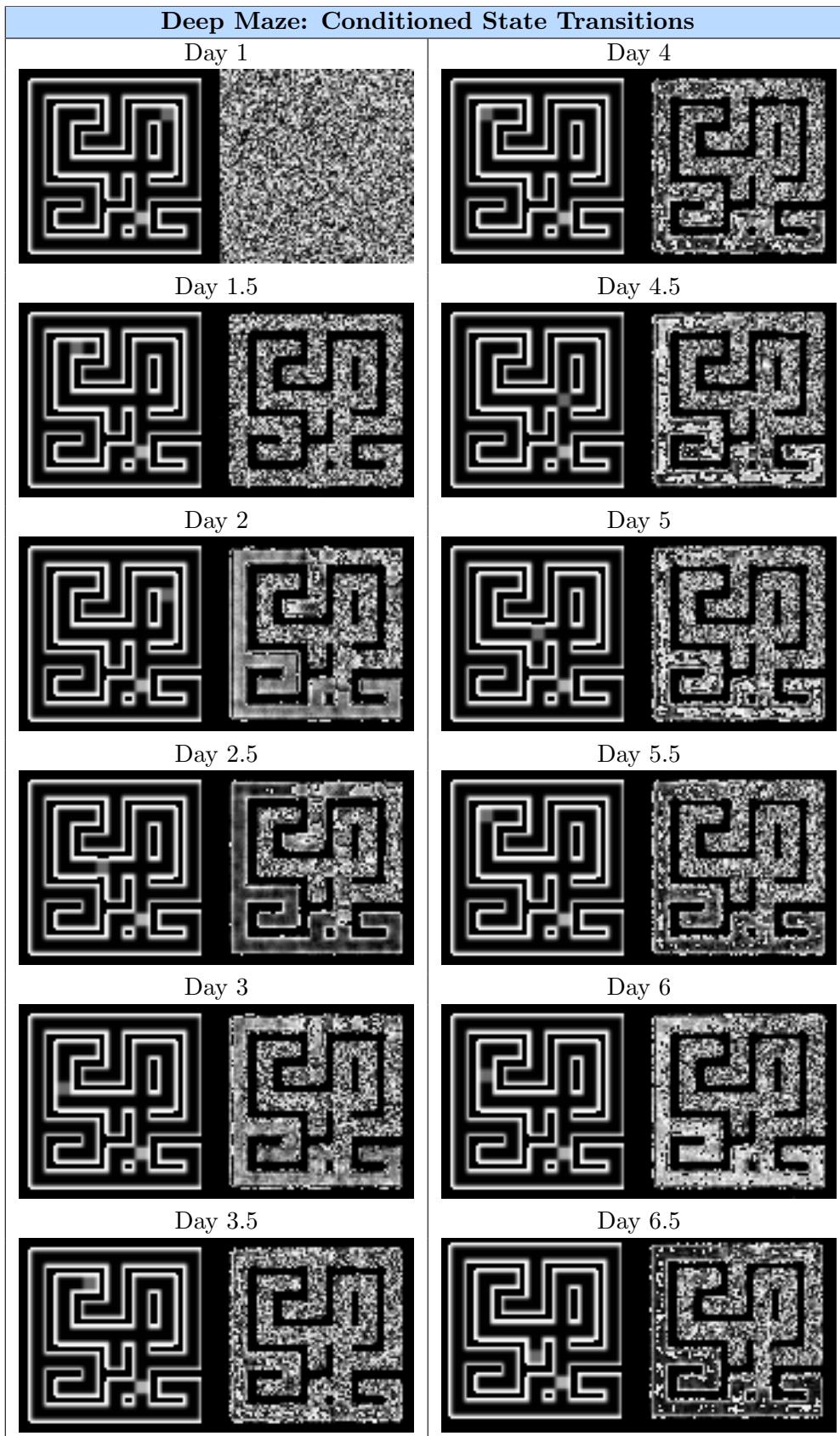


Table 6.2: CCDN: Deep Maze

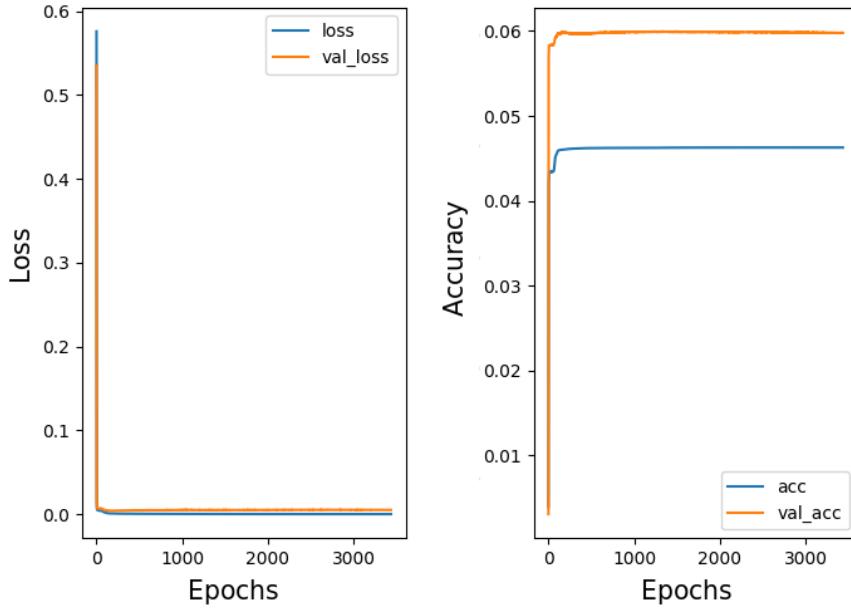


Figure 6.3: CCDN: FlashRL: Training Performance

6.4 FlashRL: Multitask

CCDN produced high-quality state transitions when applying it to Flash RL: Multitask. Since Multitask is an environment consisting of several different scenes (Menu, Stage 1, Stage 2), it was expected that it would fail to generate sensible output. Table 6.3 illustrates that CCDN was able to extract features from all states and map it to correct action. Transitions from Day 2.5 and Day 3.5 illustrates a slight change in the paddle tilt and the position of the ball. This shows that the algorithm can to some extend understand game mechanics. In addition to this, CCDN can draw the menu including a slight change in the mouse position. The results show that CCDN can learn to extract:

- The current scene layout
- Primitive physics

Figure 6.3 illustrates that CCDN did not reach more than 5% accuracy at training time even though the loss was close to zero. It is not clear what is causing the training instability because measuring loss of the images manually using MSE gave far better accuracy for most training samples. The results indicate that CCDN did indeed learn to extract features from the Multitask environment.

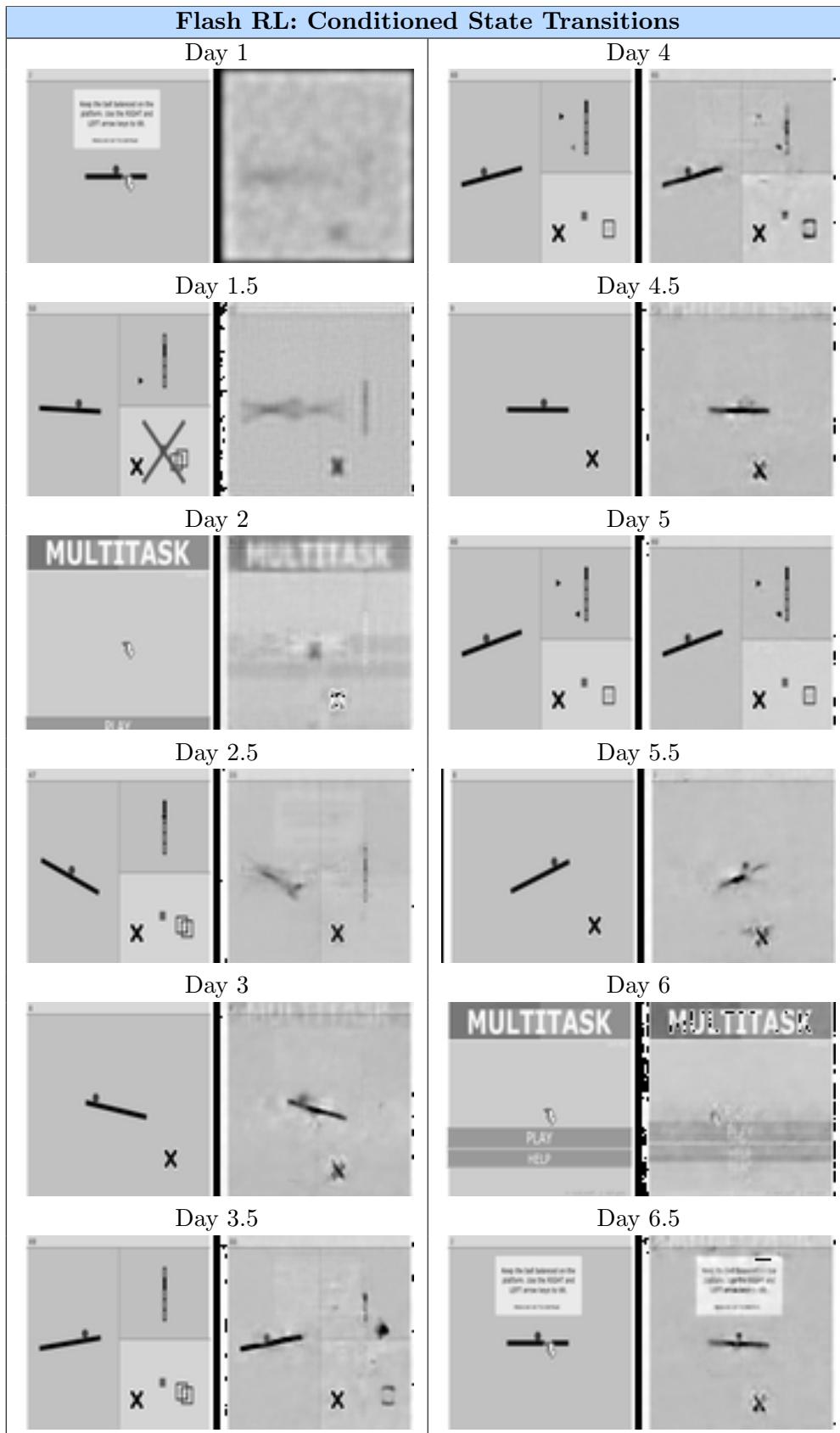


Table 6.3: CCDN:6 FlashRL: Multitask

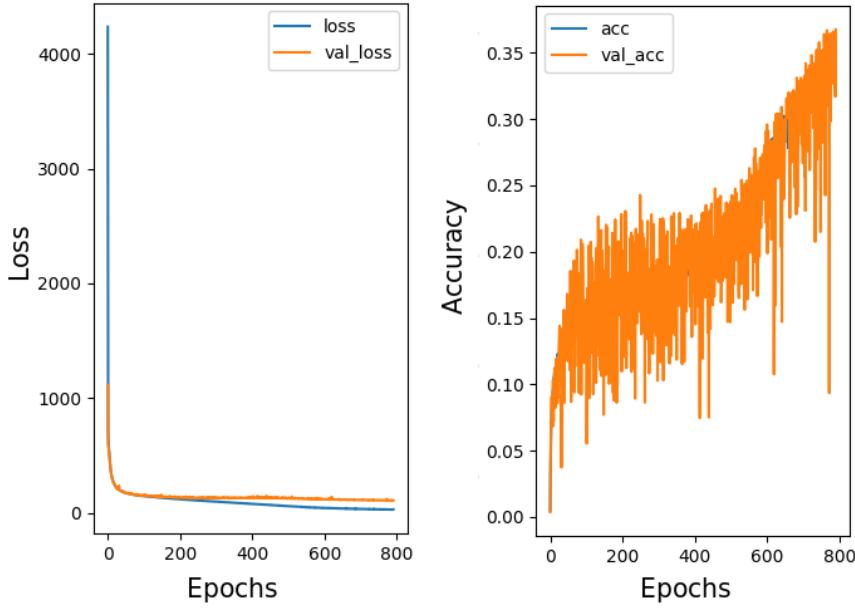


Figure 6.4: CCDN: Flappy Bird: Training Performance

6.5 Flappy Bird

Table 6.4 illustrates the generated transitions for the third party game Flappy Bird. Figure 6.4 show that CCDN has a gradual decrease in the loss while the accuracy increases to approximately 35%. Flappy Bird has the highest accuracy for the tested game environments, but observations shows that CCDN is only able to generate noise.

The reason is that Flappy Bird has a scrolling background, meaning that CCDN must encode a lot more data than in the other environments. Because of this, CCDN could not determine how to generate future state representations for this game.

It is expected that this problem could be mitigated by performing data preprocessing. Literature indicates that RL algorithms often strip away the background to simplify the game-state [13]. Also, it is likely that CCDN could successfully encode Flappy Bird with additional parameters, but this would increase the training time to several weeks.

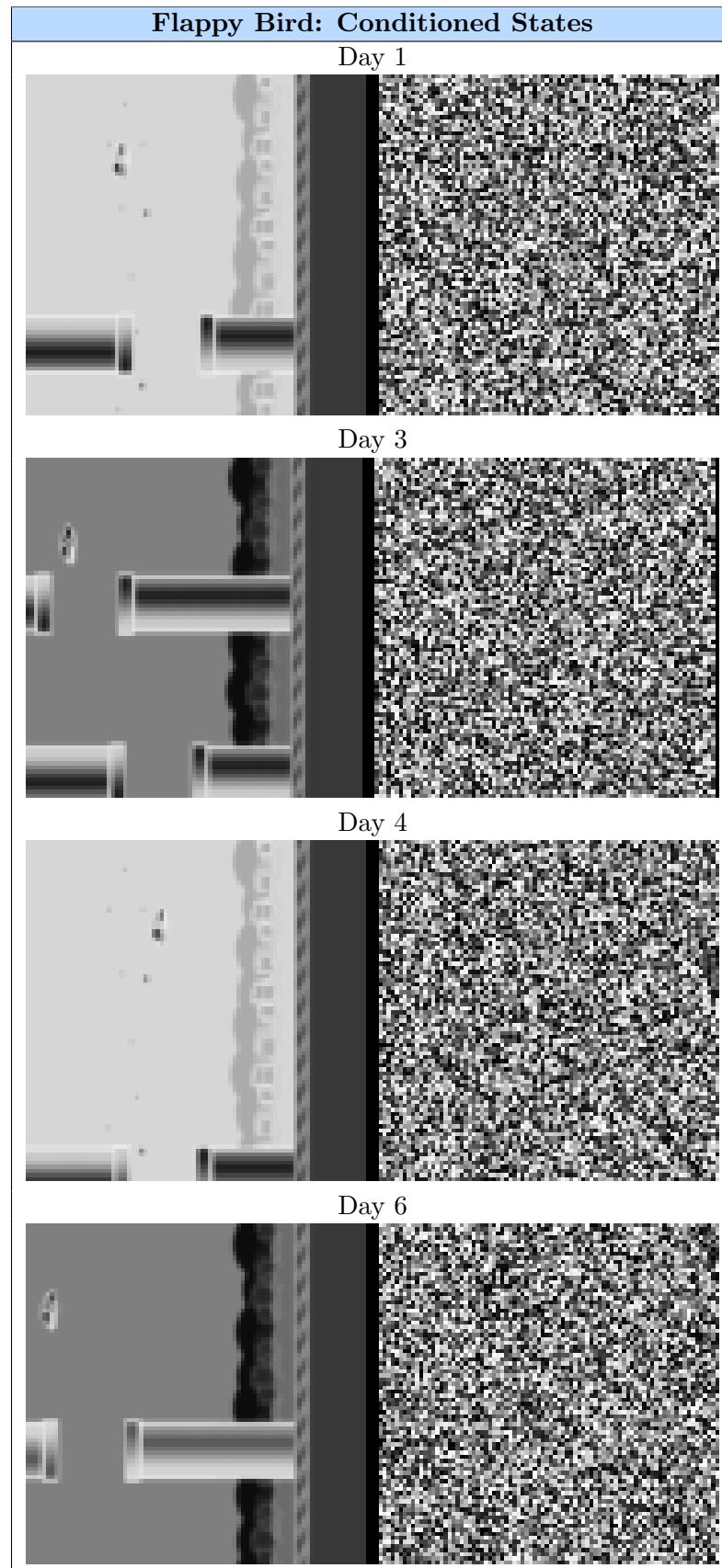


Table 6.4: CCDN: Flappy Bird

6.6 Summary

CCDN is a novel algorithm suited for generating artificial training data for RL algorithms and shows great potential for some environments. The results indicate that CCDN has issues in game environments with a sparse state-space representation. Flappy Bird illustrates the problem well because CCDN generates noise instead of future states for action and state pairs. One method to combat this problem may be to increase the neuron count for the fully-connected layer in the CCDN model.

