Abstract

This thesis investigates how general the knowledge stored in deep-Q-networks are. This general knowledge can be used to reduce the training time of deep neural networks. Recent advances in the field of deep reinforcement learning have yielded more general solutions than previously possible. Deep architectures are computationally expensive to train, and general knowledge can be used to kick-start the training, effectively reducing training time.

We know that the low-level features in deep convolutional neural networks trained on image recognition tasks tend to be of a somewhat general nature. To investigate if this is the case for deep reinforcement learning, deep-Q-networks were implemented and trained on two similar Atari 2600 games; Pong and Breakout. First, the low-level features between two networks were visually compared. Second, the differences between the low-level features were quantified. Third, the first convolutional layer of a fully trained base network was transferred to a target network before training. This could determine if the general features in the base network would give a cutback in training time for the target network.

The results were mixed. Visually, there were few similarities between the two tasks, and many filters resembled task-specific features. Nevertheless, the quantified difference showed that there were indeed similarities. Using Breakout as base network and Pong as target network resulted in faster convergence and a possible cutback in training time. However, using Pong as base task and Breakout as target task did not. This may be due to the variation in difficulty between the two tasks.

Sammendrag

Denne avhandlingen undersøker hvor generell kunnskapen i deep-Q-networks er. Generell kunnskap kan brukes til å redusere treningstiden for dype nevrale nettverk. Nylige fremskritt innen forskning på dyp reinforcement learning har resultert i mer generelle løsninger enn tidligere. Dype arkitekturer er beregningsmessig krevende å trene, og generell kunnskap kan bli brukt til å fremskynde treningen, noe som kan redusere treningstiden.

Vi vet at lavnivå bildetrekk i dype nevrale konvolusjonsnettverk som trenes til bildegjenkjenningsoppgaver ofte er av generell natur. For å undersøke om dette også stemmer for dyp reinforcement learning, ble deep-Q-networks implementert og trent på to like Atari 2600 spill; Pong og Breakout. Først ble lavnivå bildetrekkene mellom de to nettverkene visuelt sammenlignet. Deretter ble forskjellen mellom lavnivå bildetrekkene kvantifisert. Til slutt ble konvolusjonslaget til et ferdigtrent grunnettverk overført til et målnettverk før det startet treningen. Dette skulle gi svar på om de generelle bildetrekkene i grunnettverket kunne redusere treningstiden til målnettverket.

Resultatene var varierende. Det var få visuelle likheter mellom de to oppgavene, og mange filtre virket å være oppgavespesifikke. Den kvantifiserte forskjellen viste derimot at det var likheter mellom dem. Da Breakout ble brukt som grunnettverk og Pong som målnettverk, konvergerte treningen tidligere og treningstiden kunne dermed reduseres. Men, da Pong ble brukt som grunnettverk og Breakout som målnettverk gjorde den ikke det. Grunnen til dette kan være den store forskjellen i vanskelighetsgrad mellom de to oppgavene.

Preface

This thesis concludes my Master of Science in Computer Science at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU). The supervisor for this thesis was Professor Keith L. Downing.

I would like to thank my supervisor, Keith, for the opportunity to do this project and for providing valuable feedback throughout the past months. Furthermore, I would like to thank my girlfriend, friends and family for their support. Special thanks to Olav Vatne for proofreading, his insights, and critique.

Torgeir Haaland
Trondheim, June 10, 2016

Acronyms

AGI Artificial General Intelligence.

AI Artificial intelligence.

ANN Artificial neural network.

CNN convolutional neural network.

DNN deep neural network.

DQN deep-Q-network.

GPU graphics processing unit.

MDP Markov Decision Process.

SGD Stochastic Gradient Descent.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

1.1 Overview

In this thesis, the state of the art in the field of deep learning and the generality of deep learning systems has been investigated. Further, the generality of the knowledge contained in deep-Q-networks has been studied.

This chapter gives an introduction to the thesis. Section 1.2 explains the background and motivation behind the work in this thesis. Next, Section 1.3 describes the goal and research questions. The research method used to achieve the goal is described in Section 1.4. Finally, Section 1.5 gives an overview of the structure of this.

1.2 Background and Motivation

Artificial intelligence (AI) involves the science and engineering used to make intelligent machines, especially intelligent computer programs. One subfield of AI deals with biological inspired AI, or AI that is inspired by various natural phenomena. The Artificial neural network (ANN) is one particular family of models in this category. ANNs are inspired by the biological neural networks in animals which give the brain its computational power. An ANN consists of neurons connected with each other to exchange information. Each connection has a weight that can be adjusted based on experience. This enables the networks to learn from different inputs. Although the first experiments with ANNs started in the 40s, it was not until 1974 when the backpropagation algorithm was introduced that the networks could be trained to compute any function. Hardware limitations and lack of understanding on how to efficiently train an ANN resulted

in the technique not being widely used in comparison to other machine learning techniques. This, however, changed when Hinton and Salakhutdinov [2006] proposed an efficient way of training neural networks with several hidden layers, introducing the term *deep learning*.

Today, international companies like Google, Facebook and Microsoft has their own AI divisions, employing some of the most known researchers within the field of deep learning. The technology has become part of the state of the art systems in disciplines spanning from computer vision to speech recognition.

Deep learning systems have provided great results in a range of disciplines. An example is the contributions from Mnih et al. [2015], a research team in the Google-owned company DeepMind. They combined reinforcement learning with a convolutional neural network and trained it on Atari 2600 games using only the pixel values and the score in the game as inputs. The result was an agent exceeding human expert level. Because of their results, the publication has received a lot of attention. The important thing to take from this publication, is that they used the same network architecture with the same hyperparameters for every game they trained on. There was no need to tweak or adapt the network when training a specific game. This *general* trait of the network may take us one step closer towards achieving Artificial General Intelligence (AGI).

Many subfields of specialized "narrow AI" aims to create programs for a specific task, like playing games or driving cars. AGI, on the other hand, aims to engineer general intelligence that can learn and be applied across various domains [Goertzel and Pennachin, 2007]. Ask the average man in the street what he associates with AI, and he will most probably mention R2D2, C3PO or other thinking machines from science fiction. If an academic within the field was asked the same question, the answer would be different. The AI research today is concerned with building more narrow and specialized software, far from the motivation that started the field. One can say that what the average man in the street considers AI, is the goal of the AGI community.

There is long way until AGI is achieved, but small steps are taken all the time. Mnih et al. [2015] is an example of this. Personally, I think AGI is an exciting thought and the result produced by DeepMind made me realize that a general AI is an attainable goal. It was this work that woke my interest in deep learning, and it is indeed interesting to see how general these systems can be. The motivation behind this thesis is just that, to explore how general the knowledge contained in a deep learning system is. In addition, how can its generality benefit the field of neural networks? These are some of the questions I seek an answer to when

researching the state of the art in this field of AI.

1.3 Goals and Research Questions

This section presents the task description, goal and research questions of this thesis.

Task description *Deep Learning has become an extremely successful (and hence popular) classification method in AI and Machine Learning over the past couple of years. Many of the big(gest) computer companies have bought up top-level neural network gurus to help them take advantage of this technology. In this project, the student can suggest a domain to which they would like to apply deep networks, and then they will use the Theano system to solve a problem in that domain.*

This is the task description as specified by my supervisor. From this description, goals and research questions were defined.

Goal *Investigate how general the information stored in a deep neural network is.*

The generality of deep neural networks will be investigated by considering the research questions below.

Research question 1 *Will there be any similarities between the lower level features in deep neural networks trained on similar tasks?*

We know that there is evidence that low-level features in deep convolutional neural networks contain somewhat general knowledge. The lower layers often correspond to known image processing filters for detecting edges and other similar low-level traits. This question is necessary to determine because the task a network is trained to do may affect how features are learned. If lower level features are similar, it may indicate that the knowledge contained in that part of the network is of a somewhat general nature, and may be *reused* and applied to different tasks.

Research question 2 *Will there be a significant cutback on the training time for a network already trained on a similar task? How significant?*

Deep neural networks may require millions of labeled examples to be trained in a supervised fashion. This may take days with the technology available today. It is important to find ways of making the training process more effective. If a network contains general information, this information can perhaps contribute to reducing the training time of other networks.

1.4 Research Method

A deep-Q-network system has been implemented and background reading were performed, much of it presented in Chapter 2. Experiments were designed according to the research questions and applied to the domain of Atari games utilizing the implemented system. The results have been presented, discussed and analyzed and finally, conclusions in regards to the research questions were made.

1.5 Thesis Structure

This thesis is divided into five chapters. This chapter gives a brief introduction to the thesis in addition to stating its goal and research questions. Chapter 2 contains background theory and a literature review covering relevant studies for this thesis. Chapter 3 provides a detailed description of the implemented system as well as experimental setup. In Chapter 4, the results from the experiments are presented, analyzed and discussed. Finally, Chapter 5 gives a summary of the results and relates them to the research questions.

Chapter 2

Background Theory and Motivation

2.1 Overview

This chapter contains background theory and presents research in the field of deep neural networks and deep-Q-networks. Section 2.2 introduces concepts within deep learning. Then, Section 2.3 introduces a technique to learn multiple tasks at the same time. Furthermore, Section 2.4 covers reinforcement learning. Section 2.5 describes the literature review. Then, the related work found by the literature review is presented in Section 2.6. Finally, Section 2.7 gives a summary of the studies and how they are connected to the research questions.

2.2 Deep Learning

The term *deep learning* is believed to originate from the mid 2000s when [Hinton et al., 2006] and [Hinton and Salakhutdinov, 2006] successfully managed to train a deep neural network (DNN) within reasonable time. A neural network with more than one hidden layer can be called a DNN. The additional layers make deep architectures able to learn representations of data with multiple levels of abstraction. This have resulted in breakthroughs in areas such as speech recognition and visual object recognition.

Conventional approaches to machine learning are limited in their ability to process raw data. For a long time, constructing a conventional machine learning system would often require specific knowledge of the application domain and

careful engineering to be able to create a system that processed raw data into a useful internal representation. Deep learning comprises methods that *learn* representations which can be further used to extract features from the data.

An example of this is recognition of objects in images. A deep neural network takes the raw pixel values as input. For each layer in the network, the representations are more abstracted. This property exploits the fact that images (together with other natural signals) are compositional hierarchies where higher-level features consist of combinations of lower-level features. In the lower layers, edges are often detected. In higher layers, more specific patterns are detected. For instance, in a network trained to detect human faces, the higher layers often highlights the eyes, mouth or nose. The various layers of representation make deep learning methods able to learn very complex functions and structures in high-dimensional data. This has resulted in deep learning methods beating records in image recognition and speech recognition, it has performed better than previous methods at predicting the activity of potential drug molecules, analyzing particle accelerator data, reconstructing brain circuits among others [LeCun et al., 2015].

The convolutional neural network (CNN) plays a big role in the history of deep learning. CNNs performed well before other deep learning approaches were viable. One example is the network developed by AT&T in 1998 which could read checks [LeCun et al., 1998]. The commercial interest in deep learning in the recent years started when Krizhevsky et al. [2012] won the ImageNet object recognition challenge using a CNN.