ANN based algorithms frequently suffer from training instability. The results show that the CCDN algorithm was not able to accurately determine the loss using regular MSE. This could potentially be the cause of the training instability because the optimizer would not be able to determine how well it is doing when updating network parameters. It is likely that replacing the MSE loss function could improve the generated images drastically.

The results presented in this Chapter shows excellent potential in using CCDN for generation of artificial training data for game environments. It shows excellent performance in Deep Line Wars and Flash RL: Multitask and could potentially reduce the required amount of exploration in RL algorithms

Chapter 7

Deep Q-Learning

This chapter presents experimental results of the research done using Deep Q-Learning with CapsNet and ConvNet based models. The goal is to use CapsNet in Deep Q-Learning to solve the environments from Chapter 4.

RL algorithms are known to be computationally intensive and are thus difficult to train for environments with large state-spaces [62]. Models are trained using hardware specified in Appendix A. Chapter 5 proposed 7 DQN architectures that could potentially control an agent well within the environments. Model 1 and 6 from Table 5.2 was selected as the primary research area to limit the scope of this thesis ¹. To increase training stability for all environments, hyper-parameters from Table 5.4 is tuned further per environment. The datasets are populated with 20% artificial training data, generated from CCDN. Table 7.1 illustrates updated hyper-parameters that performed best when experimenting with CapsNet and ConvNet based models. The DQN models use SDG to optimize its parameters. Initial training data is sampled using random-play strategies, gradually moving into exploitation using ϵ -greedy.

Experiments conducted in this thesis are available at <http://github.com/UIA-CAIR>.

¹Training time for 7 models in 5 environments: $7 \times 5 \times 7 = 245$ days (Approx 7 days per experiment)

Environment	α	γ	ϵ -decay	Batch Size	Dataset-Size
Deep Line Wars	3e-5	0.98	0.005	16	1M
Deep Maze	3e-5	0.98	0.005	16	1M
FlashRL:Multitask	1e-4	0.98	0.005	16	1M
Deep RTS	3e-5	0.98	0.005	16	1M
Flappy Bird	2e-4	0.98	0.005	16	1M

Table 7.1: DQN: Hyper-parameters

7.1 Experiments

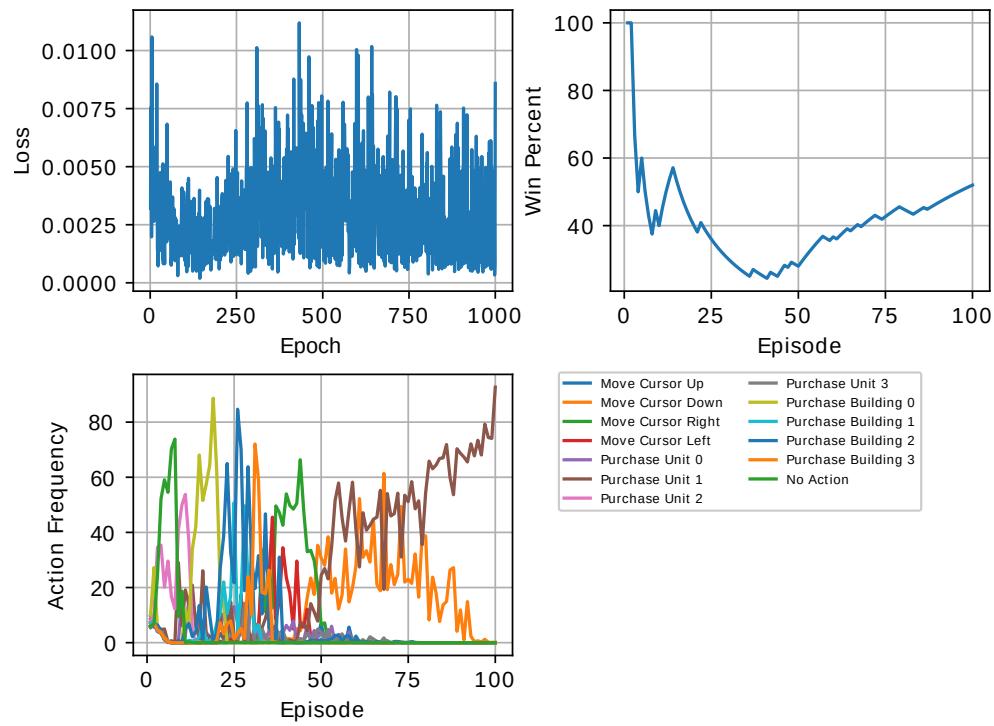


Figure 7.1: DQN-CapsNet: Deep Line Wars

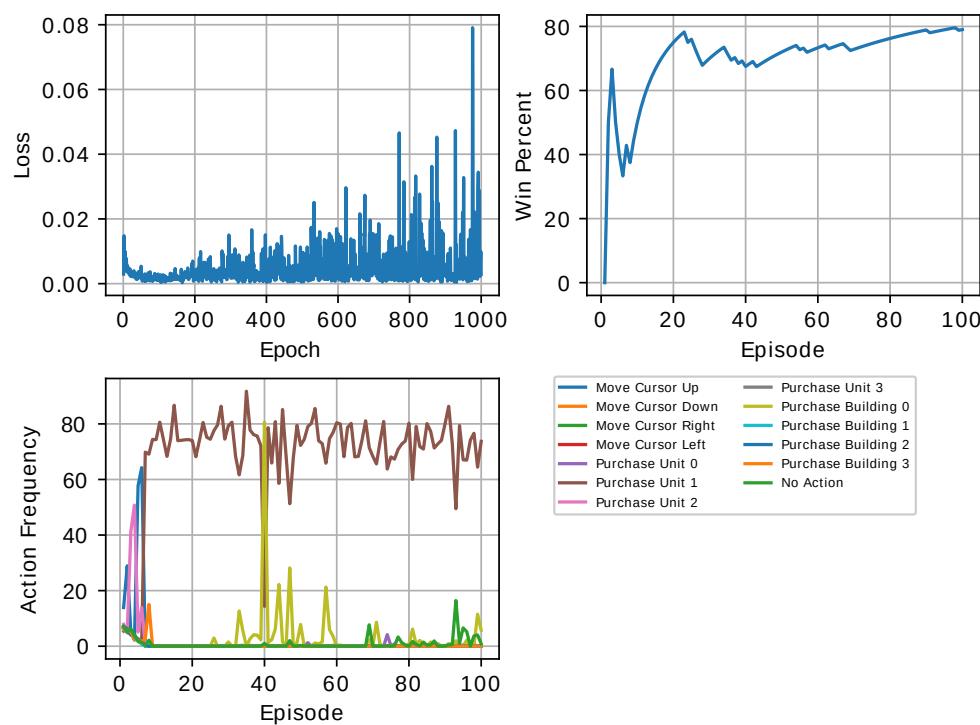


Figure 7.2: DQN-ConvNet: Deep Line Wars

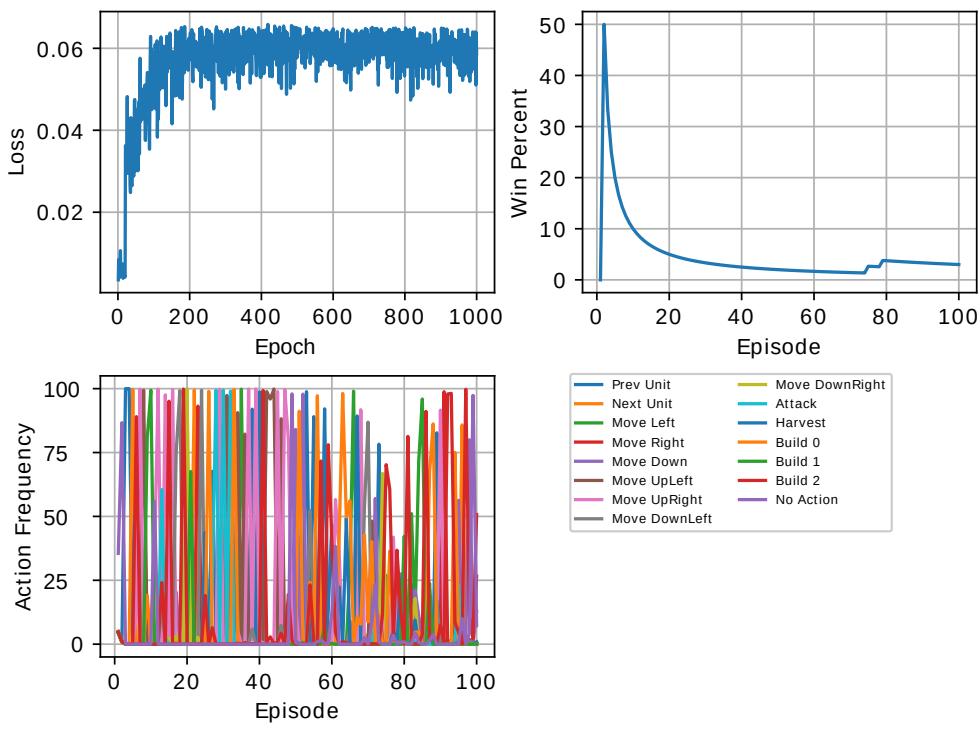


Figure 7.3: DQN-CapsNet: Deep RTS

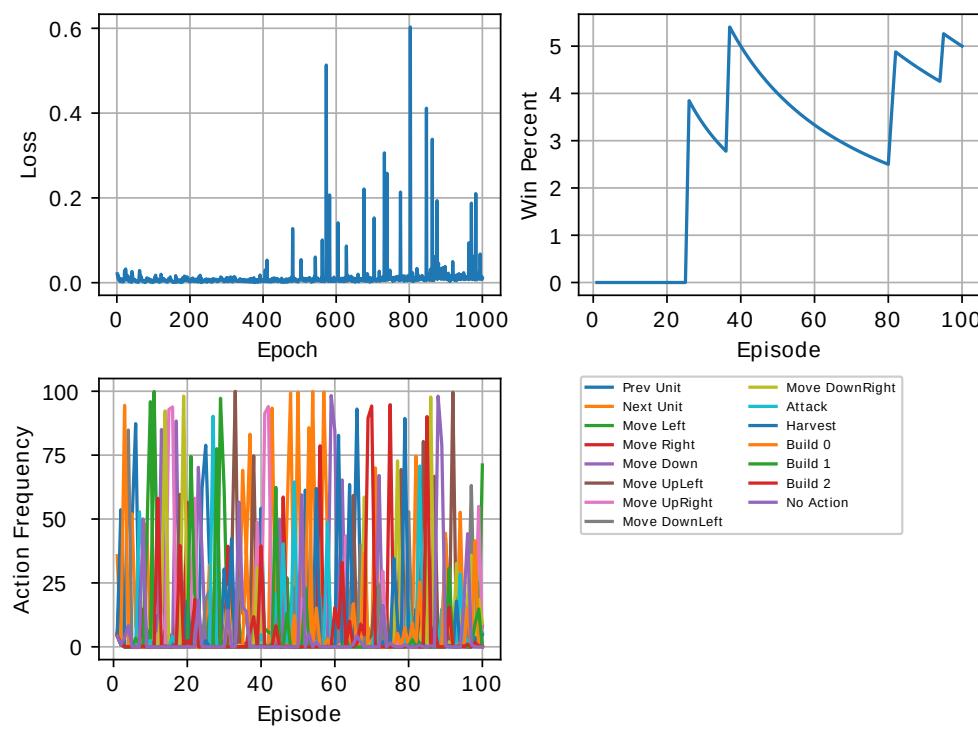


Figure 7.4: DQN-ConvNet: Deep RTS

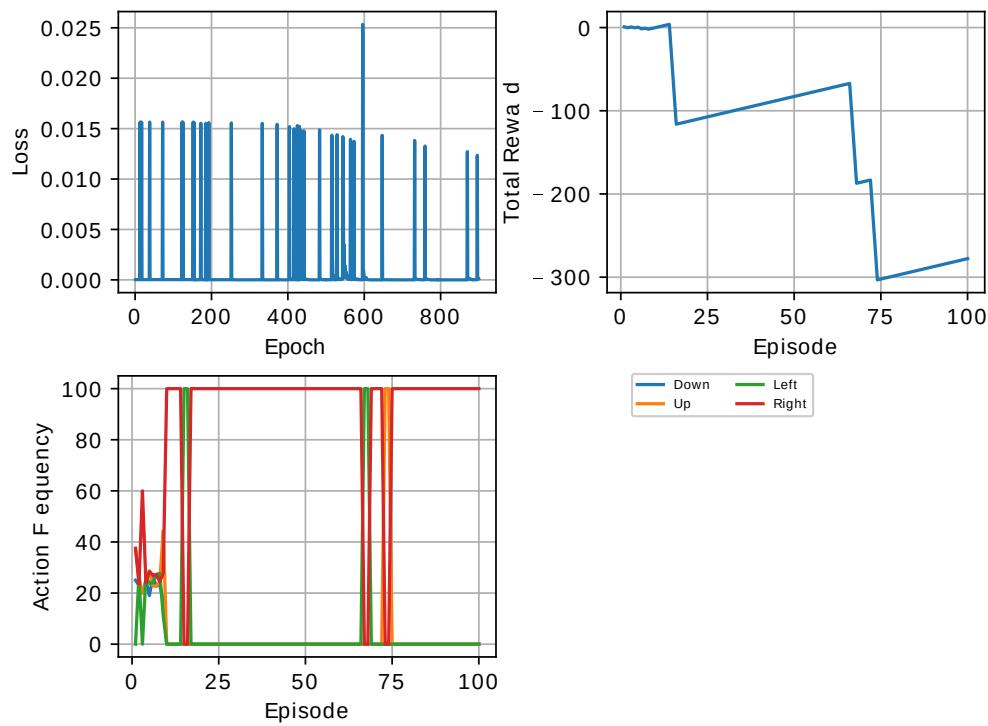


Figure 7.5: DQN-CapsNet: Deep Maze

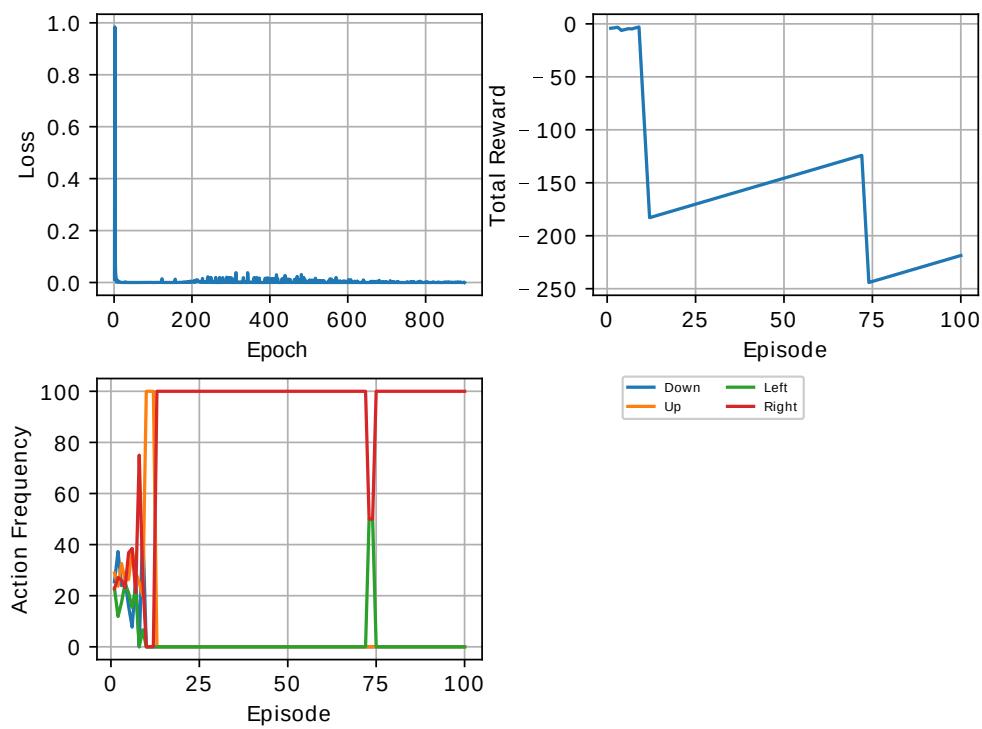


Figure 7.6: DQN-ConvNet: Deep Maze

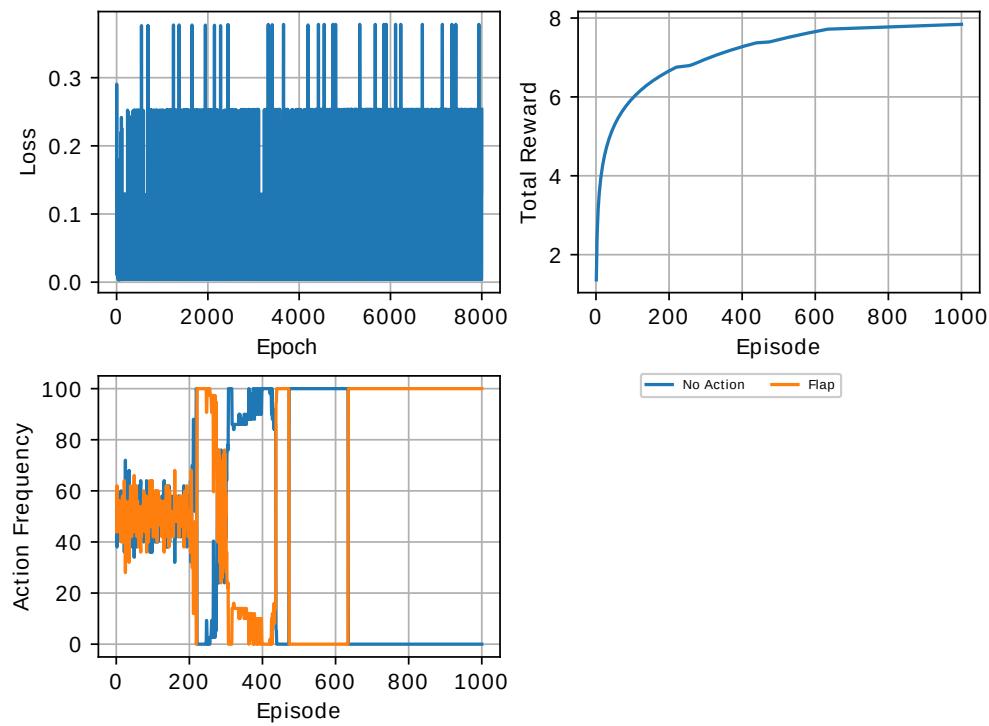


Figure 7.7: DQN-CapsNet: Flappy Bird

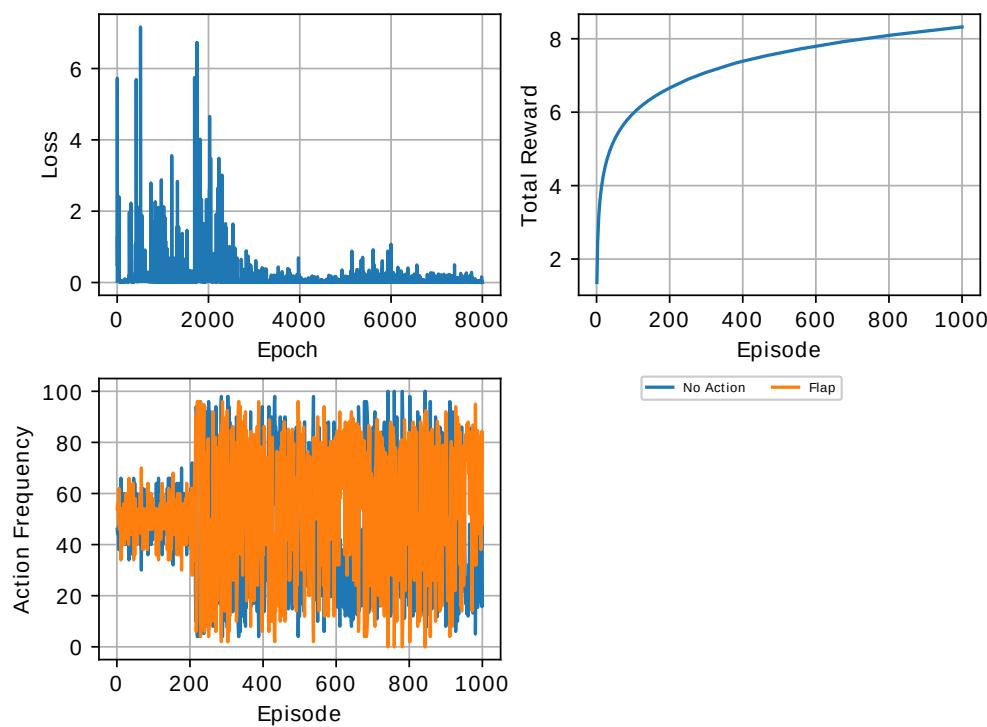


Figure 7.8: DQN-ConvNet: Flappy Bird

7.2 Deep Line Wars

For Deep Line Wars, both agents illustrated relatively strong capabilities when it comes to exploiting game mechanics and finding the opponents weakness. The opponent is a random-play agent, that builds an uneven defense, sending units without any economic considerations. Figure 7.1 and Figure 7.2 show that both agents find the opponents weakness to be defense.



Figure 7.9: DQN-CapsNet: Agent building defensive due to low health in Deep Line Wars

Results shows that the game mechanics are not balanced, making the *Purchase Unit 1* the obvious choice for offensive actions. This unit is strong enough to survive most defenses and does the most damage to the opponents health pool. The ConvNet agent performs better in a period of 100 episodes, and both agents can master the random-play opponent.

7.3 Deep RTS

Deep RTS shows exciting results, where DQN-CapsNet starts at a low loss with a high total reward, slowly diverging in reward and loss. The results show that DQN-CapsNet and DQN-ConvNet perform comparably. It is not clear why DQN-CapsNet diverged, but the high action-space is a likely candidate. It is difficult to see any sense in the determination of action-state mapping, but

some observations indicated that the agent favor gathering instead of military actions.

7.4 Deep Maze

The goal of Deep Maze is to find the shortest path from start to goal in a 25×25 labyrinth. Figure 7.5 and Figure 7.6 shows that DQN-CapsNet had issues with the training stability. The algorithm is tested with several different hyper-parameter configurations, but there was no solution to remedy this. DQN-ConvNet did not indicate any issues during the training. Both agents had issues finding the shortest path, looking at the total reward, both agents had a negative score. For each move done after reaching the optimal move count, a negative reward is given the agent. Observations show that both agents have similar performance in this experiment.

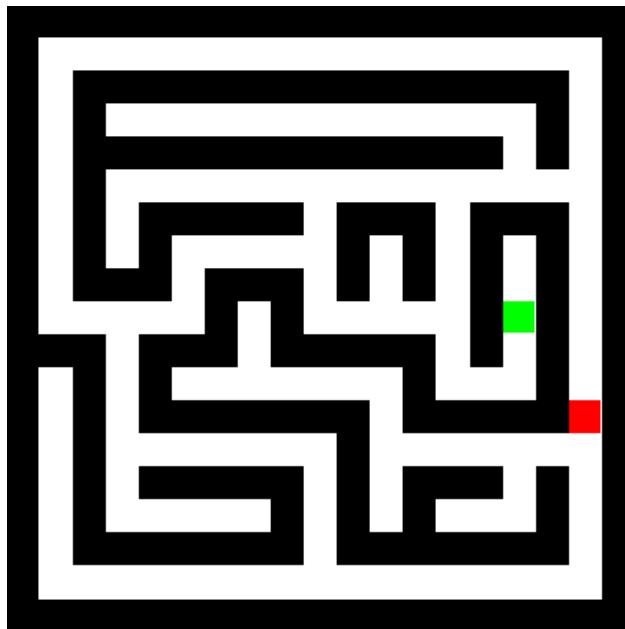


Figure 7.10: DQN-CapsNet: Agent attempting to find the shortest path in a 25×25 *Deep Maze*

Figure 7.10 illustrates an in-game image of the 25×25 map used in this experiment. The green square is the start area, while the red is the goal. The optimal path for this experiment is a series of 21 actions.

7.5 FlashRL: Multitask

For FlashRL: Multitask, the DQN-CapsNet was not able to compete with DQN-ConvNet. It was not able to learn how to control the first paddle. The results are for this reason not included for

this environment. Refer to **Publication B** for results using DQN-ConvNet.

7.6 Flappy Bird

Flappy Bird is a difficult environment for an agent to master because the state-space is large due to the scrolling background. In literature, the training time for this environment is between one and four days. For this experiment, the agent trained for seven days, in the hope that both agents would converge. Figure 7.7 and Figure 7.8 shows that both agents performed well, where DQN-ConvNet scored 0.4 points higher. For each pipe, the bird passes, 0.1 points are awarded to the total reward.

7.7 Summary

Looking at the results, it is clear that DQN-CapsNet overestimates actions for almost all environments. Instead of having a sensible distribution of actions, it often chooses to favor a particular move after a short period of training.

Recent state-of-the-art suggests that self-play using dueling methods may increase stability and performance in the long-term [43], but this was not possible due to GPU memory limitations. It is clear that DQN-CapsNet can work for other tasks than image recognition, but there are still many challenges to solve before it can perform comparably to DQN-ConvNet. A significant issue is that Capsules do not scale well with several outputs (actions), resulting in a model that quickly becomes too large for the GPU memory to handle. The upcoming paragraphs summarize the findings of the experiments conducted using DQN-CapsNet and DQN-ConvNet.

Training Loss

An interesting observation during the training was that none of the models had a gradual decline in the loss during training. This may be because the state-space was quite large for all environments in the test-bed. Some investigation revealed that environments with sparse input had a more significant loss increase. By comparing Figure 7.1 and Figure 7.3, it is clear that Deep RTS has far more loss compared to Deep Line Wars when predicting the best Q-Value for an action. Since CapsNet primarily detects objects, it is likely that the sudden jumps in loss (Figure 7.5) can be explained by several capsules changing its prediction vector at the same time. A possible improvement would be to decay the learning rate throughout the training period. It is likely that the training loss issues can be managed for models with several new hyper-parameter configurations.

Action Frequency

Results shows that CapsNet tends to overestimate actions drastically in environments with few actions (Deep Maze and Flappy Bird). It is possible that this is because a Capsule looks for *parts in the whole*. Since CapsNet is positional invariant, one explanation may be that the model classifies states by looking for the existence of an object, instead of the likelihood of the best action. For Flappy Bird, the model determines that the agent should use *Flap* as long as the bird exists in the input. For environments with large action-spaces, observations show a more consistent action frequency.

Environment	Random	DQN-CapsNet	DQN-ConvNet
Deep Line Wars	50	57	78
Deep RTS	1.4	5.0	5.1
Deep Maze	-600	-275	-225
FlashRL: Multitask	14	N/A	300
Flappy Bird	1.4	7.9	8.3

Table 7.2: Comparison of DQN-CapsNet, DQN-ConvNet, and Random accumulative reward (Higher is better)

Agent Performance

Table 7.2 shows that DQN-CapsNet does indeed perform above random-play agents in selected environments, but falls behind compared to DQN-ConvNet. For all environments, a higher score is better. In Deep Line Wars, the reward increases as the agent keep surviving the game or defeat the enemy. The CapsNet agent has approximately 57% win chance while ConvNet wins in 78% of the games against a random-play agent.

In Deep RTS, the accumulated score is measured during the first 600 seconds of the game. This is typically resource harvesting, as the agent was never able to create long-term strategies. In early training, CapsNet accumulated far more resources than ConvNet, but it gradually declined while training. This means that the model diverged from the optimal solution. It is likely that this is because the model starts to overestimate action Q-values. In comparison, results show that both models perform comparably while performing well beyond the capability of random-play agents.

In Deep Maze, none of the agents were able to find the optimal path to the goal. Additional experiments were conducted and showed better results for smaller mazes (9x9 and 11x11). For 25x25 the CapsNet used on average 275 additional actions to reach the goal, while ConvNet performed marginally better using 225 actions.

The CapsNet agent is able to perform well in Flappy Bird. With only 0.4 points less than ConvNet, it is clear that both agents perform at the same level of expertise. It is possible that the CapsNet agent could achieve far better results if a solution is found for the Q-Value overestimation problem.

Chapter 8

Conclusion and Future Work

This thesis conclusively shows that Capsules are viable to use in advanced game environments. It is further shown that capsules do not scale as well as convolutions, implying that capsule networks alone will not be able to play even more advanced games without improved scalability.