CNNs take advantage of data that comes in the form of arrays. 1-dimensional arrays for language signals, 2-dimensional arrays for images and 3-dimensional arrays for video are examples of data that CNNs have been used on with success. They use convolution instead of matrix multiplication in some of its layers. Convolution is a mathematical operation on two functions, producing a third function:

$$s(t) = (x * w)(t) \tag{2.1}$$

The continuous convolution is defined as:

$$s(t) = \int x(a)w(t-a)da \tag{2.2}$$

It is the discrete definition of convolution that is used in machine learning:

$$s[t] = (x * w)(t) = \sum_{a=-\infty}^{\infty} x[a]w[t-a] \tag{2.3}$$

In the context of CNNs, x is the input, w is the *kernel*, or the parameters that can learn and s is the output, or *feature map*. Equation 2.3 defines the discrete convolution for a single dimension, but in machine learning the input and kernel is often multidimensional arrays called tensors. Each element of the input and kernel must be explicitly stored separately. For this reason implementation of the infinite summation often use a sparse approach where the functions are assumed to be zero everywhere but the finite set of points for which values are stored. This means that the infinite summation effectively becomes the finite summation over the number of elements in the array. In addition, the convolution is often computed over more than one axis at a time. For a two-dimensional image, a two-dimensional kernel is probably also wanted. The discrete convolution over two dimensions is:

$$s[i,j] = (I * K)[i,j] = \sum_{m} \sum_{n} I[m,n]K[i-m, j-n] \tag{2.4}$$

where I is the two-dimensional image and K is the two-dimensional kernel. For images, convolution can be seen as sliding the kernel over the larger input image. The pixel values in the output image are calculated by multiplying each value in the kernel with the corresponding pixel values in the input image, see Figure 2.1.

As Goodfellow et al. [2016] explains, there are several reasons why CNNs generally performs well on data such as images. One of them is *sparse connectivity* where the kernel is smaller than the input. With images, it is not necessary with a fully connected network. A subset of the pixels is often enough to extract useful features. This results in fewer operations to compute the output as well as less memory usage. See Figure 2.2.

In traditional neural networks, the elements of the weight matrix are used once to compute the output from a layer. However, in CNNs each unit of the kernel is used for every position of the input. This means that the memory requirements is reduced as only one set of parameters are learned instead of a separate set of parameters for every location. This is called *parameter sharing*. In images, for example, this is beneficial since low-level features like edges often appear in the entire image.

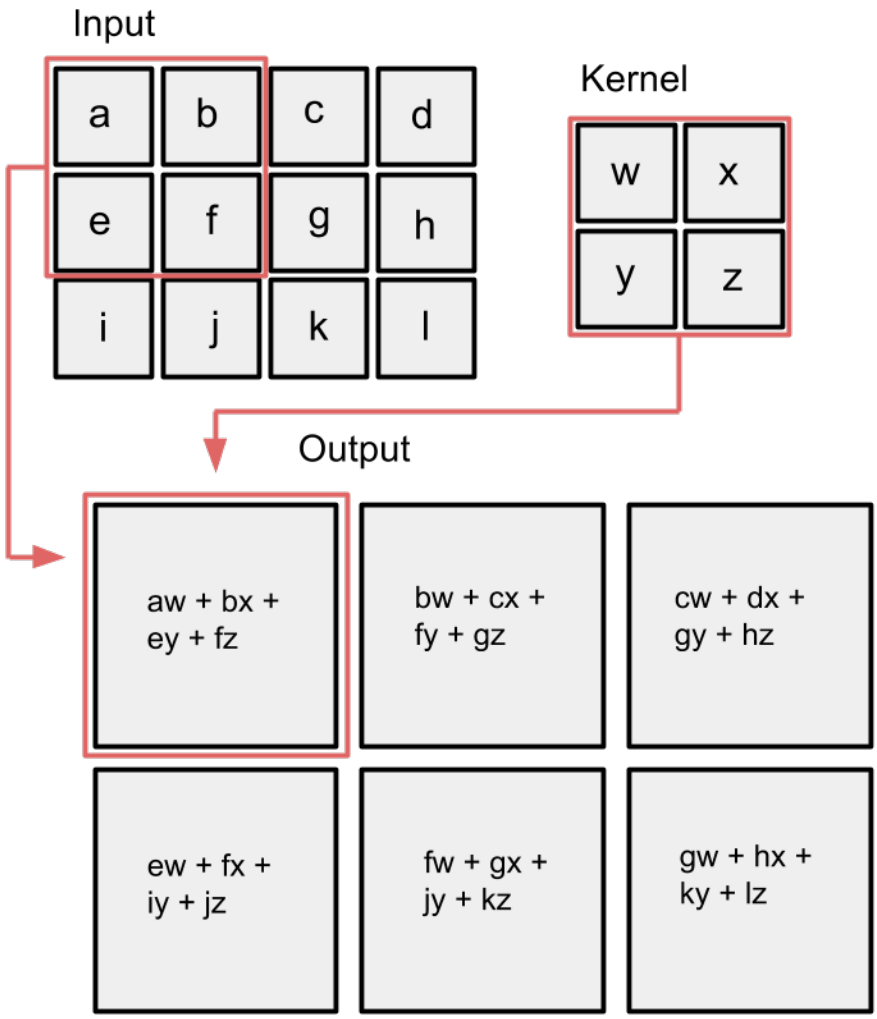


Figure 2.1: 2D convolution, figure based on Figure 9.1 in [Goodfellow et al., 2016]. The outlined part of the output tensor is formed by applying the kernel to the corresponding region of the input tensor.

Another reason CNNs have good performance on images is *pooling*. Pooling replaces the output of a specific location with a statistic summary of the nearby outputs. An example is *max pooling* which returns the max output in a neigh-

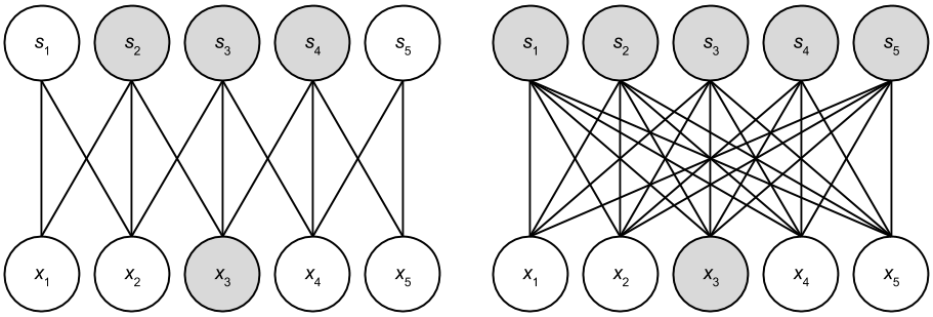


Figure 2.2: Figure based on Figure 9.2 in [Goodfellow et al., 2016]. **(Left)** When input unit x_3 is activated in a sparse CNN with kernel width = 3, only the highlighted output units $s_{2...4}$ are affected. **(Right)** When input unit x_3 is activated in a fully connected network, all output units $s_{1...5}$ are affected.

borhood. Pooling makes the representation invariant to small translations of the input. This is convenient when the exact pixel position of a feature is not relevant. In images, it is enough to know that if there is a face in it, there should be an eye on the left and an eye on the right.

For a CNN the typical stages of a layer is to first perform the convolution on the input. Second, the resulting function is passed through an activation function, often a rectifier[1]. Last, a pooling operation is often performed.

Stochastic Gradient Descent (SGD) is an algorithm used to tune the weights of a DNN. It works by taking small steps downhill on an error surface defined by a loss- or objective function. One of the main advantages of SGD is that it does not need to be run over the full training set, effectively reducing the computational cost of training while still converging relatively fast. In its simplest form only one training example is used to find and follow the negative gradient of the objective. However, by introducing *minibatches*, more than one training example can be used for each estimate of the gradient. The advantage of this is that the variance in the estimated gradient is reduced, in addition to taking advantage of the hierarchical memory organization in computers. SGD is seldom used in its pure form when training neural networks. There are a number of different techniques used to speed up convergence, one of them is *momentum*. The error surface of deep architectures often has the form of a shallow ravine

[1] Glorot et al. [2011] showed that the rectified linear unit performed better than previously used activation functions.

with steep walls. Dealing with this landscape, SGD has a tendency to oscillate across the ravine since the gradient will point down the steep walls rather than towards the minimum. By introducing momentum, the updates are pushed more quickly along the ravine, resulting in faster convergence. Adjusting the weights in CNNs during training are done with the backpropagation algorithm [Hecht-Nielsen, 1989], as with other neural networks.

With the use of deep learning methods, an increase in the available data is not hard to take advantage of. This is because the engineering needed to apply deep learning on a task is small compared to techniques where the data representation must be hand-crafted. It is not unreasonable to think that deep learning will have many more accomplishments in the coming years.

2.3 Multitask Learning

Traditionally, machine learning techniques has been based on learning from a single task at the time. Like learning to recognize cats on an image based on examples of images with or without cats. Humans and animals in the real world often use information from different domains and tasks when learning new things; this is what inspired the introduction of *multitask learning*. Multitask learning is a method that uses information from domains of related tasks in the training signal. The idea is to improve the generality of the learned knowledge by learning similar tasks from different domains in parallel and sharing the representation. The result is that what is learned for one task may also help to learn other tasks. Benefits occur due to the extra domain knowledge contained in the extra training signals.

Caruana [1997] tried using multitask learning combined with simple shallow neural networks and k-nearest neighbors. He found that multitask learning improved generalization and that the benefit from using extra tasks can be substantial. However, he stresses that the results are most promising when not using sanitized data, as the benefits from the extra information provided are often lost when engineering features by hand. For this reason, it is interesting to see how multitask learning provides benefits when being combined with DNNs. We know that one of the strengths of DNNs is that they do not need handcrafted features, but learns them.

When Caruana [1997] was published, there was not a lot of situations where multitask learning would be beneficial to use. He stated that the benefits would be more clear when the technique is used on problems closer to those encountered in a real world setting and that the opportunities for the technique will increase.

Multitask learning is used by some research mentioned in the related works (Section 2.6). However, because of the amount of computational power needed to perform experiments involving multitask learning combined with DNNs, the experiments in this thesis will not use it.

2.4 Reinforcement Learning

Reinforcement learning is a machine learning technique biologically inspired by the *basal ganglia*, a part of the brain that responds to dopamine. The neurotransmitter dopamine is released as a reward when we experience something positive, like eating good food or winning the jackpot on a slot machine. In machine learning, reinforcement learning is concerned with how an agent should take actions in an environment based on the maximum calculated cumulative reward. Unlike a supervised learning scheme which needs a correct mapping between each input and output, a reinforcement learning agent receives feedback at rarer intervals, which means that a certain feedback signal can not be tied directly to a particular action. For an agent trained on a video game, the feedback can be the score or simply if it won the game or not. By repeatedly playing the game, the agent can learn how it should behave in order to maximize the cumulative reward.

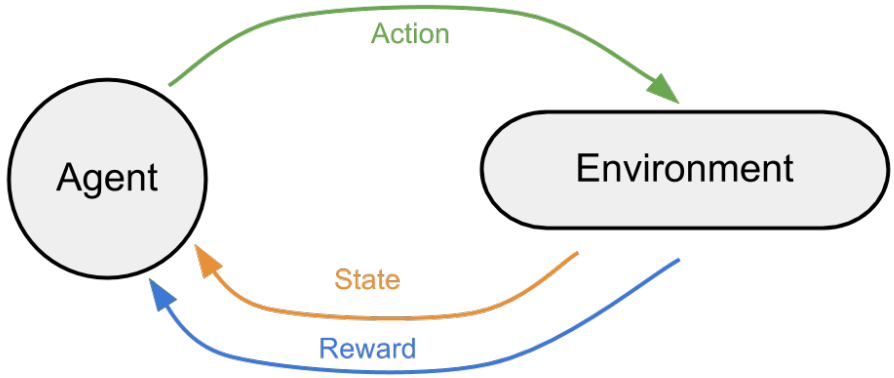


Figure 2.3: Reinforcement learning. An agent performs an action on the environment. The agent's state representation of the environment changes and a reward is obtained.

Figure 2.3 shows the general idea of reinforcement learning. The agent performs an action and the environment changes. By observing the changes in the

environment and by tying it to the reward it is receiving, it can learn to judge if its behavior is favorable.

For a reinforcement learning agent, it is not enough to always choose the action which leads to the state with highest cumulative reward. Exploitative behavior like this will create a greedy agent that seldom finds the optimal solution [Russel and Norvig, 2010]. This is because the agent has learned a model that most likely does not correlate with the true environment. Exploring the environment may improve the agent's model, leading to greater rewards in the future. The agent must find the balance between exploring and exploiting when deciding what action to take. It needs to both maximize its current reward, but at the same time maximize its long-term welfare. One behavior policy is the ϵ-greedy policy. By choosing the greedy option with a probability of $1 - \epsilon$, where ϵ is a value between 0 and 1, this policy ensures that the agent will explore. One variant of the ϵ-greedy policy starts with a high ϵ-value which decreases over time. This results in an agent that mainly explores the environment at the start. As time progresses and the environment is more known, it tends to exploit the choices that give the highest expected reward.

Many reinforcement learning methods are based on the *Bellman equation*. The Bellman equation states that *the utility of a state is the immediate reward for that state plus the expected discounted utility of the next state, assuming the agent chooses the optimal action* (Russel and Norvig [2010]). In other words, the utility of a state is given by:

$$U(s) = R(s) + \gamma max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s') \tag{2.5}$$

where $R(s)$ is the reward in state s, γ is a discount factor, a is the action and s' is the next state.

Q-learning is a model-free reinforcement learning method. By letting an agent experience the consequences of its actions, it does not need to map the domain (Watkins and Dayan [1992]). The agent will try an action at a state and decide the consequences in terms of the immediate reward (or penalty) it receives. This is added with the value, or *utility*, of the state it is taken to. The agent learns which states are best by trying all actions in all states repeatedly and thereby judge the long-term discounted reward. Russel and Norvig [2010] defines the value of a state as its utility, $U(s)$. Further, the value of doing action a in state s is $Q(s,a)$ and is called the *Q-value*. Q-learning does not learn the utilities of the states, but rather the action-utility representation. The relation between the Q-value and the utility is:

$$U(s) = max_a Q(s, a) \tag{2.6}$$

Since Q-learning does not require a model of the state transitions, only the Q-values are updated:

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma max_{a'} Q(s', a') - Q(s, a)) \tag{2.7}$$

This is calculated when action a is executed in state s leading to s'. $Q(s, a)$ is the Q-value for doing action a in state s, α is the learning rate, $R(s)$ is the reward in state s and γ is the discount factor ensuring immediate rewards are more valuable than later rewards. It is worth mentioning that Q-learning is an *off-policy* method [Sutton and Barto, 1998], meaning that it updates its Q-values using the next state s' and the greedy action a'. It estimates the total future discounted reward assuming a greedy policy was followed, although this is not the case. This trait makes Q-learning able to explore while still being able to update its Q-values with the best action available.

Reinforcement learning can be combined with other techniques to create more sophisticated learning methods. In some problem domains, it can be computationally unfeasible to use Q-learning directly, especially if the state space becomes large. Recently, Mnih et al. [2015] has received attention for combining Q-learning with a CNN. The CNN is used to extract the visual features of the game and approximate the Q-function. This reduces the computational cost needed to find the Q-values. Several tricks were incorporated to make the training converge, as it turns out it is not straight forward to approximate the Q-function with neural networks. This will be discussed further in Chapter 3.

2.5 Structured Literature Review

The purpose of this structured literature review is to explore what already exists in the field of deep learning and deep learning combined with reinforcement learning. The SLR is based on a template by Anders Kofod-Petersen, Adjunct Professor at the Department of Computer and Information Science at the Norwegian University of Science and Technology.

2.5.1 Identification of Research

Table 2.1 displays various keywords used for the search. The table consists of groups and terms. The groups are supposed to contain words that have the same

semantic meaning or words that can be interchanged in a search term. By using the logical operators OR and AND, a collection of different search terms was constructed. A combination of all groups produced a too narrow result. Hence, combinations excluding different groups were also tried. The search engines used were:

- Google Scholar
- IEEE Xplore
- SpringerLink
- ACM digital library
- arXiv
- NIPS

In addition to the research material found using this search approach, several papers were found in more informal ways. My supervisor suggested to start with papers by some leading researchers within deep learning, namely Yoshua Bengio, Yann LeCunn and Geoffrey Hinton. Furthermore, Mnih et al. [2015] was used as a starting point.

Table 2.1: Search terms.

	Group 1	**Group 2**	**Group 3**	**Group 4**
Term 1	Deep learning	Artificial general intelligence	Reinforcement learning	Transferability
Term 2	Neural network	General intelligence	Q-learning	General features
Term 3	Deep neural network	General	Atari	Transfer learning
Term 4	Convolutional neural network			General knowledge
Term 5	Machine learning			Multitask learning

2.5.2 Screening Process

By using the search terms in the mentioned search engines, 400 papers were found. A screening process was introduced to reduce the amount of papers to 20. The papers were either used in the related work section or incorporated into the

background theory.

First, duplicates were removed. Second, papers published after 2000 were preferred as it is only in the last decade or so that deep networks have been successfully used. Third, the title was used to filter out papers which was not relevant. This left a set of papers that could be filtered with respect to quality. The studies main concern should be on DNNs and preferably focus on the generality of the network. Studies combining reinforcement learning with DNNs or in some way consider techniques on how to learn more general knowledge were favorable. In addition, there should be a clear statement of the aim of the research and the study should be put into context of other studies and research.

2.6 Related Work

This section is the result of the structured literature review. A subsection for each of the most relevant studies are included. Other papers are then presented in Section 2.6.5. Then, in Section 2.7, the related work will be summarized and viewed in light of the research questions.

2.6.1 Human-Level Control Through Deep Reinforcement Learning

Humans and animals are good at deriving efficient representations of high-dimensional sensory input. This is important to use reinforcement learning successfully in situations approaching real-world complexity. Reinforcement learning has been used successfully across domains, but has previously been limited to fully observed domains, domains with handcrafted features or low-dimensional state spaces. **Mnih et al. [2015]** uses recent advances in deep neural networks to develop a novel artificial agent they call a deep-Q-network (DQN). The agent learns policies directly from high-dimensional sensory inputs using end-to-end reinforcement learning. The agent was tested on Atari 2600 games and received only pixels and the game score as input. 49 different Atari games were tested. The algorithms, network architecture and hyperparameters were kept the same between each game. The results showed that the agent performed better than all previous algorithms and achieved a level comparable to a professional game tester. The paper claims that it is the first artificial agent that is capable of learning to excel at a diverse array of challenging tasks.

Atari games comprise tasks in which the agent interacts with the environment through a sequence of observations, actions and rewards. The goal of the agent

is to maximize the cumulative future reward. To do this, a CNN is used to approximate the action-value function (see Equation 2.7 in Section 2.4).

Because of the non-linear nature of neural networks, reinforcement learning is known to be unstable when a neural network is used to represent the action-value function. This is because of:

- Correlations present in the sequence of observations.
- The fact that small updates to the Q-values may change the policy significantly and therefore change the data distribution.
- Correlations between the Q-values and the target values $r + \gamma max_{a'} Q(s', a')$.

To battle these instabilities they used a novel variant of Q-learning. First, *experience replay* was introduced which averages the behavior distribution over many of the previous states. This smoothed out learning and helped avoid oscillations or divergence in the parameters. Second, a *target network* was used, effectively adjusting the Q-values towards target values generated by the target network that are only periodically updated, reducing correlations with the target. The effect of turning these techniques off was detrimental to the performance of the agent.

To evaluate the agent, it was trained on the Atari 2600 platform. It was trained with very little prior knowledge, knowing only that the input data was visual images and the number of actions available, but not their correspondences. The method was able to train large neural networks using reinforcement learning in a stable manner. The agent outperformed existing reinforcement learning methods on 43 out of 49 games. Furthermore, the paper claimed that the agent performed at a level comparable to a professional human game tester across the set of 49 games by achieving more than 75% of the human score on more than half of the games (29 games).

The games were varied in their nature, spanning from side-scrolling shooters (River Raid) to boxing games (Boxing) and three-dimensional car racing games (Enduro). In certain games, the agent was able to discover relatively long-term strategies. For instance, in Breakout, the agent learned the optimal strategy, which was to dig a tunnel on the side allowing the ball to be sent around the back and destroy a large number of blocks. Other games that require more temporally extended planning strategies was hard for the DQN agent (Montezuma's Revenge).

2.6.2 How Transferable are Features in Deep Neural Networks?

Yosinski et al. [2014] investigates the generality versus specificity of features in deep neural networks. When training deep neural networks on images, the first few layers tend to learn features similar to Gabor filters and color blobs. These features appear to not be specific to a particular task, but general in that they are applicable to many datasets and tasks. The paper experimentally quantifies the generality versus specificity of neurons in each layer of a deep CNN. They found that transferability was negatively affected by two issues:

- The specialization of higher layer neurons to their original task at the expense of performance on the target task. This was expected.
- Optimization difficulties related to splitting networks between co-adapted neurons, i.e neurons that has developed unwanted dependencies between each other. This was *not* expected.

Other results that was found was:

- The transferability decreases as the distance between the base task and target task increases, but transferring features from distant tasks are still better than randomly initialized features.
- Initializing a network with transferred features (from almost any number of layers) can produce a boost to generalization.

The motivation to quantify generality vs specificity and to find out if the transition between them occur suddenly or is spread out over several layers is to use the knowledge in transfer learning [Bengio, 2012]. In addition it can be beneficial to find out where the transition takes place (near first layer, middle or last layer). In transfer learning, a network is first trained on a base task. The learned features are then transferred to a target network to further fine-tune them. This will work if the features are general, or suitable for both the base task and the target task. The power of transfer learning appears when a target dataset is too small to train a network without overfitting. If there is a large base dataset, a network can be trained on it and transferred to the smaller target set. This reduces overfitting drastically.

ImageNet was used to conduct the experiment. This dataset contains over 1.2 million labeled training images and 50,000 test images. Each image is labeled with one of 1,000 classes. Two tasks, A and B were created by splitting the 1,000 ImageNet classes into two random groups containing half the data each. An 8-layer convolutional neural network was trained on each task, providing the

networks *baseA* and *baseB*. A layer n was chosen from {1,2,...,7} to train several new networks. n = 3 is used in the following example.

To make a control for the next transfer network a *selffer* network, B3B, was created where the first 3 layers are copied from baseB and frozen. The five higher layers (4-8) were initialized randomly and trained on dataset B. The transfer network A3B was created by copying the first 3 layers from baseA and frozen. The five higher layers (4-8) was initialized randomly and trained on dataset B. If A3B performed as well as baseB, there was evidence that the third-layer features were general, at least with respect to B. However, if the performance suffered, the third layer most probably contained features specific to A.

This was repeated for all n in $\{1,2,...,7\}^2$ in both directions (i.e AnB and BnA). Tests were the networks were not frozen were also constructed:

- A selffer network $B3B^+$: just like B3B, but where all layers learn.
- A transfer network $A3B^+$: just like A3B, but where all layers learn.