This thesis has focused on **Deep Reinforcement Learning using Capsules in Advanced Game Environments**. This work presents several new game environments that are tailored for research into RL algorithms in the RTS genre. This contribution could potentially lead to a ground-breaking performance in advanced game environments that could enable RL agents to perform well in games like Starcraft II. The combination of Capsule Networks and Deep Q-Learning illustrated comparable results to regular ConvNets, in regards to stability, on the new learning platform. As a secondary goal, a generative model was implemented, CCDN, which successfully generates future state representations in the majority of the test environments.

Since Capsule Networks are a novel research area that is in its early infant stage, more research is required to determine its capabilities in RL for advanced game environments. This chapter presents the thesis conclusion and future work for the continuation of a PhD thesis in DRL.

8.1 Conclusion

Hypothesis 1: *Generative modeling using deep learning is capable of generating artificial training data for games with sufficient quality.*

Our work shows that it is indeed possible to generate artificial training data using deep learning. Our work shows that it is of sufficient quality to perform off-line training of deep neural networks.

Hypothesis 2: *CapsNet can be used in Deep Q-Learning with comparable performance to ConvNet based models.*

The research shows that CapsNet can be directly adapted to work with Deep Q-Learning, but the stability is inferior to regular ConvNet. Some experiments show comparable results to ConvNets, but it is not clear how CapsNets do reasoning in an RL environment.

Goal 1: *Investigate the state-of-the-art research in the field of Deep Learning, and learn how Capsule Networks function internally.*

A thorough survey of the state-of-the-art in deep learning was outlined in Chapter 3. Much of the performed work was inspired by previous research, which enabled several exciting discoveries in RL. Results show that it is possible to combine CapsNet with other algorithms.

Goal 2: *Design and develop game environments that can be used for research into RL agents for the RTS game genre.*

The thesis outline four new game environments that target research into RL agents for RTS games.

Deep RTS is a Warcraft II clone that is suited for an agent of high-quality play. It requires the agent to do actions in a high-dimensional environment that is continuously moving. Since the Deep RTS state is of such high-dimension, it is still not feasible to master this environment.

Deep Line Wars was created to enable research on a simpler scale, this enabled research into some of the RTS aspects, found in Deep RTS.

To simplify it even further, Deep Maze was created to only account for trivial state interpretations. Flash RL was created as a side project, enabling research into a vast library of Flash games.

Together, these game environments create a platform that allows for in-depth research into RL problems in the RTS game genre.

Goal 3: *Research generative modeling and implement an experimental architecture for generating artificial training data for games.*

CCDN is introduced as a novel architecture for generating artificial future states from a game, using present state and action to learn the transition function of an environment. Early experimental results are presented in this work, showing that it has potential to successfully train a neural network based model.

Goal 4: *Research the novel CapsNet architecture for MNIST classification and apply this to RL problems.*

Section 5.2 outlines the research into CapsNet in scenarios that are different from the MNIST experiments conducted by Sabour et al [45]. The objective of Capsules is redefined so that it could work for RL related problems.

Goal 5: *Combine Deep-Q Learning and CapsNet and perform experiments on environments from Achievement 2.*

In Chapter 7, DQN and CapsNet were successfully combined and illustrated that it has the potential to perform well in several advanced game environments. Although these results only show minor agent intelligence, it is an excellent beginning for further research into this type of deep models.

Goal 6: *Combine the elements of Goal 3 and Goal 5. The goal is to train an RL agent with artificial training data successfully.*

Results shows that training data generated with CCDN can be used in conjunction with real data to train an DQN algorithm successfully.

All of the goals defined in the scope of this thesis were accomplished. Although the results are not astounding for all goals, it enables further research into several new deep learning fields. The work presented in this thesis enables further research into CapsNet based RL in advanced game environments. Because of the new learning platform, researchers can better perform research into RTS games. It is possible that the work from this thesis could be the foundation for novel RL algorithms in the future.

8.2 Future Work

Environments

1. Continue work on Flash RL, enabling it to replace OpenAI Universe Flash.
2. Propose partnership with ELF¹ and implement Deep RTS and Deep Line Wars into ELF.
3. Develop a full-fledged platform that expands beyond gym-cair.
4. Implement Image state-representation for Deep RTS.

Generative Modeling

1. Additional experiments with hyper-parameters with the existing models.
2. Attempt to stabilize training.
3. Investigate if it is possible to use adversarial methods to train the generative model.
4. Identify and solve the issue with the loss function in CCDN.

Deep Capsule Q-Learning

1. Improve stability of current architecture, enabling less data. preprocessing for the algorithm to function.
2. Improve the scalability of Capsules for large action spaces.
3. Do additional experiments with multiple configurations to find the cause of the training instability.
4. More research into combining Capsules with RL algorithms.

Planned Publications²

1. Deep RTS: A Real-time Strategy game for Reinforcement Learning.
2. CCDN: Towards infinite training data using generative models.
3. DCQN: Using Capsules in Deep Q-Learning.

¹ELF Source-code: <https://github.com/facebookresearch/ELF>

²Proposed Publication titles may change in final versions

References

- [1] Per-Arne Andersen, Morten Goodwin, and Ole-Christoffer Granmo. FlashRL: A Reinforcement Learning Platform for Flash Games. *Norsk Informatikkonferanse*, 2017.
- [2] Per Arne Andersen, Morten Goodwin, and Ole Christoffer Granmo. Towards a deep reinforcement learning approach for tower line wars. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10630 LNAI, pages 101–114, 2017.
- [3] Charles Beattie, Joel Z. Leibo, Denis Teplyashin, Tom Ward, Marcus Wainwright, Heinrich Küttler, Andrew Lefrancq, Simon Green, Víctor Valdés, Amir Sadik, Julian Schrittwieser, Keith Anderson, Sarah York, Max Cant, Adam Cain, Adrian Bolton, Stephen Gaffney, Helen King, Demis Hassabis, Shane Legg, and Stig Petersen. DeepMind Lab. dec 2016.
- [4] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *IJCAI International Joint Conference on Artificial Intelligence*, 2015-Janua:4148–4152, 2015.
- [5] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. jun 2016.
- [6] Kevin Chen. Deep Reinforcement Learning for Flappy Bird. page 6, 2015.
- [7] Wenliang Chen, Min Zhang, Yue Zhang, and Xiangyu Duan. Exploiting meta features for dependency parsing and part-of-speech tagging. *Artificial Intelligence*, 230:173–191, sep 2016.
- [8] Gianfranco Ciardo and Andrew S. Miner. Storage alternatives for large structured state spaces. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 1245, pages 44–57, 1997.
- [9] Bo Dai, Sanja Fidler, Raquel Urtasun, and Dahua Lin. Towards Diverse and Natural Image Descriptions via a Conditional GAN. mar 2017.
- [10] Kenji Doya, Kazuyuki Samejima, Ken-ichi Katagiri, and Mitsuo Kawato. Multiple model-based reinforcement learning. *Neural computation*, 14(6):1347–1369, 2002.
- [11] Eyal Even-dar, Shie Mannor, and Yishay Mansour. Action Elimination and Stopping Conditions for Reinforcement Learning. *Icml*, 7:1079–1105, 2003.

- [12] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Liyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, Gang Wang, Jianfei Cai, and Tsuhan Chen. Recent advances in convolutional neural networks. *Pattern Recognition*, dec 2017.
- [13] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates, 2017.
- [14] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous Deep Q-Learning with Model-based Acceleration. mar 2016.
- [15] Abhishek Gupta, Clemens Eppner, Sergey Levine, and Pieter Abbeel. Learning dexterous manipulation for a soft robotic hand from human demonstrations. In *IEEE International Conference on Intelligent Robots and Systems*, volume 2016-Novem, pages 3786–3793, 2016.
- [16] Matthew Hausknecht and Peter Stone. Deep Recurrent Q-Learning for Partially Observable MDPs. jul 2015.
- [17] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7):1527–1554, 2006.
- [18] Harold L Hunt and I I Jon Turney. Cygwin/X Contributor’s Guide. 2004.
- [19] Aaron Courville Ian Goodfellow, Yoshua Bengio. *Deep Learning*, volume 521. MIT Press, 2017.
- [20] Matthew Johnson, Katja Hofmann, Tim Hutton, and David Bignell. The malmo platform for artificial intelligence experimentation. *IJCAI International Joint Conference on Artificial Intelligence*, 2016-Janua:4246–4247, 2016.
- [21] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.
- [22] I Kanter, Y LeCun, and S Solla. Second-order properties of error surfaces: learning time and generalization. *Advances in Neural Information Processing Systems (NIPS 1990)*, 3:918–924, 1991.
- [23] Shohei Kinoshita, Takahiro Ogawa, and Miki Haseyama. LDA-based music recommendation with CF-based similar user selection. *2015 IEEE 4th Global Conference on Consumer Electronics, GCCE 2015*, pages 215–216, jun 2016.
- [24] George Konidaris and Andrew G. Barto. Autonomous shaping. *International Conference on Machine Learning*, pages 489–496, 2006.
- [25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks, 2012.
- [26] Ben J.A. Kröse. *Learning from delayed rewards*. PhD thesis, King’s College, Cambridge, UK, 1995.
- [27] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551, dec 1989.

- [28] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2323, 1998.
- [29] Chen-Yu Lee, Patrick W. Gallagher, and Zhuowen Tu. Generalizing Pooling Functions in Convolutional Neural Networks: Mixed, Gated, and Tree. sep 2015.
- [30] Yuxi Li. Deep Reinforcement Learning: An Overview. *arXiv*, pages 1–30, 2017.
- [31] L J Lin. Reinforcement Learning for Robots Using Neural Networks. *Report, CMU*, pages 1–155, 1993.
- [32] Björn Lindström, Ida Selbing, Tanaz Molapour, and Andreas Olsson. Racial Bias Shapes Social Reinforcement Learning. *Psychological Science*, 25(3):711–719, feb 2014.
- [33] Chunhui Liu, Aayush Bansal, Victor Fragoso, and Deva Ramanan. Do Convolutional Neural Networks act as Compositional Nearest Neighbors? *arXiv*, pages 1–15, 2017.
- [34] Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andrew J. Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, Dharshan Kumaran, and Raia Hadsell. Learning to Navigate in Complex Environments. nov 2016.
- [35] Mehdi Mirza and Simon Osindero. Conditional Generative Adversarial Nets. nov 2014.
- [36] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. dec 2013.
- [37] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, feb 2015.
- [38] Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. DeepStack: Expert-Level Artificial Intelligence in No-Limit Poker. jan 2017.
- [39] Fernando Naclerio, Marco Seijo-Bujia, Eneko Larumbe-Zabala, and Conrad P. Earnest. Carbohydrates alone or mixing with beef or whey protein promote similar training outcomes in resistance training males: A double-blind, randomized controlled clinical trial. *International Journal of Sport Nutrition and Exercise Metabolism*, 27(5):408–420, 2017.
- [40] Bruno A. Olshausen and David J. Field. Sparse coding of sensory inputs. *Current Opinion in Neurobiology*, 14(4):481–487, sep 2004.
- [41] Etienne Perot, Maximilian Jaritz, Marin Toromanoff, and Raoul De Charette. End-to-End Driving in a Realistic Racing Game with Deep Reinforcement Learning. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2017-July:474–475, may 2017.
- [42] Laurent Praly and Yuan Wang. Stabilization in spite of matched unmodeled dynamics and an equivalent definition of input-to-state stability. *Mathematics of Control, Signals, and Systems*, 9(1):1–33, 1996.

- [43] Carlos Ramirez-Perez and Victor Ramos. SDN meets SDR in self-organizing networks: Fitting the pieces of network management. *IEEE Communications Magazine*, 54(1):48–57, nov 2016.
- [44] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, oct 1986.
- [45] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic Routing Between Capsules. *Nips*, oct 2017.
- [46] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved Techniques for Training GANs. jun 2016.
- [47] Wojciech Samek, Alexander Binder, Gregoire Montavon, Sebastian Lapuschkin, and Klaus Robert Muller. Evaluating the Visualization of What a Deep Neural Network Has Learned. *IEEE Transactions on Neural Networks and Learning Systems*, sep 2016.
- [48] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6354 LNCS, pages 92–101, 2010.
- [49] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.
- [50] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. dec 2017.
- [51] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Van Den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.
- [52] Richard S Sutton and Andrew G Barto. Time-Derivative Models of Pavlovian Reinforcement. *Learning and Computational Neuroscience: Foundations of Adaptive Networks*, (Mowrer 1960):497–537, 1990.
- [53] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*, volume 9. MIT Press, 1998.
- [54] W.W. Swart, C.E. Gearing, and T. Var. *A dynamic programming—integer programming algorithm for allocating touristic investments*, volume 27. 1972.
- [55] Techtonik. python-vnc-viewer. <https://github.com/techtonik/python-vnc-viewer>, 2015.

-
- [56] Gerald Tesauro. TD-Gammon, a Self-Teaching Backgammon Program, Achieves Master-Level Play. *Neural Computation*, 6(2):215–219, 1994.
 - [57] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, 1995.
 - [58] Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C. Lawrence Zitnick. ELF: An Extensive, Lightweight and Flexible Research Platform for Real-time Strategy Games. jul 2017.
 - [59] Harm van Seijen, Mehdi Fatemi, Joshua Romoff, Romain Laroche, Tavian Barnes, and Jeffrey Tsang. Hybrid Reward Architecture for Reinforcement Learning. jun 2017.
 - [60] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. StarCraft II: A New Challenge for Reinforcement Learning. aug 2017.
 - [61] Fang Wan and Chaoyang Song. Logical Learning Through a Hybrid Neural Network with Auxiliary Inputs. may 2017.
 - [62] Jiang Wang, Yang Song, Thomas Leung, Chuck Rosenberg, Jingbin Wang, James Philbin, Bo Chen, and Ying Wu. Learning fine-grained image similarity with deep ranking. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1386–1393, apr 2014.
 - [63] Songtao Wu, Shenghua Zhong, and Yan Liu. Deep residual learning for image steganalysis. *Multimedia Tools and Applications*, pages 1–17, dec 2017.
 - [64] Edgar Xi, Selina Bing, and Yang Jin. Capsule Network Performance on Complex Data. dec 2017.
 - [65] Lecun Yann. Efficient backprop. *Neural networks: tricks of the trade*, 53(9):1689–1699, 1998.

Appendices

A Hardware Specification

Operating System	Ubuntu 17.10
Processor	Intel i7-7700K
Memory	64GB DDR4
Graphics	1x NVIDIA GeForce 1080TI

Part IV

Publications

Appendix A

Towards a Deep Reinforcement Learning Approach for Tower Line Wars

Towards a Deep Reinforcement Learning Approach for Tower Line Wars

Per-Arne Andersen⁽¹⁾, Morten Goodwin, and Ole-Christoffer Granmo

University of Agder, Grimstad, Norway
per-arne.andersen@uia.no

Abstract. There have been numerous breakthroughs with reinforcement learning in the recent years, perhaps most notably on Deep Reinforcement Learning successfully playing and winning relatively advanced computer games. There is undoubtedly an anticipation that Deep Reinforcement Learning will play a major role when the first AI masters the complicated game plays needed to beat a professional Real-Time Strategy game player. For this to be possible, there needs to be a game environment that targets and fosters AI research, and specifically Deep Reinforcement Learning. Some game environments already exist, however, these are either overly simplistic such as Atari 2600 or complex such as Starcraft II from Blizzard Entertainment.