The ImageNet dataset contains clusters of similar classes, for example different Felidae: tabby cat, tiger cat, Persian cat etc. These classes were distributed as equally as possible between A and B. To create a split of the ImageNet dataset that was as semantically different as possible, the hierarchy of parent classes of the ImageNet dataset was exploited: A containing only man-made entities and B containing natural entities.

Figure 2.4 illustrates the result of the main experiment. The AnB^+ networks, where layers are transferred and fine-tuned, improved generalization. The BnB networks seem to suffer from co-adaptation, which means that the features in the middle layers seem to develop some dependence on each other. The BnB^+ networks where layers are fine-tuned, recover the co-adapted features. The performance of AnB networks where layers are transferred and frozen suffers from both co-adaptation and the specificity of the higher layers.

The findings of Yosinski et al. [2014] were strengthened by Azizpour et al. [2015]. They came to the same conclusion, that there are performance gains to be made by fine-tuning a pre-trained network towards a target task. Their results are drawn from a wider range of target tasks than Yosinski et al. [2014].

2.6.3 Simultaneous Deep Transfer Across Domains and Tasks

Picture a manufacturer that trains a robot to visually recognize thousands of common objects. When the robot is shipped to a customer its performance will

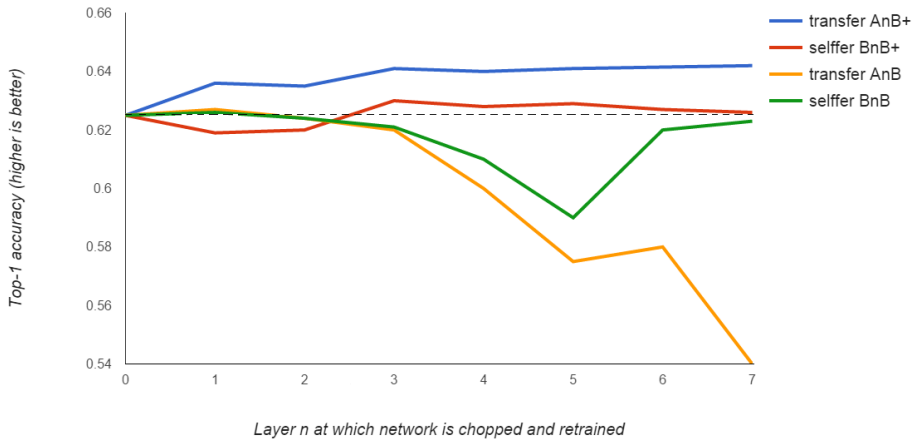


Figure 2.4: Figure based on Figure 2 in Yosinski et al. [2014]. Average performance of each treatment. AnB^+: transfer and fine-tuning improves generalization. BnB^+: Fine-tuning recovers the co-adapted features. AnB: Performance drops due to task specialization. BnB: Performance drops due to co-adapted features. (Best viewed in colors).

be negatively affected due to the new environment and change in domain. By fine-tuning it in a supervised manner the original performance can be regained. However, this is hard given the state-of-the-art today. Current CNNs may require thousands of labeled examples from each category to be fine-tuned, this is not practical. The motivation behind **Tzeng et al. [2015]** is to develop an algorithm that adapts between the source- and target-environments. This can be done using generic statistics from unlabeled data collected in the target environment as well as some (human) labeled examples from a subset of the categories of interest. Previous efforts have shown that CNNs trained on large datasets are good at reducing dataset bias, but unable to remove it completely. This paper introduces a novel CNN architecture that works on unlabeled or sparsely labeled target domain data. This facilitates transfer between domains and optimizes for domain invariance. The results of this architecture exceed previously published results on standard visual domain adaptation tasks.

One technique used to be able to transfer domain is *domain confusion*. This is done by making the marginal feature distributions of the source and target as similar as possible. A domain classifier is learned to correctly classify each image

into the domain from which it came. After this, it attempts to learn a representation such that the domain classifier cannot distinguish the two domains in feature space.

Domain confusion will reduce the distance of the marginal distributions for the domains, but source and target classes are not yet aligned. The process of aligning source and target classes is called *task transfer* and is done by transferring empirical category correlations learned on the source to the target domain and optimize the representation to recreate the structure in the target domain. The few labeled target examples are used as reference points. First, the *soft label* is computed. This is the average output probability distribution over the source training examples in each category. Second, the model is optimized to match the target labels to the soft labels. The result is that information is transferred to categories with no label in the target domain.

Domain confusion and task transfer is solved with a novel CNN architecture. When a small amount of labeled data is available in each category, supervised adaptation is performed. When a small amount of data is only available in a subset of the categories, semi-supervised adaptation is performed.

To evaluate the efficiency, the method is conducted on the Office dataset, which is a standard benchmark dataset used for visual domain adaptation. In addition, a new large-scale cross-dataset collection for classification across visually distinct domains. Positive results were achieved for *supervised adaptation*, where labeled data was available in all source categories and sparsely available in the target categories. Using only soft labels or domain confusion performed better than hard label training in 5 out of 6 domain transfers. By combining the two, slightly higher performance on average was achieved. Their method outperforms the baselines for *semi-supervised adaptation*, where labeled data was available in all source categories and sparsely available for a subset of the target categories. The baseline for semi-supervised adaptation was a previous adaptation method as well as a CNN trained only on the source.

2.6.4 Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning

Parisotto et al. [2016] builds on the findings of Mnih et al. [2015] and suggests a method to enable multitask- and transfer learning. This makes a network able to learn multiple tasks at the same time and generalize its knowledge to different

and new domains. A part of their motivation is that this cuts back on the time it takes to learn in new environments. They use Atari games to evaluate their method.

Mnih et al. [2015] uses the same hyperparameters for all the games they learn, but their DQN needs to be explicitly trained for each game. Parisotto et al. [2016] suggests that it would be beneficial to exploit the similarities in the different games. An example is the game Pong and Breakout, where both consist of steering a paddle to deflect a ball. They hypothesize that a network trained on many source tasks should be able to generalize between them. Also, if a network is trained on many source tasks, it could be transferred to new target tasks.

Given source games $S_1, ..., S_N$ and expert DQNs $E_1, ...E_N$, where E_i is an expert at playing S_i. The goal is to obtain a single network that is able to play at an expert level on any source game. The paper introduces a method they call *Actor-Mimic* to achieve this. The output of the expert network is used as a supervised training signal to guide a multitask network. The multitask network has the same architecture as an expert network and learns from 8 expert networks simultaneously. This shows that the model does not need to scale linearly with the number of source tasks and indicates some level of generalization between the games.

Transfer learning is achieved by learning a previously unseen game with a trained multitask network as starting point. They show that the pre-trained network converges faster than a randomly initialized network, effectively cutting down the training time and also indicating that the learned representations in the source network generalize well.

The Actor-Mimic method is evaluated across a set of 20 Atari games and reaches *close-to-expert* performance on 7 out of the 8 games trained simultaneously. The Actor-Mimic network uses around 5 million frames to reach a stable behavior. Compared to the expert networks, using up to 50 million frames for training, this is a significant cutback. In the case of transfer learning, transferring between similar games such as Breakout and Video Pinball saves up to 5 million frames of training time. They claim this corresponds to a cutback in training time of several days with the hardware used (Nvidia GTX Titan). However, the cutback in training time is significant only for relatively similar games. The game Robotank seems to learn slower with the transfer from the multitask network. This is probably because it was the only first-person game in the set of evaluated games and not similar to the other games.

2.6.5 Other Relevant Work

Because of memory limitations on the original Atari 2600 console, a maximum number of objects that could be displayed on-screen at the same time was set. Many games omitted this limitation by drawing game objects every other frame, effectively increasing the number of possible objects to use. This introduces a flickering effect. It is not visible and hence not a problem for a human player as it happens too fast to be noticed, but this must be taken into consideration when designing a learning system. Mnih et al. [2015] solved this issue by using the four last frames as input every time step. This provided enough information for the system to understand the context. A different approach to this problem is described by **Hausknecht and Stone [2015]**. They suggest replacing the first post-convolutional fully-connected layer with a recurrent Long Short Term Memory and only using one frame as input for each time step. Their *Deep Recurrent Q-Network* generalize its policies to the case of complete observations when trained with partial observations by introducing *Flickering Pong*. Flickering Pong is a variant of the game Pong where for each time step, the screen is either fully revealed or fully obscured with a probability of $p = 0.5$. They found that in the case of Flickering Pong, the performance scaled with the observability of the domain, reaching almost perfect levels when every screen is observed. They observed no improvement on other standard Atari games and concluded that their recurrent approach only served as an alternative method to the original approach.

A somewhat similar contribution as Parisotto et al. [2016] (see Section 2.6.4) was published by a Google DeepMind team late November 2015. **Rusu et al. [2016]** builds on the findings of Mnih et al. [2015]. They use *policy distillation* to extract and transfer action policies from DQNs. Distillation was earlier introduced as a method for supervised model compression. It works by using supervised regression to train a target network to reproduce the same output as the source network. The paper suggests a method to use distillation in reinforcement learning domains. They tested policy distillation in various processes, distillation to smaller networks was successful and resulted in compression and generalization advantages from the original DQN. This is interesting because small models can be used in many ways, an example is multitask performance. As mentioned in Section 2.3, multitask learning is a method to improve generalization by using some similar tasks as a shared source of inductive bias. This is hard in the Atari domain because the images are diverse and, opposed to natural images, does not share a common statistical basis. However, distillation makes this easier. Finally, they mention that future research should focus on using distillation techniques to stabilize and accelerate learning.

As mentioned in Section 2.3, Caruana [1997] stated that machine learning tech-

niques not using handcrafted features could benefit from multitask learning. An example of this is **Collobert and Weston [2008]**. In the domain of natural language processing, they used a CNN that when given a sentence, outputs several different traits of it. Examples of outputs were labeling words with their syntactic role (noun, adverb,...), synonyms, and if the sentence made sense grammatically and semantically. The interesting part of their work is that by training jointly with shared weights using multitask learning, their network was able to learn these different traits of a sentence at the same time. Earlier approaches only focused on learning one trait at a time. They focused on developing a network that was rather general and effective with very limited prior knowledge. They claim that the reason for why they succeeded was that the CNN learned the features that were relevant and did not rely on handcrafted features. Apart from showing that simultaneous learning improved generalization performance, their method was extremely fast. It could take advantage of huge databases, such as Wikipedia with 631 million words.

2.7 Background Summary

Research from the field of deep learning related to this thesis have been presented. This section will give a summary of this chapter and at the same time view the related work in light of the research questions.

Mnih et al. [2015], as described in Section 2.6.1 is the paper this thesis builds upon. The paper received a lot of publicity and has lead to many studies published by other research teams, building on its findings. This indicates that their contributions have been significant.

Yosinski et al. [2014], summarized in Section 2.6.2, is used in this background theory because their contributions are directly related to the research questions of this thesis and provide a foundation for further research in this area. The question is if the lower layer features in DQNs will have similarities as well.

In terms of the second research question, which investigates ways to achieve a cutback in training time for DNNs, we see that Section 2.6.3 describes a method to reduce the bias a trained network has for the domain it is trained on. This will indeed make it generalize better and reduce the perceived training time for a potential end user of a system since the network is already initialized with general knowledge.

In addition, as we saw in Section 2.6.5 with Collobert and Weston [2008], combining multitask learning with deep networks can provide good results in terms of

generality and the speed of learning. It turns out Parisotto et al. [2016], summarized in Section 2.6.4, successfully combined multitask learning with a DQN. It describes a novel technique to train a network on multiple tasks at the same time and reducing training time drastically. Similarly, Rusu et al. [2016], published by researchers from Google DeepMind used a distillation technique to compress the knowledge contained in a DQN. They stated that accelerating learning should be prioritized.