We propose a game environment in between Atari 2600 and Starcraft II, particularly targeting Deep Reinforcement Learning algorithm research. The environment is a variant of Tower Line Wars from Warcraft III, Blizzard Entertainment. Further, as a proof of concept that the environment can harbor Deep Reinforcement algorithms, we propose and apply a Deep Q-Reinforcement architecture. The architecture simplifies the state space so that it is applicable to Q-learning, and in turn improves performance compared to current state-of-the-art methods. Our experiments show that the proposed architecture can learn to play the environment well, and score 33% better than standard Deep Q-learning—which in turn proves the usefulness of the game environment.

Keywords: Reinforcement Learning · Q-Learning · Deep Learning · Game environment

1 Introduction

Despite many advances in AI for games, no universal reinforcement learning algorithm can be applied to Real-Time Strategy Games (RTS) without data manipulation or customization. This includes traditional games such as Warcraft III, Starcraft II, and Tower Line Wars. Reinforcement Learning (RL) has been applied to simpler games such as games for the Atari 2600 platform but has to the best of our knowledge not successfully been applied to RTS games. Further, existing game environments that target AI research are either overly simplistic such as Atari 2600 or complex such as Starcraft II.

Reinforcement Learning has had tremendous progress in recent years in learning to control agents from high-dimensional sensory inputs like vision. In simple environments, this has been proven to work well [1], but are still an issue for complex environments with large state and action spaces [2]. In games where the objective is easily observable, there is a short distance between action and reward which fuels the learning. This is because the consequence of any action is quickly observed, and then easily learned. When the objective is more complicated the game objectives still need to be mapped to the reward function, but it becomes far less trivial. For the Atari 2600 game Ms. Pac-Man this was solved through a hybrid reward architecture that transforms the objective to a low-dimensional representation [3]. Similarly, the OpenAI’s bot is able to beat world’s top professionals at 1v1 in DotA 2. It uses reinforcement learning while it plays against itself, learning to predict the opponent moves.

Real-Time Strategy Games, including Warcraft III, is a genre of games much more comparable to the complexity of real-world environments. It has a sparse state space with many different sensory inputs that any game playing algorithm must be able to master in order to perform well within the environment. Due to the complexity and because many action sequences are required to constitute a reward, standard reinforcement learning techniques including Q-learning are not able to master the games successfully.

This paper introduces a two-player version of the popular Tower Line Wars modification from the game Warcraft III. We refer to this variant as Deep Line Wars. Note that Tower Line Wars is not an RTS game, but has many similar elements such as time-delayed objectives, resource management, offensive, and defensive strategy planning. To prove that the environment is working we, inspired by recent advances from van Seijen et al. [3], apply a method of separating the abstract reward function of the environment into smaller rewards. This approach uses a Deep Q-Network using a Convolutional Neural Network to map actions to states and can play the game successfully and perform better than standard Deep Q-learning by 33%.

Rest of the paper is organized as follows: We first investigate recent discoveries in Deep RL in Sect. 2. We then briefly outline how Q-Learning works and how we interpret Bellman’s equation for utilizing Neural Networks as a function approximator in Sect. 3. We present our contribution in Sect. 4 and present a comparison of other game environments that are widely used in reinforcement learning. We introduce a variant of Deep Q-Learning in Sect. 5 and present a comparison to other RL models used in state-of-the-art research. Finally we show results in Sect. 6, define a roadmap of future work in Sect. 7 and conclude our work in Sect. 8.

2 Related Work

There have been several breakthroughs related to reinforcement learning performance in recent years [4]. Q-Learning together with Deep Learning was a game-changing moment, and has had tremendous success in many single agent

environments on the Atari 2600 platform [1]. Deep Q-Learning as proposed by Mnih et al. [1] as shown in Fig. 1 used a neural network as a function approximator and outperformed human expertise in over half of the games [1].

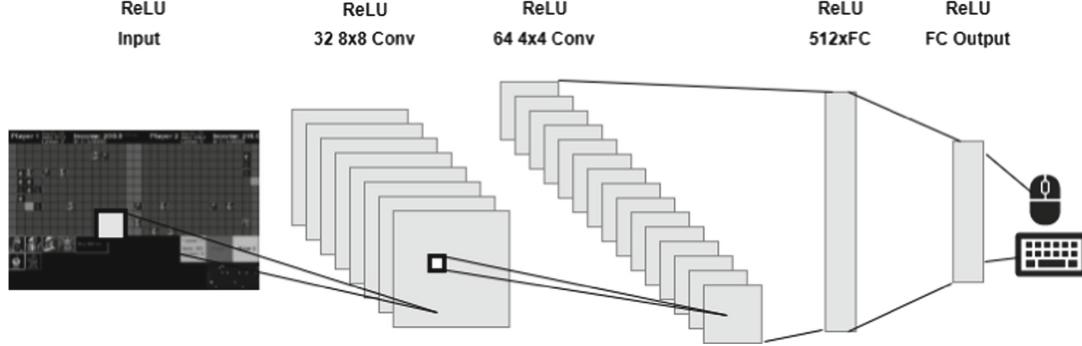


Fig. 1. Deep Q-Learning architecture

Hasselt et al. proposed Double DQN, which reduced the overestimation of action values in the Deep Q-Network [5]. This led to improvements in some of the games on the Atari platform.

Wang et al. then proposed a dueling architecture of DQN which introduced estimation of the value function and advantage function [6]. These two functions were then combined to obtain the Q-Value. Dueling DQN were implemented with the previous work of van Hasselt et al. [6].

Harm van Seijen et al. recently published an algorithm called Hybrid Reward Architecture (HRA) which is a divide and conquer method where several agents estimate a reward and a Q-value for each state [3]. The algorithm performed above human expertise in Ms. Pac-Man, which is considered one of the hardest games in the Atari 2600 collection and is currently state-of-the-art in the reinforcement learning domain [3]. The drawback of this algorithm is that generalization of Minh et al. approach is lost due to a huge number of separate agents that have domain-specific sensory input.

There have been few attempts at using Deep Q-Learning on advanced simulators specifically made for machine-learning. It is probable that this is because there are very few environments created for this purpose.

3 Q-Learning

Reinforcement learning can be considered hybrid between supervised and unsupervised learning. We implement what we call an agent that acts in our environment. This agent is placed in the unknown environment where it tries to maximize the environmental reward [7].

Markov Decision Process (MDP) is a mathematical method of modeling decision-making within an environment. We often use this method when utilizing model-based RL algorithms. In Q-Learning, we do not try to model the

MDP. Instead, we try to learn the optimal policy by estimating the action-value function $Q^*(s, a)$, yielding maximum expected reward in state s executing action a . The optimal policy can then be found by

$$\pi(s) = \operatorname{argmax}_a Q^*(s, a) \quad (1)$$

This is derived from *Bellman's Equation*, because we can consider $U(s) = \max_a Q(s, a)$, the Utility function to be true. This gives us the ability to derive following update-rule equation from Bellman's work:

$$Q(s, a) \leftarrow Q(s, a) + \underbrace{\alpha}_{\text{Learning Rate}} \left(\underbrace{R(s)}_{\text{Reward}} + \underbrace{\gamma}_{\text{Discount}} \underbrace{\max_{a'} Q(s', a')}_{\text{New Estimate}} - \underbrace{Q(s, a)}_{\text{Old Estimate}} \right) \quad (2)$$

This is an iterative process of propagating back the estimated Q-value for each discrete time-step in the environment. It is guaranteed to converge towards the optimal action-value function, $Q_i \rightarrow Q^*$ as $i \rightarrow \infty$ [1, 7]. At the most basic level, Q-Learning utilize a table for storing (s, a, r, s') pairs. But we can instead use a non-linear function approximation in order to approximate $Q(s, a; \theta)$. θ describes tunable parameters for approximator. Artificial Neural Networks (ANN) are a popular function approximator, but training using ANN is relatively unstable. We define the loss function as following.

$$L(\theta_i) = E[(r + \gamma \max_{a'} Q(s', a'; \theta_i) - Q(s, a; \theta_i))^2] \quad (3)$$

As we can see, this equation uses Bellman equation to calculate the loss for the gradient descent. To combat training instability, we use *Experience Replay*. This is a memory module which stores memories from experienced states and draws a uniform distribution of experiences to train the network [1]. This is what we call a *Deep Q-Network* and are as described in its most primitive form. See related work for recent advancements in DQN.

4 Deep Line Wars

For a player to play RTS games well, he typically needs to master high difficulty strategies. Most RTS strategies incorporate

- Build strategies,
- Economy management,
- Defense evaluation, and
- Offense evaluation.

These objectives are easy to master when separated but become hard to perfect when together. Starcraft II is one of the most popular RTS games, but due to its complexity, it is not expected that an AI-based system can beat this game anytime soon. At the very least, state-of-the-art Deep Q-Learning is not directly applicable. Blizzard entertainment and Google DeepMind has collaborated on

an interface to the Starcraft II game [8,9]. Starcraft II is for many researchers considered the next big goal in AI research. Warcraft III is relatable to Starcraft II as they are the same genre and have near identical game mechanics.

Current state-of-the-art algorithms struggle to learn objectives in the state-space because the action-space is too abstract [10]. State and action spaces define the range of possible configurations a game board can have. Existing DQN models use pixel data as input and objectively maps state to action [1]. This works when the game objective is closely linked to an action, such as controlling a paddle in Breakout, where the correct action is quickly rewarded, and a wrong action quickly punished. This is not possible in RTS games. If the objective is to win the game, an action will only be rewarded or punished after minutes or even hours of gameplay. Furthermore, gameplay would consist of thousands of actions and only combined will they result in a reward or punishment.

Game Property Chart					
Game	Stochastic	Partial Observable	Simultaneous	Solved	Date
Tic Tac Toe	NO	NO	NO	YES	1970's
Connect Four					Deep Blue 1996
Chess					DeepMind 2015
GO (19x19)					1979
Backgammon	YES	NO	NO	NO	Microsoft 2017
Deep Line Wars	YES	BOTH	YES		NO
Ms. Pac Man	NO	NO	YES	YES	Microsoft 2017
Starcraft II	YES			NO	

Fig. 2. Properties of selected game environments

Collected data in Fig. 2 argues that games that have been solved by current state-of-the-art is usually non-stochastic and is fully observable. Also, current AI prefers environments which are not simultaneous, meaning they can be paused between each state transition. This makes sense because hardware still limits advances in AI.

By doing rough estimations of the state-space in-game environments from Fig. 2, it is clear that state-of-the-art has done a big leap in recent years. With the most recent contribution being Ms. Pac-Man [3]. However, by computing the state-space of a regular Starcraft II map only taking unit compositions into account, the state space can be calculated to be $(128 \times 128)^{400} = 16384^{400} = 10^{1685}$ [11].

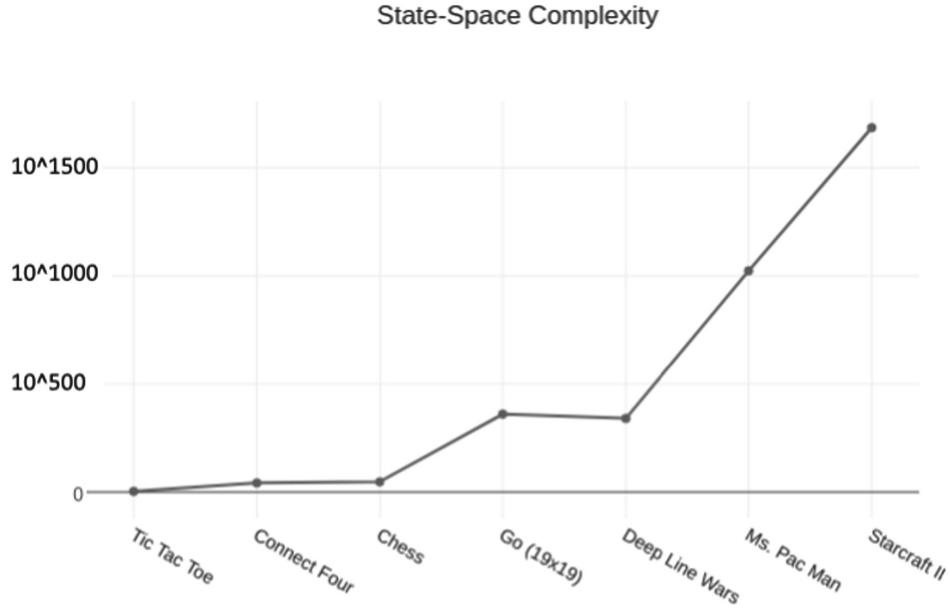


Fig. 3. State-space complexity of selected game environments

The predicament is that the difference in complexity between Ms. Pac-Man and Starcraft II is tremendous. Figure 3 illustrates a relative and subjective comparison between state-complexity in relevant game environments. State-space complexity describes approximately how many different game configurations a game can have. It is based on map size, unit position, and unit actions. The comparison is a bit arbitrary because the games are complex in different manners. However, there is no doubt that the distance between Ms. Pac-Man, perhaps the most advanced computer game mastered so far, and Starcraft II is colossal. To advance AI solutions towards Starcraft II, we argue that there is a need for several new game environments that exceed the complexity of existing games and challenge researches on multi-agent issues closely related to Starcraft II [12]. We, therefore, introduce Deep Line Wars as a two-player variant of Tower Line Wars. Deep Line Wars is a game simulator aimed at filling the gap between Atari 2600 and Starcraft II. It features the most important aspects of an RTS game.

The objective of this game is as seen in Fig. 4 to invade the opposing player with units until all health is consumed. The opposing player's health is reduced for each friendly unit that enters the red area of the map. A unit spawns at a random location on the red line of the controlling player's side and automatically



Fig. 4. Graphical interface of Deep Line Wars

walks towards the enemy base. To protect your base against units, the player can build towers which shoot projectiles at enemy units. When an enemy unit dies, a fair percentage of the unit value is given to the player. When a player sends a unit, the income variable is increased by a defined percentage of the unit value. Players gold are increased at regular intervals determined in the configuration files. To master Deep Line Wars, the player must learn following skill-set:

- offensive strategies of spawning units,
- defending against the opposing player’s invasions, and
- maintain a healthy balance between offensive and defensive in order to maximize income

and is guaranteed a victory if mastered better than the opposing player.

Because the game is specifically targeted towards machine learning, the game-state is defined as a multi-dimensional matrix. Figure 5 represents a $5 \times 30 \times 11$ state-space that contains all relevant board information at current time-step. It is therefore easy to cherry-pick required state-information when using it in algorithms. Deep Line Wars also features possibilities of making an abstract representation of the state-space, seen in Fig. 6. This is a heat-map that represent the state (Fig. 5) as a lower-dimensional state-space. Heat-maps also allows the developer to remove noise that causes the model to diverge from the optimal policy, see Formula 3.

We need to reduce the complexity of the state-space to speed up training. Using heat-maps made it possible to encode the five-dimensional state information into three dimensions. These dimensions are RGB values that we can find in imaging. Figure 6 show how the state is seen from the perspective of player 1 using gray-scale heatmaps. We define

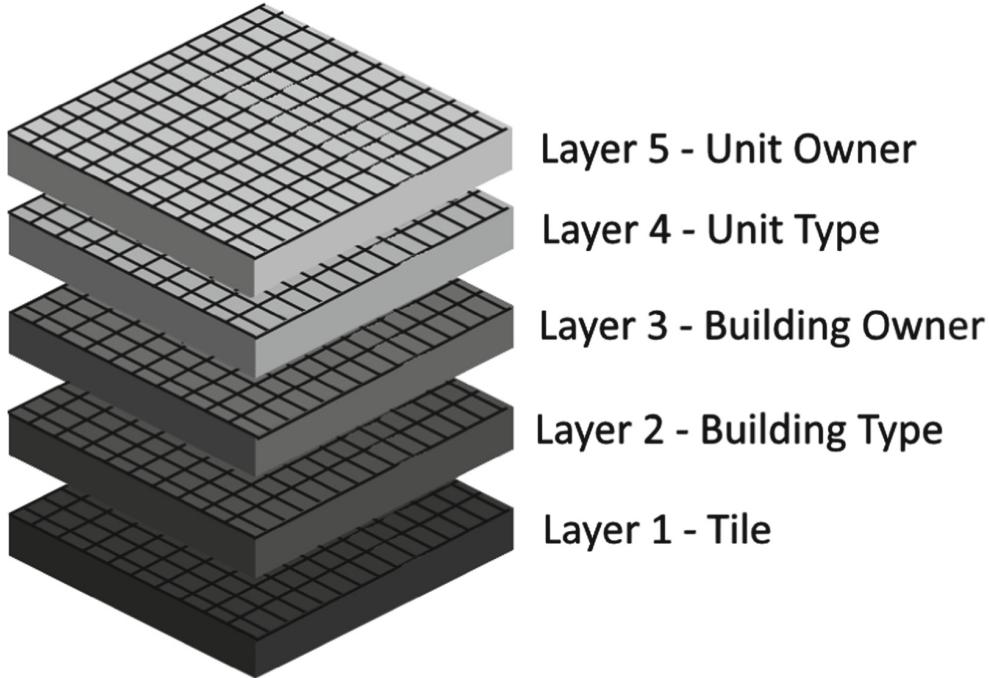


Fig. 5. Game-state representation

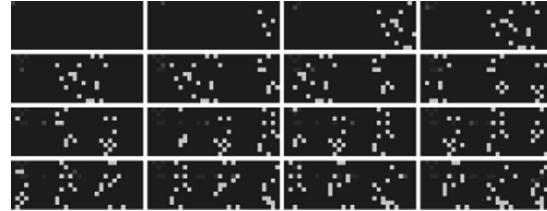


Fig. 6. State abstraction using gray-scale heat-maps

- red pixels as friendly buildings,
- green pixels as enemy units, and
- teal pixels as the mouse cursor.

We also included an option to reduce the state-space to a one-dimensional matrix using gray-scale imaging. Each of the above features is then represented by a value between 0 and 1. We do this because Convolutional Neural Networks are computational demanding, and by reducing input dimensionality, we can speed up training. [1] We do not down-scale images because the environment is only 30×11 pixels large. The state cannot be described fully by these heat-maps as there are economics, health, and income that must be interpreted separately. This is solved by having a 1-dimensional vectorized representation of the data, that can be fed into the model.

5 DeepQRewardNetwork

The main contribution in this paper is the game environment presented in Sect. 4. A key element is to show that the game environment is working properly and we, therefore, introduce a learning algorithm trying to play the game. This is in no way meant as a perfect solver for Deep Line Wars, but rather as a proof of concept that learning algorithms can be applied in the Deep Line Wars environment. In our solution we consider the environment as a MDP having state set S , action set A , and a reward function set R . Each of the weighted reward functions derives from a specific agent within the MDP and defines the absolute reward of the environment R_{env} with following equation:

$$R_{env}(s, a) = \sum_{i=1}^n w_i R_i(s, a) \quad (4)$$

where $R_{env}(s, a)$ is the weighted sum w_i of reward function(s) $R_i(s, a)$. The proposed algorithm model is a method of dividing the ultimate problem into separate smaller problems which can be trivialized with certain kinds of generic algorithms.

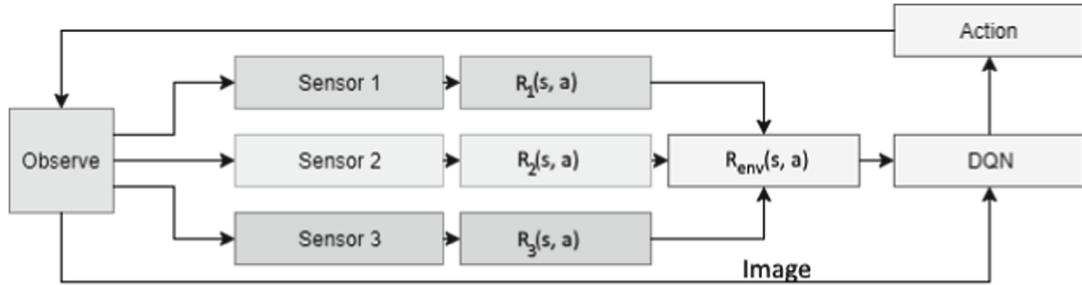


Fig. 7. Separation of the reward function

When reward for the observed state is calculated, we calculate the Q-value of $Q(s, a)$ utilizing R_{env} by using a variant of DQN.

6 Experiments

We conducted experiments with several deep learning algorithms in order to benchmark current state-of-the-art put up against a multi-agent, multi-sensory environment. The experiments were conducted in Deep Line Wars, a multi-agent, multi-sensory environment. All algorithms were benchmarked with identical game parameters.

We tested *DeepQNetwork*, a state-of-the-art DQN from Mnih et al. [1], *DeepQRewardNetwork*, rule-based, and random behaviour. Each of the algorithms was tested with several configurations, seen in Fig. 8. We did not expect any of these

Algorithm	Double Q-Learning	Prioritized Replay	Dueling DQN
Deep Q-Network	YES	YES/NO	NO
DeepQRewardNet...	NO	NO	NO
Random	N/A	N/A	N/A
Rule Based	N/A	N/A	N/A

Fig. 8. Property matrix of tested algorithms

algorithms to beat the rule-based challenge due to the difficulty of the AI. The extended execution graph algorithm (see Sect. 7) was not part of the test bed because it was not able to compete with any of the simpler DQN algorithms without guided mouse management.

Tests were done using Intel I7-4770k, 64 GB RAM and NVIDIA Geforce GTX 1080TI. Each of the algorithms was trained/executed for 1500 episodes. Each episode is considered to be a game that either of the players wins, or the 600 s time limit is reached. DQN had a discount-factor of 0.99, learning rate of 0.001 and batch-size of 32.

Throughout the learning process, we can see that DeepQNetwork and DeepQRewardNetwork learn to perform resource management correctly. Figure 9 illustrates income throughout learning from 1500 episodes. The random player is presented as an aggregated average of 1500 games, but the remaining algorithms are only single instances. It is not practical to perform more than a single run of the Deep Learning algorithms because it takes several minutes per episode to finish which sums up to a huge learning time.

Figure 9 shows that the proposed algorithms outperform random behavior after relatively few episodes. DeepQRewardNetwork performs approximately 33% better than DeepQNetwork. We believe that this is because the reward function $R(s, a)$ is better defined and therefore easier to learn the optimal policy in a shorter period of time. These results show that DeepQRewardNetwork converges towards the optimal policy better, but as seen in Fig. 9 diverges after approximately 1300 games. The reason for the divergence is that experience replay does not correctly batch important memories to the model. This causes

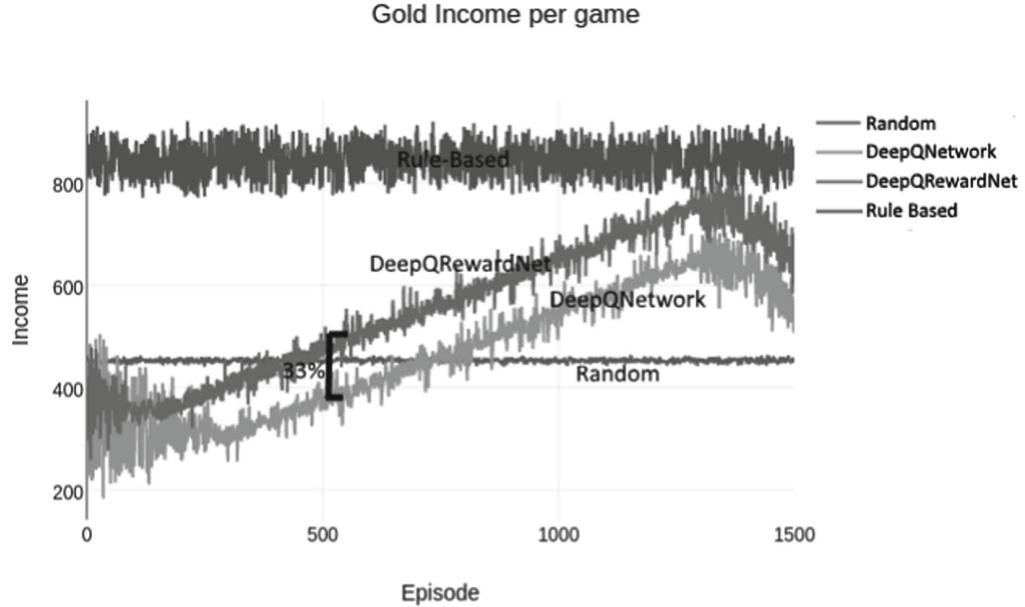


Fig. 9. Income after each episode

the model to train on unimportant memories and diverges the model. This is considered a part of future work and is addressed more thoroughly in Sect. 7. The rule-based algorithm can be regarded as an average player and can be compared to human level in this game environment.

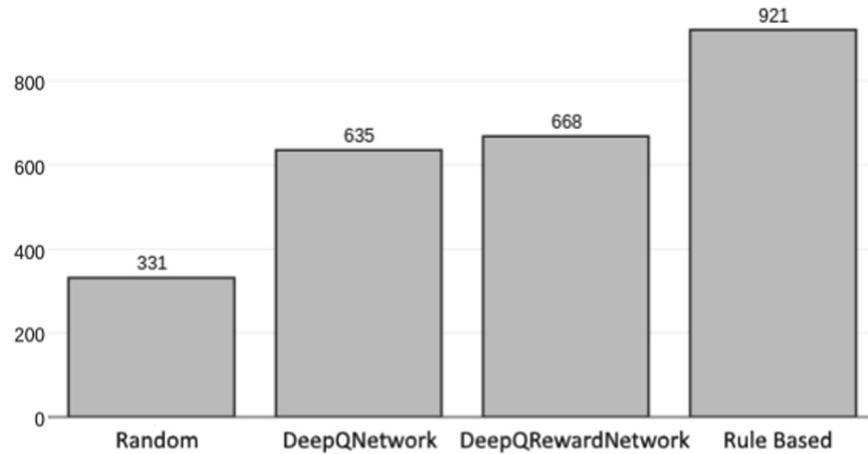


Fig. 10. Victory distribution of tested algorithms

Figure 10 shows that DeepQNetwork and DeepQRewardNetwork have about 63–67% win ratio throughout the learning process. Compared to the rule-based AI it does not qualify to be near mastering the game, but we can see that it outperforms random behavior in the game environment.

7 Future Work

This paper introduced a new learning environment for reinforcement learning and applied state-of-the-art Deep-Q Learning to the problem. Some initial results showed progress towards an AI that could beat a rule-based AI. There are still several challenges that must be addressed for an unsupervised AI to learn complex environments like Line Tower Wars. Mouse input based games are difficult to map to an abstract state representation, because there are a huge number of sequenced mouse clicks that are required, to correctly act in the game. DQN cannot at current state handle long sequences of actions and must be guided in-order to succeed. Finding a solution to this problem without guiding is thought to be the biggest blocker for these types of environments, and will be the focus for future work.

DeepQNetwork and DeepQRewardNetwork had issues with divergence after approximately 1300 episodes. This is because our experience replay algorithm did not take into account that the majority of experiences are bad. It could not successfully prioritize the important memories. As future work, we propose to instead use prioritized experience replay from Schaul et al. [13].

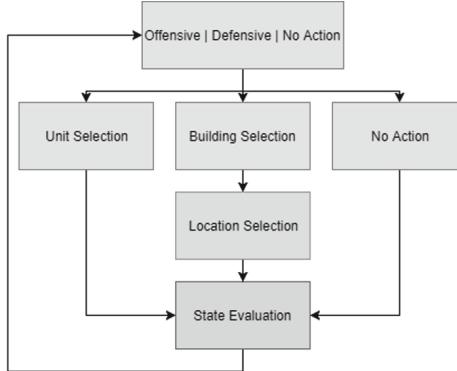


Fig. 11. Divide and conquer execution graph

Figure 7 show that different sensors separate the reward from the environment to obtain a more precise reward bound to an action. In our research, we developed an algorithm that utilizes different models based on which state the player has. Figure 11 show the general idea, where the state is categorized into three different types *Offensive*, *Defensive*, and *No Action*. This state is evaluated by a Convolutional Neural Network and outputs a one-hot vector that signal which state the player is currently in. Each of the blocks in Fig. 11 then represents a form of state-modeling that is determined by the programmer. Our initial tests did not yield any promising results, but according to the Bellman equations, it is a qualified way of evaluating the state and successfully perform learning, on an iterative basis.

8 Conclusion

Deep Line Wars is a simple but yet advanced Real-Time (strategy) game simulator, which attempts to fill the gap between Atari 2600 and Starcraft II. DQN shows promising initial results but is far from perfect in current state-of-the-art. An attempt in making abstractions in the reward signal yielded some improved performance, but at the cost of a more generalized solution. Because of the enormous state-space, DQN cannot compete with simple rule-based algorithms. We believe that this is caused by specifically the mouse input which requires some understanding of the state to perform well. This also causes the algorithm to overestimate some actions, specifically the offensive actions, because the algorithm is not able to correctly build defensive without getting negative rewards. It is imperative that a solution of the mouse input actions are found before DQN can perform better. A potential approach could be using the StarCraft II API to get additional training data, including mouse sequences [14].