It seems like research in the area of combining multitask learning with reinforcement learning, and the importance of reducing training time is a priority among research done in this field at the moment. This does indeed confirm the relevance of the research questions of this thesis. However, the task of combining multitask learning with deep reinforcement learning is computationally expensive and requires state of the art hardware. What if it is possible to reduce the training time for a DQN in a less computationally expensive way. Is it possible that DQNs trained on tasks in the domain of Atari games can have knowledge general enough that it can reduce training time on a similar task when transferred?

This is the motivation identified when doing the background research. Can the knowledge contained in a trained DQN be general enough to be transferred to another network and reduce the training time for a similar task? And, is it possible to do this directly without any distillation, multitask learning or other techniques focusing on compressing the knowledge from several trained networks? The next chapter describes the DQN in detail in addition to using the insights provided by the background research to define experiments addressing the research questions.

Chapter 3

Methodology

3.1 Overview

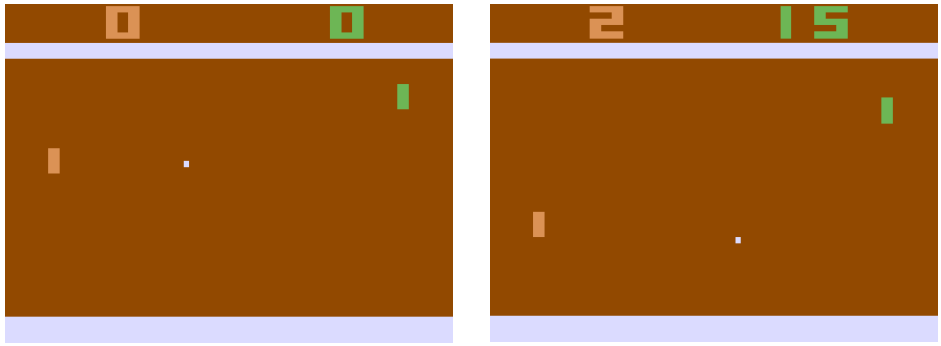
This chapter starts by introducing the domain of Atari 2600 games used in the experiments of this thesis in Section 3.2. Second, the DQN is described in detail in Section 3.3. Furthermore, the hyperparameters and implementation details are mentioned in Section 3.4 and Section 3.5, respectively. Finally, the experiments conducted with the DQN in this thesis are described in Section 3.6.

3.2 Atari 2600

The Atari 2600 video game console was released in 1977 and is by many considered the godfather of modern video game systems. Selling over thirty million consoles and hundreds of millions of games, it helped spawn a multi-billion dollar industry [Ata, 1999].

Two games have been chosen for the experiments in this thesis; *Pong* and *Breakout*. Pong, as seen in Figure 3.1 was one of the first arcade video games and was first released in 1972. It is a sports game that simulates a game of tennis. It was originally a 2-player game in which the players control a paddle each and try to deflect the ball in a way that makes the opponent miss it. By making the opponent concede the ball, the player is rewarded a point. The player that first reaches 21 points wins. The version of Pong used in this thesis includes a simple reflex agent used as the opponent to the Q-network. The reflex agent simply steers the paddle to the same y-coordinates as the ball at all times. It does this with a constant velocity, making it possible for the other player to beat it. The player controlled by the DQN agent needs to learn how to use the walls to be

able to drive the ball past the reflex agent.



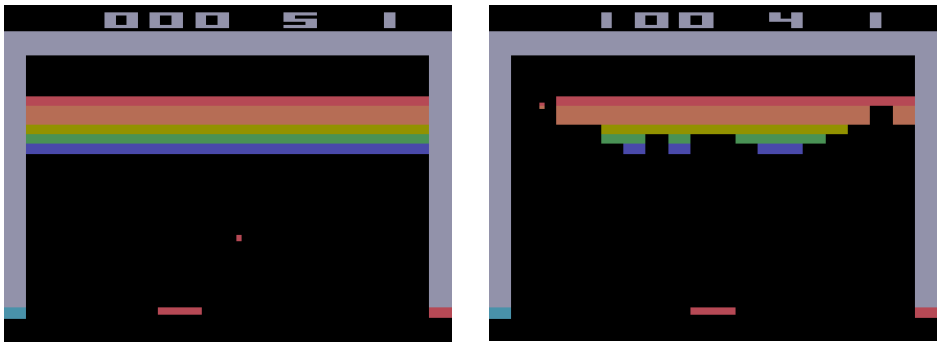
(a) (b)

Figure 3.1: The Atari game Pong. **(a)** Screen from a newly started episode. **(b)** Screen from a little later in the same episode. The left (orange) player is the simple reflex agent built into the game. The right (green) player is the agent controlled by the DQN. The ball, seen in the middle of the screen, is deflected by the players. If a player misses and concedes the ball, the opponent receives a point. The score is displayed at the top of the screen. Best viewed in colors.

Breakout, as seen in Figure 3.2, is the other game selected for this thesis. It was influenced by Pong and released in 1976. The game consists of six layers of blocks at the top of the screen. These blocks can be broken by hitting them with a ball. The player controls a horizontally aligned paddle at the bottom of the screen which can be steered left and right. By using the paddle to deflect the ball and hitting the blocks, the player is rewarded points. Each block in the two bottommost layers is worth one point. The blocks in the two middle layers are worth four points each. In the two top layers, each block is worth seven points. If the player is unable to deflect the ball, letting it exit the screen at the bottom, he or she loses a life.

Throughout this thesis, the term *episode* is used to describe a self-contained series of events in a game. For example, an episode of Pong starts with both players having a score of 0. When one of the players has obtained a score of 21 and thereby winning the game, the episode ends and a new episode can start.

There are some reasons why Pong and Breakout were chosen as tasks for the experiments in this thesis. First, they have similar game mechanics. Both involve steering a paddle and deflecting a ball. At the same time, they have differences



(a) (b)

Figure 3.2: The Atari game Breakout. **(a)** Screen from a newly started episode. **(b)** Screen from a little later in the same episode. The agent controls the paddle down in the middle of the screen. Its task is to steer the paddle horizontally to deflect the ball as it drops down. By hitting the blocks at the top of the screen with the ball, the agent is rewarded points. The numbers at the top display current points, lives left and number of times all the blocks have been cleared (from left to right). Best viewed in colors.

such as the overall alignment of the game play. In Pong, the ball mainly travels horizontally, while in Breakout, the ball mainly travels vertically. Even though they are two different games, they are similar enough that it is fair to consider the possibility that a DQN may acquire knowledge that is applicable to both.

The Atari 2600 was one of the first gaming systems. Hence, there are some reasons why using this platform as domain is suitable to test concepts and theories within reinforcement learning. First, the games themselves are easy to understand and has straightforward mechanics. Second, the fact that most of the games are two-dimensional as opposed to three-dimensional, removes some of the complexity faced by the DQN when learning from pixel values. Third, the games have a resolution of 210x160, making the number of pixels on the screen relatively low. This is, of course, important when considering the time complexity of a system working with each pixel.

3.3 Deep Q Network

This section describes the DQN system used in this thesis. First, the preprocessing step performed on the game frames before using them as input to the network

is described. Second, a description of the network model architecture is given. Last, the algorithm and how the DQN is trained is described.

3.3.1 Preprocessing

Atari frames are 210×160 pixels images with a 128-color palette. This can make the memory and computational requirements challenging. In addition, memory limitations on the original Atari system resulted in a hard limit on how many objects could be displayed at each frame. To be able to increase the number of objects displayed, many games contain objects that are only visible every other frame. This causes an unwanted flickering effect if every frame is used without modification. A preprocessing step is introduced to handle these issues. Flicker-free frames are created using the maximum pixel value of the current frame and the previous frame. Then, the frame is rescaled to an 84×84 gray-scale image. $m = 4$ frames are stacked at a time and used as input to the DQN. The m stacked frames are necessary for the agent to get a sense of how the environment is changing. As an example, it is important for the agent to know which direction the ball is traveling. This preprocessing step is represented as the function ϕ in the algorithm described in Section 3.3.3.

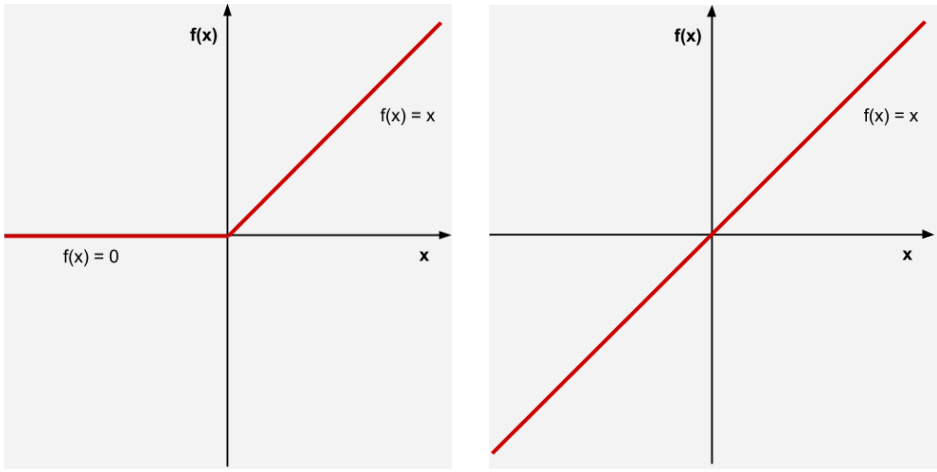
3.3.2 The Model Architecture

The input to the network is the $84 \times 84 \times 4$ array produced by the preprocessing step. This is the state representation. Each convolutional layer first convolves the result from the previous layer, then applies an activation function called the rectifier, defined as

$$f(x) = max(0, x) \tag{3.1}$$

where x is the input to the neuron. A unit applying the rectifier function is often called a *rectified linear unit* (ReLU) and is the most used activation function in neural networks today [Glorot et al., 2011]. See Figure 3.3a.

The first hidden convolutional layer has 32 8×8 kernels and a 4-pixel stride. The stride is simply how many pixels at a time the filter is moved during each step in the convolution operation. A stride of 4 results in the filter moving in 4-pixel increments over the convolved image. The second hidden convolutional layer has 64 4×4 kernels and a 2-pixel stride. The third hidden convolutional layer will have 64 3×3 kernels with a 1-pixel stride. The final hidden layer is a fully-connected layer consisting of 512 ReLU units. The output layer is a fully-connected layer with 3 - 18 linear units, one unit for each possible action



(a) (b)

Figure 3.3: **(a)** The rectifier activation function used by most layers in the DQN. **(b)** The linear activation function used in the output layer of the DQN. Best viewed in colors.

the agent can choose (see Figure 3.4). Each output corresponds to the predicted Q-value of a specific action. This is the only layer that changes depending on what game is learned. Some games have many legal actions, while others have few. Both Pong and Breakout has three valid actions: moving the paddle to each side (right/left or up/down) and *no-action* which is doing nothing. A linear activation function does not alter the output from the unit. This makes sense in the case of the output layer since its outputs are the predicted Q-values. Figure 3.3b depicts a linear activation function.

Section 2.2 explains how pooling is often used between convolutional layers in CNNs. Notice that there is no pooling operation between the layers in the DQN architecture. The reason for this is that when including a pooling operation, the learned features becomes spatially invariant and the network becomes insensitive to the location of an object in the image. This is beneficial when the location of the feature is not important for the task, such as detecting a face in a face-detection system. However, in Atari games, the location of features is an important piece of information. For instance, knowing the location of the ball in a game of Breakout is crucial in determining the potential reward.