References

1. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing ATARI with deep reinforcement learning. In: NIPS Deep Learning Workshop (2013)
2. Mirowski, P., Pascanu, R., Viola, F., Soyer, H., Ballard, A.J., Banino, A., Denil, M., Goroshin, R., Sifre, L., Kavukcuoglu, K., Kumaran, D., Hadsell, R.: Learning to navigate in complex environments. CoRR abs/1611.03673 (2016)
3. van Seijen, H., Fatemi, M., Romoff, J., Larochelle, R., Barnes, T., Tsang, J.: Hybrid reward architecture for reinforcement learning. abs/1706.04208 (2017)
4. Gosavi, A.: Reinforcement learning: a tutorial survey and recent advances. INFORMS J. Comput. **21**(2), 178–192 (2009)
5. van Hasselt, H., Guez, A., Silver, D.: Deep reinforcement learning with double q-learning. CoRR abs/1509.06461 (2015)
6. Wang, Z., de Freitas, N., Lanctot, M.: Dueling network architectures for deep reinforcement learning. CoRR abs/1511.06581 (2015)
7. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press (1998)
8. Traysent: Starcraft ii api - technical design, November. <https://us.battle.net/forums/en/sc2/topic/20751114921>
9. Vinyals, O.: Deepmind and blizzard to release starcraft ii as an ai research environment, November 2016. <https://deepmind.com/blog/deepmind-and-blizzard-release-starcraft-ii-ai-research-environment/>
10. Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. CoRR abs/1509.02971 (2015)
11. Uriarte, A., Ontañón, S.: Game-tree search over high-level game states in RTS games, October 2014
12. Bellemare, M.G., Naddaf, Y., Veness, J., Bowling, M.: The arcade learning environment: an evaluation platform for general agents. CoRR abs/1207.4708 (2012)
13. Schaul, T., Quan, J., Antonoglou, I., Silver, D.: Prioritized experience replay. CoRR abs/1511.05952 (2015)

14. Vinyals, O., Ewalds, T., Bartunov, S., Georgiev, P., Sasha Vezhnevets, A., Yeo, M., Makhzani, A., Küttler, H., Agapiou, J., Schrittwieser, J., Quan, J., Gaffney, S., Petersen, S., Simonyan, K., Schaul, T., van Hasselt, H., Silver, D., Lillicrap, T., Calderone, K., Keet, P., Brunasso, A., Lawrence, D., Ekermo, A., Repp, J., Tsing, R.: StarCraft II: a new challenge for reinforcement learning. ArXiv e-prints, August 2017

Appendix B

FlashRL: A Reinforcement Learning Platform for Flash Games

FlashRL: A Reinforcement Learning Platform for Flash Games

Per-Arne Andersen Morten Goodwin
Ole-Christoffer Granmo

University of Agder, Faculty of Engineering and Science
Serviceboks 509, NO-4898 Grimstad, Norway

Abstract

Reinforcement Learning (RL) is a research area that has blossomed tremendously in recent years and has shown remarkable potential in among others successfully playing computer games. However, there only exists a few game platforms that provide diversity in tasks and state-space needed to advance RL algorithms. The existing platforms offer RL access to Atari- and a few web-based games, but no platform fully expose access to Flash games. This is unfortunate because applying RL to Flash games have potential to push the research of RL algorithms.

This paper introduces the Flash Reinforcement Learning platform (FlashRL) which attempts to fill this gap by providing an environment for thousands of Flash games on a novel platform for Flash automation. It opens up easy experimentation with RL algorithms for Flash games, which has previously been challenging. The platform shows excellent performance with as little as 5% CPU utilization on consumer hardware. It shows promising results for novel reinforcement learning algorithms.

1 Introduction

There are several challenges related to developing algorithms that can interact with human-level performance in real-world environments, such as computer games. Researchers often use toy experiments when working with *Reinforcement Learning* (RL), because it is easier, cheaper and consumes less time to orchestrate. With several applications for RL in daily life, it has become an essential field of research [13, 4]. However, existing learning platforms for games have major limitations such as few game environments and little environment control.

OpenAI is a non-profit company that is currently one of the leading researchers of RL. *OpenAI Universe* is a software platform that has several game environments aimed at artificial research. The problem with this software is that individual developers are not directly permitted to supplement new environments to the repository, and there is little documentation on how to contribute to new environments. *FlashRL* changes this with our proposed architecture as the control is given back to each researcher.

Adobe Flash is a multimedia software platform used for the production of applications and animation. The Flash run-time was recently declared deprecated by Adobe, and by 2020, no longer supported. Flash is still frequently used in web applications, and there are several thousand games created for this platform. Several browsers have removed support for Flash, making it impossible to access the mentioned game environments. Games have proven to be an excellent area of machine learning benchmarking, due to size and diversity of its state-space. It is therefore essential to preserve Flash as an environment for reinforcement learning.

Automating Flash applications is a relatively untouched area. The technology has been succeeded by several better options for web development, for example, HTML5. This makes it hard for algorithms to control Flash environments programmatically. There are already reinforcement learning platforms that support Flash games as part of their game library, but these use browsers to execute the Flash run-time.

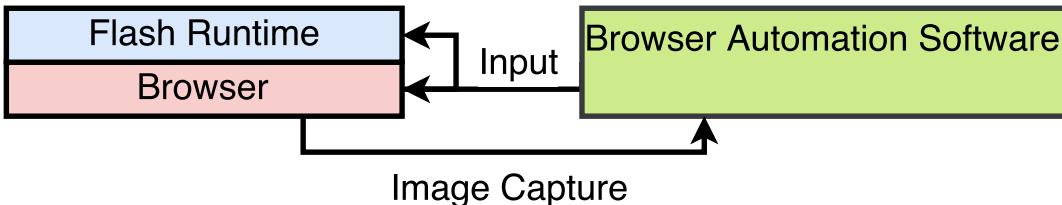


Figure 1: Interacting with Flash through browser automating

Figure 1 illustrates how interacting with the Flash environment would typically be carried out through browser automation software such as *Selenium*. *Selenium* can automate most modern browsers. It does not directly support Flash automation, but can easily be used for this purpose with minimal customisation [3]. With the loss of browser support, the difficulty of controlling Flash applications increases, and there is a significant risk that excellent game environments for reinforcement learning are lost.

FlashRL is unique for reinforcement learning as it allows researchers to use any desired Flash environment. It gives full control of the game environment and is not based on running Flash applications in the browser.

FlashRL is targeted research in reinforcement learning, but can also be used in other machine learning algorithms. It supports all kinds of Flash applications but is primarily used for agent-based gameplay. Several thousand game environments are included in the first release of the software¹. Multitask 2 is a Flash game that is excellent for reinforcement learning as it requires the agent to perform several tasks simultaneously. We show in this paper that our learning platform can be used to train novel reinforcement algorithms without any customisation.

In Section 2, we discuss related work for existing learning platforms in machine learning. We also argue why web browsers are no longer viable as Flash runtime. Section 3 briefly outline what reinforcement learning is and explains how Q-Learning works. Section 4 outlines the proposed platform and thoroughly describe its underlying architecture. In Section 5 we show initial results of utilizing the proposed learning platform for reinforcement learning. At Section 6 summarises the work and argue why the proposed learning platform is used for reinforcement learning research. Section 7 outlines a road-map for further development of the platform.

2 Related Work

With the increasing popularity in RL, there is a need for flexible learning platforms. Several learning platforms exist that can run a limited number of games, but no platform that features an open-source interface with possibility to run *any* Flash game.

Bellemare et al. provided in 2012 a learning platform *Arcade Learning Environment* (ALE) that enabled scientists to conduct edge research in general deep learning [1]. The package provided hundreds of Atari 2600 environments that in 2013 allowed Minh et al. to do a breakthrough with Deep Q-Learning and A3C. The platform has been a key component in several breakthroughs in RL research. [11, 9, 8]

In 2016, Brockman et al. from OpenAI released GYM which they referred to as "*a toolkit for developing and comparing reinforcement learning algorithms*" [2]. GYM provides various types of environments from following technologies [2]: Algorithmic tasks, Atari 2600, Board games, Box2d physics engine, MuJoCo physics engine, and Text-based environments. OpenAI also hosts a website where researchers can submit their performance for comparison between algorithms. GYM is open-source and encourages researchers to add support for their environments.

OpenAI recently released a new learning platform called *Universe*. This environment further adds support for environments running inside VNC. It also supports running Flash games and browser applications. However, despite OpenAI's open-source policy, they do not allow researchers to add new environments to the repository. This limits the possibilities of running any environment. Universe is, however, a significant learning platform as it also has support for desktop games like Grand Theft Auto IV, that allow for research in autonomous driving [7].

Selenium is a software for automating web browsers and is used primarily for unit-testing of web content. There were some efforts to create a version that allowed to interact with Flash content, but it was quickly abandoned. There is limited support for interacting with Flash, by selecting the DOM-Element in HTML and sending

¹Author of this paper takes no credit for any game environments

key-presses via Javascript. Several learning platforms utilize this method, but due to the deprecation of Flash in browsers, it is no longer a viable option.

3 Reinforcement Learning

Reinforcement learning can be considered hybrid between supervised and unsupervised learning. We implement what we call an agent that acts in our environment. This agent is placed in the unknown environment where it tries to maximize the environmental reward [14].

Markov Decision Process (MDP) is a mathematical method of modeling decision-making within an environment. We often use this technique when utilizing model-based RL algorithms. In *Q-Learning*, we do not try to model the MDP. Instead, we try to learn the optimal policy by estimating the action-value function $Q^*(s, a)$, yielding maximum expected reward in state s executing action a . The optimal policy can then be found by

$$\pi(s) = \operatorname{argmax}_a Q^*(s, a) \quad (1)$$

This is derived from *Bellman's Equation*, because we can consider $U(s) = \max_a Q(s, a)$, the utility function to be true. This gives us the ability to derive following update-rule equation from Bellman's work:

$$Q(s, a) \leftarrow Q(s, a) + \underbrace{\alpha}_{\text{LearningRate}} \left(\underbrace{R(s)}_{\text{Reward}} + \underbrace{\gamma}_{\text{Discount factor}} \underbrace{\max_{a'} Q(s', a')}_{\text{New Estimate}} - \underbrace{Q(s, a)}_{\text{Old Estimate}} \right) \quad (2)$$

This is an iterative process of propagating back the estimated Q-value for each discrete time-step in the environment. It is guaranteed to converge towards the optimal action-value function, $Q_i \rightarrow Q^*$ as $i \rightarrow \infty$ [14, 10]. At the most basic level, Q-Learning utilize a table for storing (s, a, r, s') pairs. But we can instead use a non-linear function approximation in order to approximate $Q(s, a; \theta)$. θ describes tunable parameters for approximator. Artificial Neural Networks (ANN) are a popular function approximator, but training using ANN is relatively unstable.

4 Flash Reinforcement Learning (FlashRL)

The proposed platform is an interface that acts as a bridge between the *Gnash Flash player* and the reinforcement learning algorithms. *Flash Reinforcement Learning* (FlashRL) is a new platform that allows researchers to run algorithms on any Flash-based game efficiently.

The learning platform is developed primarily for the operating system Linux but is likely to run on Cygwin with few modifications. There are several key components that FlashRL uses to operate adequate, see Figure 2. It uses a Linux library called XVFB to create a virtual frame-buffer that is used for graphics rendering [6]. Inside this frame-buffer, a Flash game chosen by the researcher is executed by a third party flash player, for example, *Gnash*. A VNC server serves the XVFB frame-buffer and allows FlashRL to access it by utilizing a VNC Client. The VNC Client can then issue commands like keyboard presses and mouse movements. The VNC Client *pyVLC* was specially made for this learning platform. The code base originates from python-vnc-viewer [15]. The last component of FlashRL is the Reinforcement

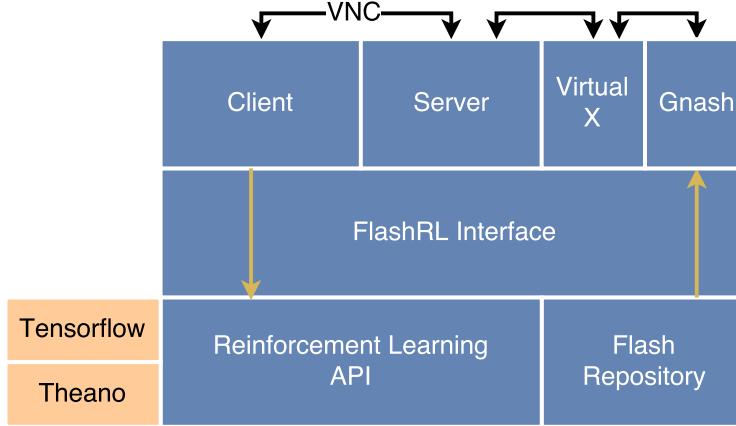


Figure 2: FlashRL Architecture Overview

Learning API that allows the developer to access the input/output of the VNC client. This makes it easy to develop sequenced algorithms by using the API callbacks or manually by threading.

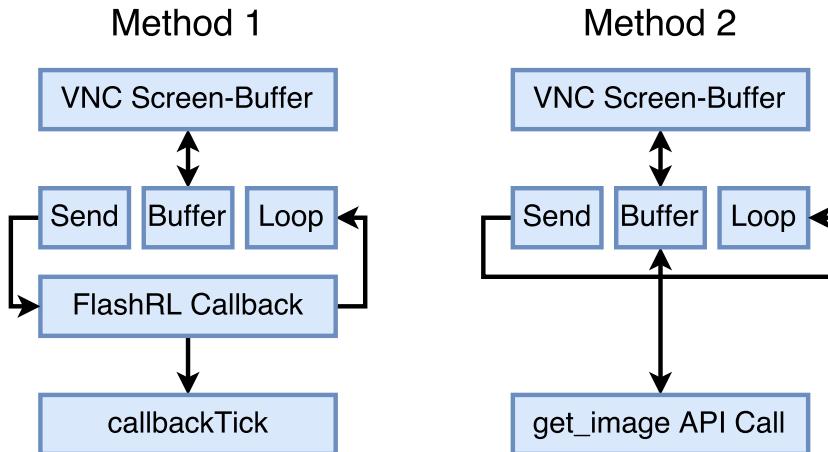


Figure 3: Frame-buffer Access Methods

Figure 3 illustrates two methods of accessing the frame-buffer from the Flash Game. Both approaches are sufficient to perform reinforcement learning, but each has its strength and weaknesses. Method 1, seen in Figure 3 allows the developer to get frames served at a fixed rate, for example, 60 frames per second. Method 2 does not restrict the frequency of how fast the frame-buffer is captured. This is preferable for developers that do not require images from fixed time-steps as it requires less processing power per frame. The framework was developed with deep learning in mind and is proven to work with Keras and Tensorflow.

Several thousand game environments are shipped with the initial version of FlashRL. These game environments were gathered from different sources on the web. FlashRL has a relatively small code-base and to preserve this size, all of the Flash games are hosted remotely. The quality varies, and some of the games are not tested or labeled. Most games are however tested and can be played without issues,

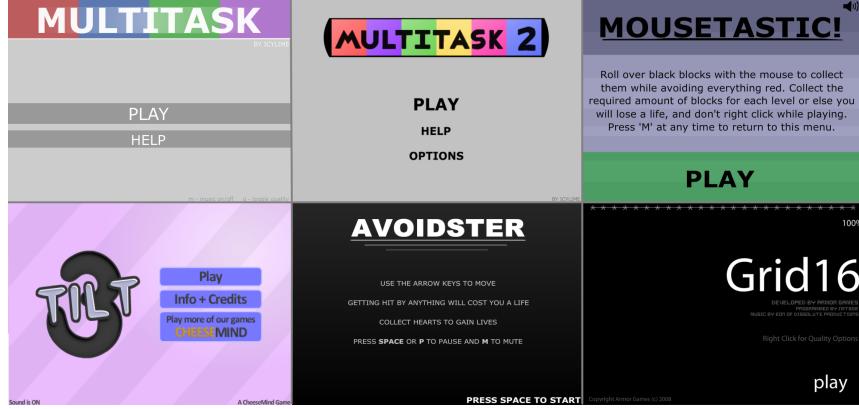


Figure 4: Selected environments from the FlashRL game repository

see Figure 4.