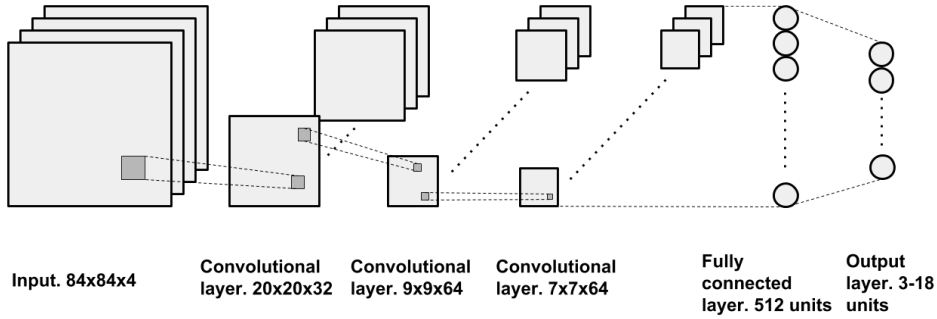


Figure 3.4: The DQN model architecture. The numbers under each layer represent the size of the input frames (feature map) after each convolution step.

3.3.3 The Algorithm

The task of the DQN is to train an agent that interacts with the environment. For every time step it chooses an action a_t from the set of legal actions $A = \{1, ..., K\}$. The action is sent to the emulator which modifies its internal state. The only thing visible to the agent is the stack of images from the emulator representing the game screen as vectors of pixel values. The agent also receives the change in game score as a reward signal r_t for each time step. Although a reward is received every time step, positive rewards are often a result of long sequences of actions and observations. By only observing the current stack of screen images, the task is partially observable. This is because it is not possible to understand the current situation and the potential rewards received solely based on the current stack of frames, that is the observation x_t. Because of this the input to the algorithm is a sequence of actions, a_t, and observations, each being a distinct state $s_t = x_1, a_1, ...a_{t-1}, x_t$. This makes it possible to view it as a Markov Decision Process (MDP) and reinforcement learning methods for MDPs can be applied. An MDP relies on the Markov assumption, that the probability of the next state, s_{t+1}, is only dependent on the current state, s_t, and not any of the preceding states.

As mentioned in Section 2.4, the goal of a reinforcement learning agent is to maximize its future reward. Consequently, it is advantageous to know the total future reward. Since the environment is stochastic, there is no guarantee that a reward received by doing an action now will lead to the same reward in the future. This is the reason the *discounted future reward* is used in the DQN, emphasizing immediate rewards more than future rewards:

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \tag{3.2}$$

where t is a given time step, t' is the next time step, T is the time step where the game episode ends and γ is the discount factor.

The optimal Q-learning function $Q^*(s, a)$ is defined as the maximum discounted future reward achievable by following a policy, π, after observing a state, s, and then taking some action a:

$$Q^*(s_t, a_t) = max_\pi E[R_t | s_t, a_t, \pi] \tag{3.3}$$

in which the policy, π, maps sequences to actions. The best policy will be to pick the action with the highest Q-value. The optimal Q-learning function obeys the Bellman equation, explained in Section 2.4. As long as the agent chooses the optimal action, the utility of a state is the immediate reward plus the expected discounted utility for the next state. Or in terms of the Q-function:

$$Q^*(s, a) = r + \gamma max_{a'} Q^*(s', a') \tag{3.4}$$

where the Q-value for the current state, s, and action, a, is given by the immediate reward, r, plus the maximum future reward for the next state, s', and action, a' discounted by γ.

The DQN is introduced as a non-linear function approximator to estimate the Q-learning function $Q(s, a; \theta) \approx Q^*(s, a)$, where θ are the adjustable weights in the network being optimized. The network is trained to reduce the mean-squared error in the Bellman equation. The optimal target values from Equation 3.4 are replaced with the approximate target values:

$$y = r + \gamma max_{a'} Q(s', a'; \theta_i^-) \tag{3.5}$$

where θ_i^- are the model parameters from some previous iteration. In supervised learning the targets, or labels, are fixed before learning. Here, as we can see, the targets, y, are dependent on the network weights, θ. Using the targets, y, and the predicted Q-value, $Q(s, a; \theta_i)$, we can define regression tasks optimized with the *squared error loss*:

$$L_i(\theta_i) = \frac{1}{2}[y - Q(s, a; \theta_i)]^2 \tag{3.6}$$

for each iteration, i. This loss function defines the error surface in which a variation of SGD called *RMSProp* is used to tune the weights of the DQN iteratively. RMSProp is explained further in Section 3.3.4.

Using a CNN to approximate the Q-function is not a very stable approach by itself. Thus, Mnih et al. [2015] introduces several techniques to improve the stability of the DQN. One of these techniques is the *replay memory*. It works by storing the experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ of the agent at each time step t in a dataset $D_t = \{e_1, ..., e_n\}$. Q-learning updates are applied to these samples of experience, $(s, a, r, s') \sim U(D)$. These are drawn at random from the pool of stored samples in the replay memory D_t. The size of the replay memory starts out at 50,000 experiences and is constantly filled up. Over time, the 1 million most recent experiences are kept in the memory.

The introduction of a replay memory has several advantages. First, previous experiences are reused making the data usage more efficient. Second, there are most likely correlations between consecutive experiences. By randomly selecting samples from the replay memory the correlations will vanish. From Equation 3.5 we can see that the targets are dependent on the weights of the DQN. It is not hard to imagine that by using subsequent training samples of experience, the behavior of the agent will be biased. As a result, when averaging the behavior distribution over many of the previously seen states, the algorithm is prevented from getting stuck in a poor local minimum.

Another technique used to improve the stability of the DQN is the employment of a *target network*. This is a separate network used to generate the targets in the Q-learning update. It works by cloning the DQN to a new network, $\hat{Q}$, every C updates. $\hat{Q}$ is used to generate targets for the Q-learning update the next C updates. Since targets are dependent on the network weights, using older parameters to generate the targets will effectively create a delay between the Q-update and when that update affects the targets. This makes oscillations and divergence of the algorithm less likely.

In addition, Mnih et al. [2015] found that clipping the difference between the generated target and the predicted output of the network to be between [-1, 1] improved the stability of the algorithm. This keeps the gradient constant when the loss is outside the clipping range. By doing this, the high score variation between different games has less impact on how the DQN learns. For instance, in Breakout, the score can be between 0 and several hundred while in Pong, the score is always between -21 and 21.

Algorithm 1 Deep Q-learning with experience replay, Mnih et al. [2015]

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
for episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = argmax_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_i = \begin{cases} r_j & \text{if epsiode terminates at step j+1} \\ r_j + \gamma max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform gradient descent step on $(y_i - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  end for
end for

3.3.4 Training

A DQN is trained for each game. The hyperparameters, model architecture and learning algorithm are all kept the same on all games trained. The exception is the output layer in the model architecture. As mentioned in Section 3.3.2, there is one output neuron for each valid action the agent can choose. The games themselves are not altered except for clipping the reward coming from the Atari emulator to be between 1 and -1. By clipping the reward, it is possible to keep the learning rate fixed between the different games. However, the downside of clipping the reward is that the agent is unable to separate different magnitudes of the score. In some games, the number of lives is important. This information is also sent from the Atari emulator and is used to mark the end of a game episode.

A variation of SGD (described in Section 2.2) called RMSProp [Hinton et al., 2012] approximates the optimal gradient used to adjust the weights. When using SGD, each weight is updated at its own pace due to the large variation in the magnitudes of the various gradients. In addition, it can also change during learning. This makes it hard to determine a global learning rate. RMSProp solves this by dividing the gradient by a running average of the magnitude of the gradient. This is used as the momentum described in Section 2.2 where earlier gradients affect the current gradient. Intuitively, minimizing the loss with momentum can be seen as rolling a ball down the error landscape defined by the loss function. The ball gains momentum as it rolls downhill and variations in the gradients has less influence on its velocity. This reduces oscillations in the updates of the weights, effectively speeding up the learning. A ρ-decay determines the decay factor of the moving average tracking the gradients which are used to find the momentum. Furthermore, a minibatch size of 32 is used, effectively using 32 training examples to estimate each gradient.

The Q-learning behavior policy is the ϵ-greedy policy mentioned in Section 2.4 where the agent chooses a greedy action with a probability of $1 - \epsilon$ and a random action otherwise. This ensures a desirable divide between exploring and exploiting. Exploring new states can potentially result in the discovery of actions leading to higher rewards, while exploiting the current best action is necessary to collect the current best-known reward. The ϵ value starts at 1.0 and is reduced to 0.1 over the first 1 million frames where it is kept fixed for the rest of the training. The agent is trained for a total of 25 million frames. This is in contrast to Mnih et al. [2015] where the agent is trained for 50 million frames. The reason for this is limitations in available computing power, elaborated in Section 3.5.

The replay memory dataset has a maximum size of 1 million experiences and a start size of 50,000 experiences. Before the training starts, the replay memory

is populated with experiences generated from random actions until reaching the start size. The DQN employs a frame skipping technique based on repeating the previously found action $k = 4$ times. The network returns an action based on the observation, or stack of frames it receives as input. Limiting the frequency in which the network receives observations and simply repeating the previously found action k times has some advantages. First, no game is designed to require a new action input for every frame, so the introduction of frame skipping seems like a reasonable choice. Second, it is less computationally expensive to have the Atari emulator repeat the last action than to propagate a new observation input through the DQN and wait for a new action to be returned. This helps making the DQN feasible to run on current available hardware.

After every training epoch, lasting 250,000 steps, a testing epoch of 125,000 steps is conducted. This testing epoch has no effect on training and is there simply to give feedback on how the training is progressing. During testing, the ϵ value is kept fixed at 0.05. The reason for this is that the Atari emulator does not have a random generator. Consequently, the randomness involved in many games needs to be introduced externally. This is done through the player input. This works for human players, as they are not accurate enough to be able to exploit or even notice these deterministic subtleties. However, an agent trained by the DQN may be able to both detect and exploit it, resulting in unwanted behavior. Giving the agent a 5 % chance of performing a random action, introduces the needed randomness and effectively removes the possibility of the agent exploiting the Atari emulator.

3.4 Hyperparameters

When training DNNs there are a lot of hyperparameters involved. There are even more hyperparameters to consider when combining DNNs with reinforcement learning. The hyperparameters, their values and descriptions are presented in Table 3.1. They are similar to those used by Mnih et al. [2015] and are held constant for both games and all tests performed in the experiments. Most of the hyperparameters are mentioned and explained in Section 3.3.

An important hyperparameter that should be mentioned is the *update frequency*. It is included to limit the weight update frequency. Updating the network weights at every action step chosen by the network, results in a higher computational cost with no apparent performance gain in terms of learning. The RMSProp, therefore, updates the network weights only at every fourth action step while learning.

This results in a lower computational cost and has almost no impact on the performance.

Furthermore, the *no-op max* hyperparameter determines how many random actions the agent should perform at the start of each game episode. There are two reasons for this, the first being that the DQN is certain that the screen buffer for the Atari emulator is ready. Second, the initial game screen should be randomized. Recall that the Atari emulator is dependent on external input to generate randomness. Doing a number of random actions before the game episode starts should ensure that the agent is unable to take advantage of the deterministic behavior of the Atari emulator.

3.5 Implementation Details

Several frameworks and libraries exist that can make the design and implementation of DNNs easier and training them faster. Some of the most popular frameworks are Torch, Theano, PyLearn, TensorFlow and Caffe. A comparison between these technologies is beyond the scope of this thesis. However, it is worth mentioning that they all allow the user to implement high-level code that is executed on a graphics processing unit (GPU).