5 Experiments

This section presents experiments of reinforcement learning algorithms applied in FlashRL. We use the game Multitask 2² to test the learning platform. Multitask 2 was chosen because it challenges the algorithm to master four different mini-games simultaneously.

The experiments are grouped in two. The first experiment determines the hardware requirements of the platform and benchmarks the speed of critical operations. The second experiment is an implementation of standard Deep Q-Learning trained on raw state images from Multitask 2 to perform game actions. The latter is meant as a proof of concept that RL algorithms can be applied in FlashRL.

All experiments were conducted on Ubuntu Linux 17.04 x64 running Python 3.5.3. The machine has 64GB memory, Nvidia GeForce 1080TI, and Intel I7-7770k as hardware.

Multitask 2

Figure 5 illustrates the game-play of Multitask 2. The game is split into four-game phases. The first phase (lower right corner in Figure 5) is a single paddle that the player must balance a ball on. In state two (lower left corner in Figure 5), the player must control the second paddle to avoid arrows traveling towards it. The third phase (upper right corner in Figure 5) consist of an arrow with mechanics relatable to the game Flappy Bird [12]. In the final phase (upper left corner in Figure 5), the player must additionally jump over holes on the ground. For the player to succeed the game, he must control eight actions simultaneously. The score is calculated by adding a single point for each second survived in the game.

Experiment 1: Hardware Requirements

Recall from section 4 that there are two methods of accessing the frame-buffer. The first method (Method 1) is based on retrieving the frame-buffer at fixed time

²Multitask 2 - <http://multitaskgames.com/multitask-2.html>

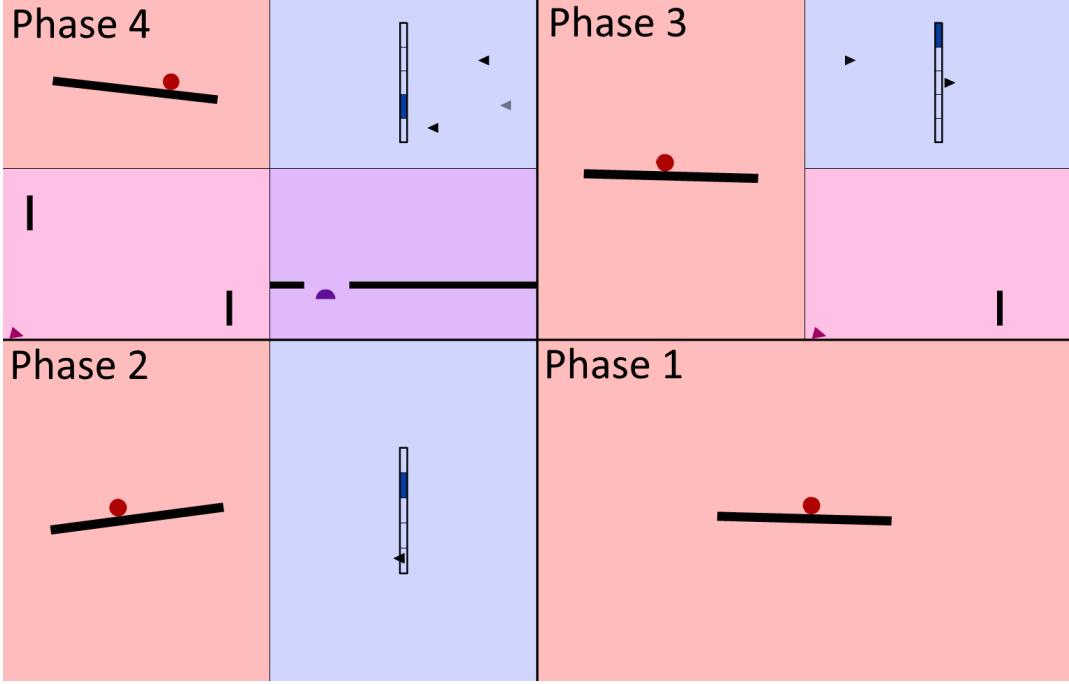


Figure 5: In-game footage of the game Multitask

intervals. The second method (Method 2) does not have any interval restriction. This makes Method 2 faster because it does not require sleep between frames. This causes the framework to consume all available CPU, which is not always preferable.

We can see from Figure 6 that using Method 1 with the interval set to 30 fps uses approximately 5% of the CPU. Increasing the interval to 300 increases it to 13%. We gradually increased the interval until the CPU ran at maximum. A single I7-7700k can compute approximately 6300 fps images from the frame-buffer before struggling to keep up.

The GPU Did not recognize any load during these test because the Flash environment is software rendered. Memory consumed were between 200MB and 500MB depending on the speed. We believe that the reason for memory increase is that Python does not garbage collect old frame-buffer snapshots between iterations, and therefore gets an increased memory load.

Experiment 2: Reinforcement Learning

Deep Q-Network (DQN) is a novel algorithm architecture developed by Minh et al. at Google DeepMind. It combines Q-Learning estimating Q-Values from a neural network. [11]

In our tests we used Double Q-Learning from Hasselt et al. [5]. We also used Dueling from Wang et al. that increases the learning precision by using two estimators: state-value and action-advantage function [16]. We used a discount factor of 0.99, learning rate of 0.001 and mini-batch of 16. We used exploration/exploitation strategy with ϵ -greedy where it started at 0.9 and finished at 0.1. The ϵ annealing was set to 10 000 steps. This is a relatively low epsilon phase. But it seemed to work well in this environment.

Figure 7 illustrates the training of DQN, where the x-axis represents episodes

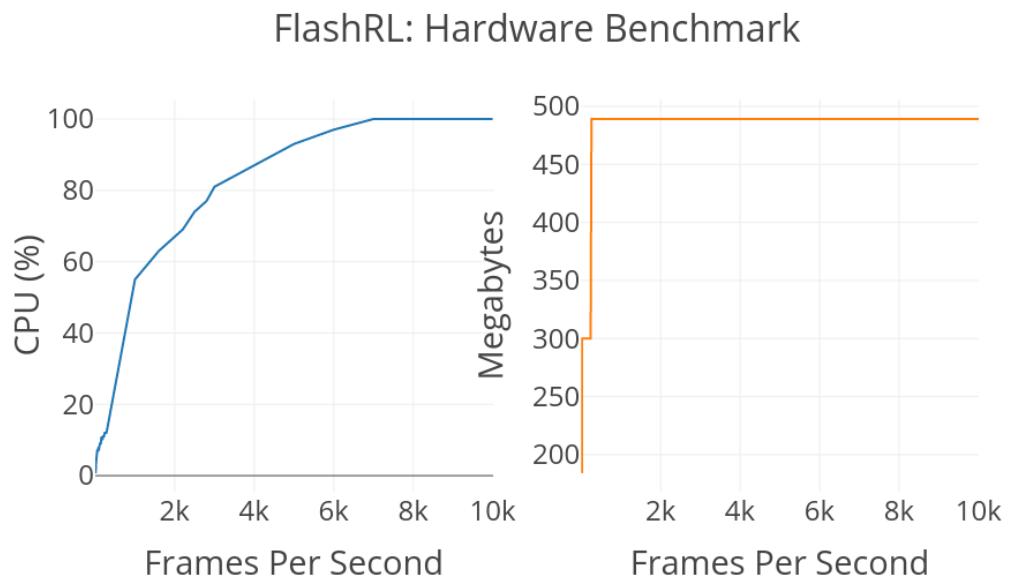


Figure 6: Hardware benchmark

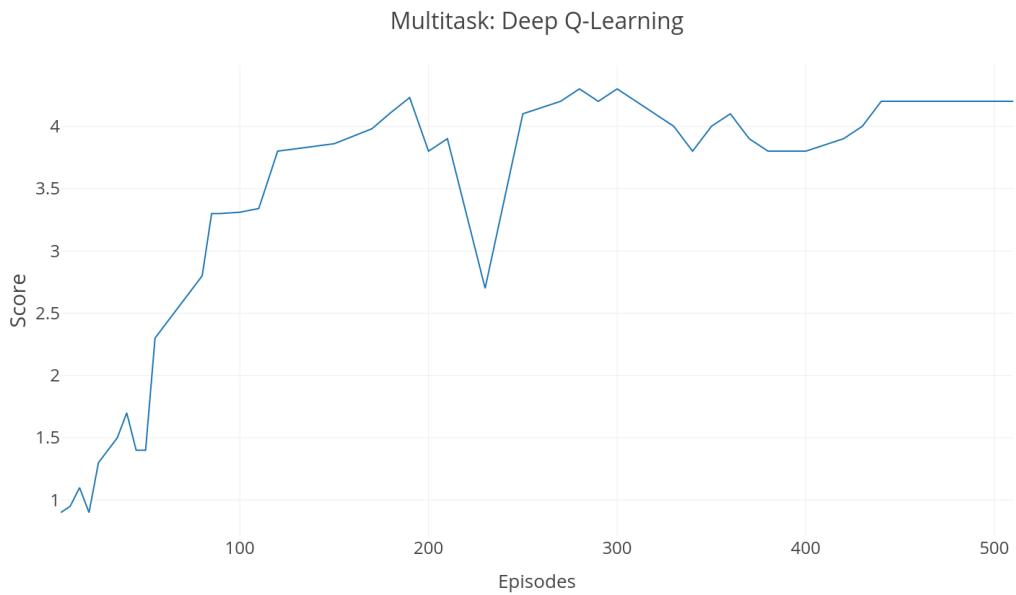


Figure 7: Deep Q-Learning Training

of the game and y-axis score before reaching the terminal state. The agent had troubles adapting to the third phase (see Section 5). Phase 3 is relatively hard to master because it requires the user balance the arrow in the air. At around 230 episodes we saw a drop in score. This is because the network seems to prioritize the first phase of the game. It reached the second phase a few times but was not able to successfully control the paddle for longer periods of time. This is why it stales at approximately 400 episodes. We believe that the network could have performed better with additional training time. It trained for a total of two days. Hopefully, it will be easier to train the network when FlashRL can speed-forward games, see section 7. The results are overall acceptable as we can see that FlashRL deliver quality states that a reinforcement learning agent can learn from.

6 Conclusion

FlashRL offers an easy-to-use architecture for performing RL in Flash-based games. It is demonstrated to work well for Multitask 2, one of the environments included. FlashRL fills the gap that emerged with the deprecation of Flash. Its main focus is RL, but can also be used for other machine learning genres. This paper shows that FlashRL can be used to train RL algorithms, in particular, Multitask 2. The work shows promising results and continuing to expand the game repository may provide new insights about RL in the future.

FlashRL will be kept alive as long as flash environments are an asset to the machine learning community. It is available to the public at <https://github.com/UIA-CAIR/FlashRL>, and can easily be adapted to every research requirement.

7 Future Work

Several improvements are planned for FlashRL. This paper outlined features of the initial version of the FlashRL, and it is by far sufficient for simple reinforcement learning research. As seen in section 5, a Deep Q-Learning based agent can successfully learn from the environment *Multitask* and gradually perform better.

Speed-forward Option

Learning algorithms often require several thousand episodes to gain expert knowledge of the environment. FlashRL is currently limited to the speed of which the game loop is executed (usually 30 fps in real-time). An important improvement would be to lift this restriction and allow algorithms to train at an accelerated rate. This would certainly improve training duration of feedback based algorithms.

Game Repository Analysis

The game repository features many unlabeled, unrated and untested games. Some games are potentially useless in a machine learning setting and require a review. The review phase is time-consuming, and authors of this paper did not have enough time to analyze each of the environments manually. The goal is to add labels and categorize all games in the repository gradually.

Website

A future goal is to allow execution of algorithms from a web interface and to add gamification aspects to the library. This would potentially create competition between researchers much like Kaggle and OpenAI Universe.

Cross-Platform Support

FlashRL is in the initial version, only supported in Python 3 on the Linux platform. The goal is to extend it so that it also can run without modifications on Microsoft Windows operating systems.

References

- [1] Marc G. Bellemare et al. “The Arcade Learning Environment: An Evaluation Platform for General Agents”. In: *CoRR* abs/1207.4708 (2012). URL: <http://arxiv.org/abs/1207.4708>.
- [2] Greg Brockman et al. *OpenAI Gym*. 2016. eprint: arXiv:1606.01540.
- [3] *Flash Testing with Selenium*. Aug. 2017. URL: <https://www.guru99.com/flash-testing-selenium.html>.
- [4] Michael E. Groot et al. “Applications of Artificial Intelligence”. In: *CAD/CAM Robotics and Factories of the Future: Volume II: Automation of Design, Analysis and Manufacturing*. Ed. by Birendra Prasad, S. N. Dwivedi, and K. B. Irani. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 165–229. ISBN: 978-3-642-52323-6. DOI: 10.1007/978-3-642-52323-6_3. URL: https://doi.org/10.1007/978-3-642-52323-6_3.
- [5] Hado van Hasselt, Arthur Guez, and David Silver. “Deep Reinforcement Learning with Double Q-learning”. In: *CoRR* abs/1509.06461 (2015). URL: <http://arxiv.org/abs/1509.06461>.
- [6] Harold L Hunt and II Jon Turney. “Cygwin/X Contributor’s Guide”. In: (2004).
- [7] Yuxi Li. “Deep Reinforcement Learning: An Overview”. In: *CoRR* abs/1701.07274 (2017). URL: <http://arxiv.org/abs/1701.07274>.
- [8] Volodymyr Mnih et al. “Asynchronous Methods for Deep Reinforcement Learning”. In: *CoRR* abs/1602.01783 (2016). URL: <http://arxiv.org/abs/1602.01783>.
- [9] Volodymyr Mnih et al. “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: <http://dx.doi.org/10.1038/nature14236>.
- [10] Volodymyr Mnih et al. “Playing Atari With Deep Reinforcement Learning”. In: *NIPS Deep Learning Workshop*. 2013.
- [11] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: *CoRR* abs/1312.5602 (2013). URL: <http://arxiv.org/abs/1312.5602>.
- [12] Matthew Piper. “How to Beat Flappy Bird: A Mixed-Integer Model Predictive Control Approach”. PhD thesis. The University of Texas at San Antonio, 2017.
- [13] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3rd. Upper Saddle River, NJ, USA: Prentice Hall Press, 2009. ISBN: 0136042597, 9780136042594.
- [14] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning : An Introduction*. MIT Press, 1998.
- [15] Techtonik. *python-vnc-viewer*. <https://github.com/techtonik/python-vnc-viewer>. 2015.
- [16] Ziyu Wang, Nando de Freitas, and Marc Lanctot. “Dueling Network Architectures for Deep Reinforcement Learning”. In: *CoRR* abs/1511.06581 (2015). URL: <http://arxiv.org/abs/1511.06581>.



UiA University of Agder
Master's thesis
Faculty of Engineering and Science
Department of ICT