The work done in this thesis uses Theano [Bergstra et al., 2010] and [Bastien et al., 2012]. Theano is a Python library and optimizing compiler for mathematical expressions using NumPy [Ascher et al., 2001]. NumPy is a Python package for scientific computation. Theano supports C/C++ implementations of tensors, or n-dimensional arrays, and other functionality important to implement a deep learning system. Theano uses *g++* or *nvcc* to compile parts of the code to CPU- or GPU instructions which are more efficient than pure python.

The implementation of the DQN in this thesis is based on Sprague [2015], which is a Python implementation aimed at recreating the system of Mnih et al. [2015] using Theano. A Theano-based Python library called Lasagne [Battenberg et al., 2015] is used to easily create the CNN model. To emulate Atari games, the Arcade Learning Environment [Bellemare et al., 2013] is used. It is a framework built on top of an Atari 2600 emulator and is specifically designed to develop AI agents. The framework makes it easy to interact programmatically with the emulator.

The system was run on a desktop computer with an Nvidia GTX 570 GPU with 1 GB of video memory. Training an agent to play a game took between 120 and 150 hours depending on the game. Optimizations of this code were done by

Table 3.1: Hyperparameters for the DQN.

Hyperparameter	**Value**	**Description**
Epochs	100	Number of epochs trained.
Training steps per epoch	250000	Number of training steps in one epoch.
Testing steps per epoch	125000	Number of testing steps in on epoch.
Minibatch size	32	How many training examples RMSProp uses to compute the gradient.
Replay memory size, N	1000000	The maximum number of experiences stored in the replay memory to randomly sample from.
Replay memory start size	50000	Initial size of the replay memory. Before training starts, the replay memory is populated with experiences resulting from random actions.
Target network update frequency, C	10000	Number of training steps between each time $\hat{Q}$ is reset.
Learning rate	0.00025	Used by RMSProp to scale how fast the agent should learn.
ρ-decay	0.95	Decay factor of gradient moving average in RMSProp. Used to track the history of the gradient and the squared gradient.
Minimum squared gradient	0.01	Small value added in RMSProp for numerical stability.
Clip limit	1.0	Where the loss should be clipped. 1.0 indicates clipping values outside [-1, 1] to corresponding limits, keeping the gradient constant.
Discount factor, γ	0.99	The factor of which future rewards are discounted by in the Q-learning update.
Exploration probability, ϵ	1.0 - 0.1	ϵ-greedy exploration starts at 1.0 and decreases over 1000000 steps to 0.1 where it is kept static.
Frame skip, k	4	How many frames the current action is repeated by the agent.
Agent history length, m	4	How many frames is stacked to create the input to the Q-network.
Update frequency	4	Number of actions selected by the agent between each update from the RMSProp. The agent will choose 4 actions before weights are updated.
No-op max	30	Number of random actions at the start of each episode. This is done to ensure the screen buffer is ready and to randomize the initial game state.

eliminating unnecessary CPU - GPU cycles. In addition, redundant data structures that were draining memory was removed. The alterations were done while still maintaining the core principles of the system described by Mnih et al. [2015]. This brought the run time down to between 70 and 90 hours for a single training run. This was deemed feasible for a system of this complexity executed on the specified hardware. In addition to the optimizations, several scripts were implemented to aid in conducting the experiments in this thesis. These include being able to transfer layers between two trained networks, visualization of weights in specific layers of a trained network, comparison of the weights in specific layers between two trained networks as well as plotting score, expected reward and loss signal. Appendix B and Appendix C describes the code briefly, including where it is available.

Due to the somewhat dated GPU used in this thesis, Amazon Web Services was also considered to do the training, as hey provide cloud-based GPU instances. However, at an hourly price of around $0.65 per hour and considering the total hours required by the experiments, this price was too high.

3.6 Experiments

The DQN system described in this chapter was used to address the research questions. In Section 2.7, the current state of the field is summarized. Based on the insights provided by the background research and the research questions, three experiments have been designed. This section describes the experimental setup for all experiments. In addition, it will specify how the results are presented and what the expected outcome of the experiments are.

The research questions, first mentioned in Section 1.3 are restated here for simplicity:

Research question 1 *Will there be any similarities between the lower level features in DNNs trained on similar tasks?*

Research question 2 *Will there be a significant cutback on the training time for a network already trained on a similar task? How significant?*

Table 3.2 provides an overview of the experiments conducted in this thesis. Each experiment was given a unique ID to clearly separate them. Experiment E1 is related to research question 1 and consists of two parts. First, a visual comparison between the low-level features in DQNs trained on similar tasks was performed. Second, a method to compute a score describing the difference between the first layers of two trained networks was executed. Experiment E2 and E3 are almost

identical. The difference being what task is used as target network. The various experiments will be described in the following subsections.

Table 3.2: Experiment overview.

ID	RQ	Description
E1	1	Visually compare the first convolutional layers of networks trained on similar tasks. In addition, compute and compare the difference score between the first convolutional layers of networks trained on similar tasks.
E2	2	Cutback in training time, Pong as target task.
E3	2	Cutback in training time, Breakout as target task.

3.6.1 Selection of Tasks

Two games were chosen as tasks for the experiments performed with the DQN in this thesis. The research questions clearly state that the tasks should be similar. In addition, the background research also confirmed the importance of similarity between tasks. Since the DQN consists of a CNN, the visual similarity was heavily emphasized when considering different games to use as tasks. In addition, the gameplay and game mechanics were evaluated. This includes what the goal of the player is and how that goal is achieved.

As described in Section 3.2, Pong and Breakout were the games chosen as tasks for the experiments in this thesis. They are somewhat visually similar, hopefully similar enough that the DQN will be able to generalize its learning. However, there are some visual differences. Pong has less critical information on the screen, only two paddles and a ball. Breakout has several layers of breakable blocks in addition to a paddle and a ball. In terms of gameplay and game mechanics, there are more similarities. The goal of both games is to prevent a ball from moving past the paddle in which the player controls. More specifically, in Pong, the goal of the game is for each player to make the opponent concede the ball. Whereas in Breakout, the goal of the game is to break the blocks while not conceding the

ball. The two games are similar enough that it is fair to hypothesize whether the DQN may be able to learn some features that can be used in both games.

3.6.2 E1 - Low-Level Features - Visual Comparison

This subsection describes the setup of the first part of Experiment E1, which makes a visual comparison between the first convolutional layer in two networks. A DQN is trained for each task, Pong and Breakout, to investigate if there are any similarities between the weights learned in the first convolutional layer of the DQNs. A comparison between the two networks is done by visualizing the first convolutional layers and comparing them manually. A layer consists of 32×4 filters, each being 8×8 pixels each. If the knowledge contained in these weights is of a general nature, they should correspond to image processing filters like edge detectors. However, if the weights have evolved as more specific feature detectors for the task it is trained on, they may be less similar and of a less general nature.

The expected results from this experiment are that the learned features will be somewhat general. Hopefully, filters resembling edge detectors can be detected. In addition, the focus will be to detect filters appearing similar across the tasks being learned. For example, the ball is a game object that both Pong and Breakout have in common. Investigating the filter visualizations should reveal feature detectors that can detect this ball.

The result will be visualized as an array of 32×4 filters, as this is the number of filters in the first convolutional layer of the DQN. Each filter is 8×8 pixels, where each pixel has a gray value between 0 and 1. A pixel value of 0 is black, whereas a value of 1 is white.

3.6.3 E1 - Low-Level Features - Difference Score

This subsection describes the setup for the second part of Experiment E1. In addition to visually comparing the first convolutional layers, the difference between them is also quantified in this part of E1.

A quantification method compares two DQNs: DQN_A and DQN_B. Only the first convolutional layer is considered and consists of 32×4 filters, each 8×8 pixels large. The weights, or pixels, in each filter are first normalized to make the comparison easier and more accurate. The average of the pixel-wise difference between each filter in DQN_A and DQN_B results in a number describing the difference between two arbitrary filters, DQN_{Am} and DQN_{Bn}, in the first layer

of two different networks:

$$DQN_{Am,Bn} = \frac{\sum_{i=0}^{p} DQN_{Am,i} - DQN_{Bn,i}}{p} \tag{3.7}$$

in which the difference between each pixel, i, in the m'th filter in DQN_A and n'th filter in DQN_B is averaged into a number describing the difference between those two filters, $DQN_{Am,Bn}$. p is the number of pixels in a filter and in the first convolutional layer, $p = 64$.

This results in $32 \times 4 = 128$ comparisons for each filter in DQN_A, all stored in its own collection $DQN_{Am,B}$. Next, each filter from DQN_A locates the comparison in $DQN_{Am,B}$ with the lowest difference, i.e the most similar filter from DQN_B:

$$DQN_{Am,Bmin} = min(DQN_{Am,B}) \tag{3.8}$$

This is done for all m filters in DQN_A and the results are stored in another collection, D_{Amin}. D_{Amin} now contains one difference number for each filter in DQN_A. Each difference number is the lowest difference found in DQN_B for the m'th filter in DQN_A. Finally, this collection of most similar filters is averaged and comprise the *difference score* between the two layers:

$$D_{score} = \frac{\sum_{i=0}^{N} D_{Amin,i}}{N} \tag{3.9}$$

where $N = 128$ and is the number of filters in D_{Amin}.

The reason the difference score, D_{score}, is computed this way is because the filters evolve differently depending on their random initial values. If both DQN_A and DQN_B are trained on Breakout, similar filters may appear in both, but the actual location within the layer is arbitrary. By comparing them this way, the location of the filters is not important. However, the problem with this approach is that several filters from DQN_A may select the same filter from DQN_B, thus not utilizing all the filters from DQN_B in the comparison. This is prevented by doing the comparison both ways. The assumption is that this two-way comparison will produce difference scores that will be both very similar, as well as reasonable to use to quantify the difference between layers.

Table 3.3 describes how the different trained DQNs will be compared. The leftmost column contains the trained networks being used as DQN_A, while the first, top row contains the trained networks being used as DQN_B. Cell numbers are used to indicate corresponding comparisons. For instance, the cells containing the number 4 are the comparisons between the Pong network and the Breakout network. First, Pong is used as DQN_A in row 2. Then, Breakout is used as DQN_A in row 3.

Table 3.3: Comparison scheme for different trained models. Cell numbers indicate corresponding comparisons.

$\mathbf{DQN_A} \downarrow \mathbf{DQN_B} \rightarrow$	**Pong**	**Breakout**	**Random**
Pong	1	4	5
Breakout	4	2	6
Random	5	6	3

The expectation is that when comparing different networks trained on equal tasks, like Pong with Pong or Breakout with Breakout, the D_{score} should be relatively low. Furthermore, the D_{score} between a Pong and a Breakout network should be somewhat higher, although it should be lower than the D_{score} between a Pong and a randomly initialized untrained network or Breakout and a randomly initialized untrained network. This will indicate whether the filters have knowledge applicable to both tasks.

The comparison of the D_{score} values will be performed with two different Pong networks, two different Breakout networks and two different randomly initialized untrained networks. Two networks trained for each task is used to be able to compare the same task with itself. In addition, it adds reliability to the rest of the numbers. The results from the difference score comparison will be presented in a table similar to Table 3.3.

3.6.4 E2 and E3 - Cutback in Training Time

As the setup for Experiment E2 and E3 are almost identical, this subsection describes both. A DQN is trained on each task, Pong and Breakout. These networks are trained from randomly initialized weights, and constitute the *base networks*. The first convolutional layer in the trained base networks is transferred to a *target network*, which is an untrained randomly initialized network. Next, the target network is trained on the task different to the task used to train the base network.

Experiment E2 and E3 are very similar. In E2, Pong is used as target task, employing the transferred first convolutional layer from a Breakout base network. In E3, Breakout is used as target task, employing the transferred first convolutional layer from a Pong base network.

As mentioned, only the first convolutional layer is transferred from the base network to the target network. The reason for this is simple. Intuitively, the lower level features are more general and less specific to the task being learned; they can more easily be applied to a wider range of tasks with minimal adjustments. When transferring this general first layer, it may affect the training time as it has already been trained to fit a similar task. From the related work in Section 2.6.2, Yosinski et al. [2014] claims that by transferring all the convolutional layers, the target network will generalize better on images. However, he does not consider training time when transferring the different layers. Most likely, the higher layers will contain knowledge more closely related to the task being learned. It can, therefore, be argued that these high-level features may contain knowledge so specific that they need to be "unlearned" before being able to learn anything new. If this is the case, it may be worse in terms of training time to transfer all the convolutional layers. Of course, this is just a hypothesis that needs to be tested. Due to the limitations in hardware, this is not in the scope of this thesis, but is the reason why only the first layer was transferred.

Figure 3.5 depicts the transferring process in this experiment. For example, Pong and Breakout are the two tasks that will be learned. In this example, the tasks are termed task A and task B. First, a DQN is trained on task A which results in a base network. Second, the first convolutional layer is transferred from the base network to a new untrained randomly initialized network, the target network. The layer being transferred replaces the randomly initialized first layer of the target network. Then, the target network is trained on task B. There are no differences regarding the way a target network is trained. All hyperparameters are kept the same and all layers, including the transferred layer are allowed to be adjusted. The experiment is performed by transferring a task A base network to a task B target network and a task B base network to a task A target network.

The performance of a trained network is evaluated based on three numbers. First, the average score per epoch. This is the game score the agent achieves and is recorded during the testing epochs which happen after every training epoch (see Section 3.3.4). Second, the average action value per epoch, which is the expected reward for the agent and is also gathered during the testing epochs. Finally, the moving average of the mean training loss. As mentioned in Section 3.3.3, the loss

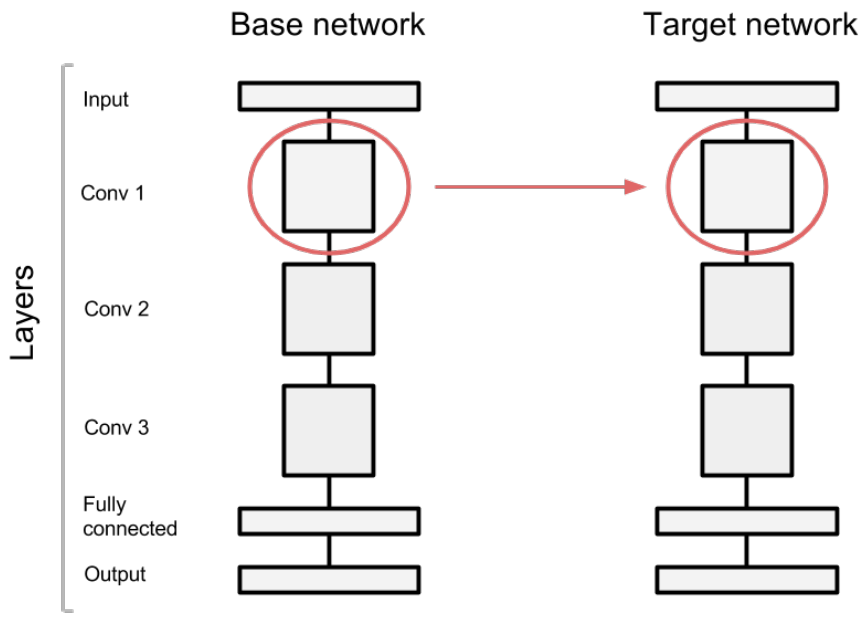


Figure 3.5: The first convolutional layer of the fully trained base network are used as initial starting weights in the target network before it starts training.

is what the network uses for the backpropagation and adjustment of the weights. The loss is gathered for each game episode during the training epochs.

By comparing the result of a task A base network and a task A target network, it is possible to identify if the transferred layer has any effect on the training time for the target network. If the target network converges at an earlier point during training, this may indicate that it needs less training time to achieve the same result as the base network. This comparison is also done with a task B base network and a task B target network. To mitigate the statistical anomalies that may lead to poor results, each experiment is done three times and the data averaged. Ideally, more experiments should be performed, but the limitations in available hardware prevented this.

The expected results from these experiments are an earlier convergence of the target network compared to the base network trained on the same task. The most important metrics are probably the average score and the average action value. They are more closely related to the performance of the agent than the

loss metric is. If both the average score and the average action value climbs faster and converges earlier for the target networks, it is certainly a good sign in terms of a cutback in training time.

3.7 Methodology Summary

In this chapter we have seen an in-depth description of the DQN and the hyper-parameter configuration. Further, the selection of tasks and experimental setup were defined. The next chapter will present, analyze and discuss the results from the experiments.

Chapter 4

Results and Analysis

4.1 Overview

This chapter presents, analyzes and discusses the results from the three experiments defined in Chapter 3. Section 4.2 presents the visual comparison part of Experiment E1. Second, the difference score from Experiment E1 is presented in Section 4.3. Section 4.4 presents the results from Experiment E2, using Pong as target network when investigating cutback in training time. The results from the last Experiment, E3, are presented in Section 4.5 using Breakout as target network when investigating cutback in training time. Then, Section 4.6 states how well the agent in this thesis performed compared to Mnih et al. [2015]. Finally, Section 4.7 will do a brief summary of the results, showing some general trends.

4.2 E1 - Low-Level Features - Visual Comparison

This section first presents the results from part one of Experiment E1 regarding the visual comparison done to identify similarities between the low-level features in networks trained on similar tasks. Finally, the results are discussed and analyzed.

4.2.1 Results

The visualized filters of the first layers in DQNs trained on Pong and Breakout are illustrated in Figure 4.1 and Figure 4.2. Each figure depicts the first convolutional layer of three different DQNs. A layer consists of 32 stacks of filters, each containing 4 filters. A filter stack is used in the convolution process on the corresponding stack of input frames to the network. Figure 4.1a and Figure 4.1b

are both trained on Pong, while Figure 4.1c is a randomly initialized untrained network. Figure 4.2a and Figure 4.2b are both trained on Breakout, while Figure 4.2c is a randomly initialized untrained network. Furthermore, Figure 4.1b and Figure 4.2a are the result of training Figure 4.1c to play Pong and Breakout, respectively. Finally, Figure 4.1a and 4.2b is the result of training Figure 4.2c to play Pong and Breakout, respectively.

Each pixel in the filters is an adjustable weight, normalized to make the visualization and comparison easier. The normalized weights have a value between 0 and 1, 0 being black and 1 being white. The values in between represent different gray values.

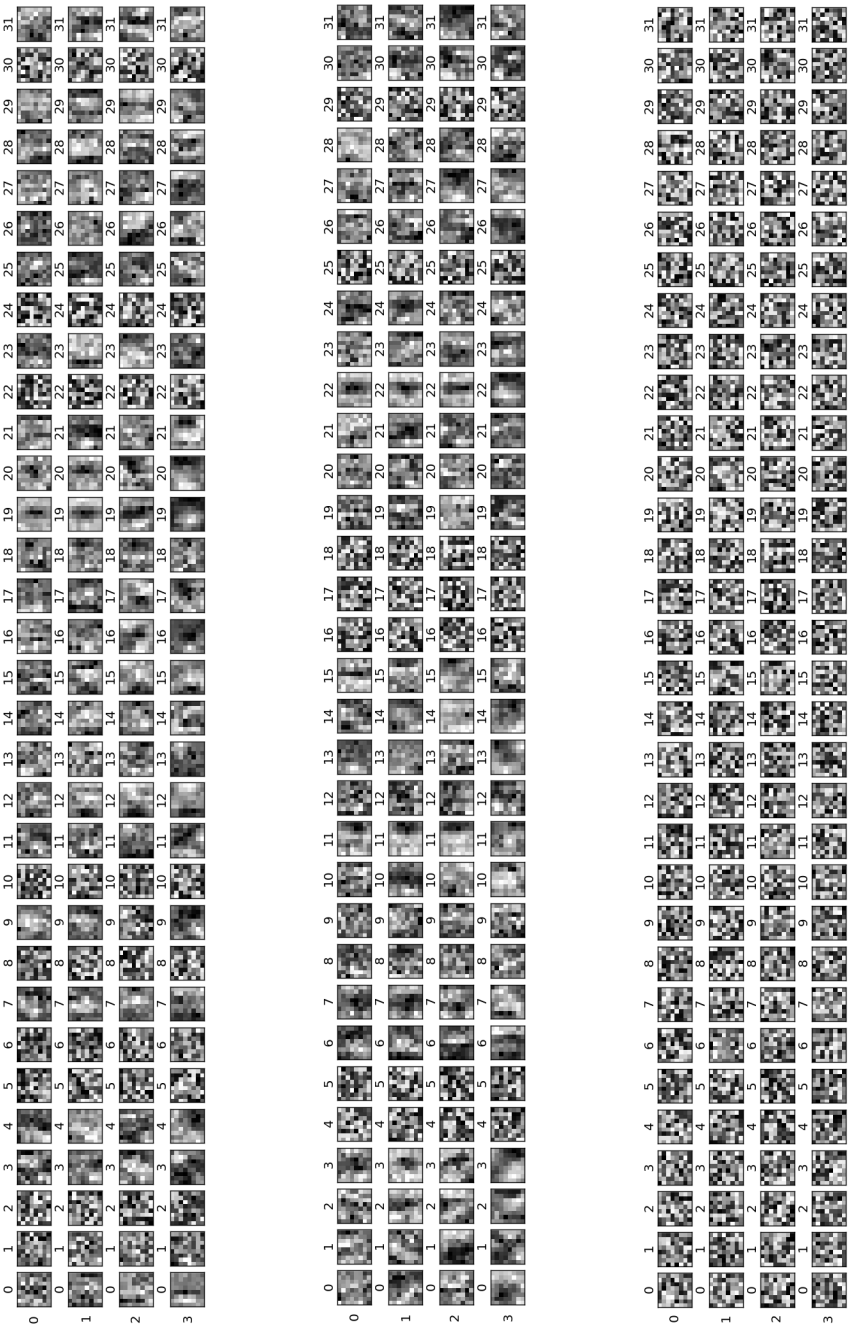
4.2.2 Analysis and Discussion

The first thing that comes to mind when looking at Figure 4.1 and Figure 4.2 is the overall alignment of the filters. The filters from the Pong networks in Figure 4.1, seems to be more vertically aligned while the filters from the Breakout networks in Figure 4.2, seems to be more horizontally aligned. The overall alignment of the pixels suggests that some of the filters give a high response to horizontal and vertical objects within the games. In the game of Pong, there are two vertically aligned paddles in addition to the ball that the agent is required to detect. In Breakout, both the paddle controlled by the agent and the breakable blocks at the top are horizontally aligned.

In CNNs trained to recognize objects in images, the first convolutional layer often corresponds to low-level image processing filters, like edge detectors. Typically, filters that are able to detect edges have one side filled with high pixel values, the middle filled with low pixel values and the other side filled with high pixel values again. By sliding this type of filter over an image, sudden changes in the texture are detected, like edges. The Gabor filter is a well-known edge detection filter.

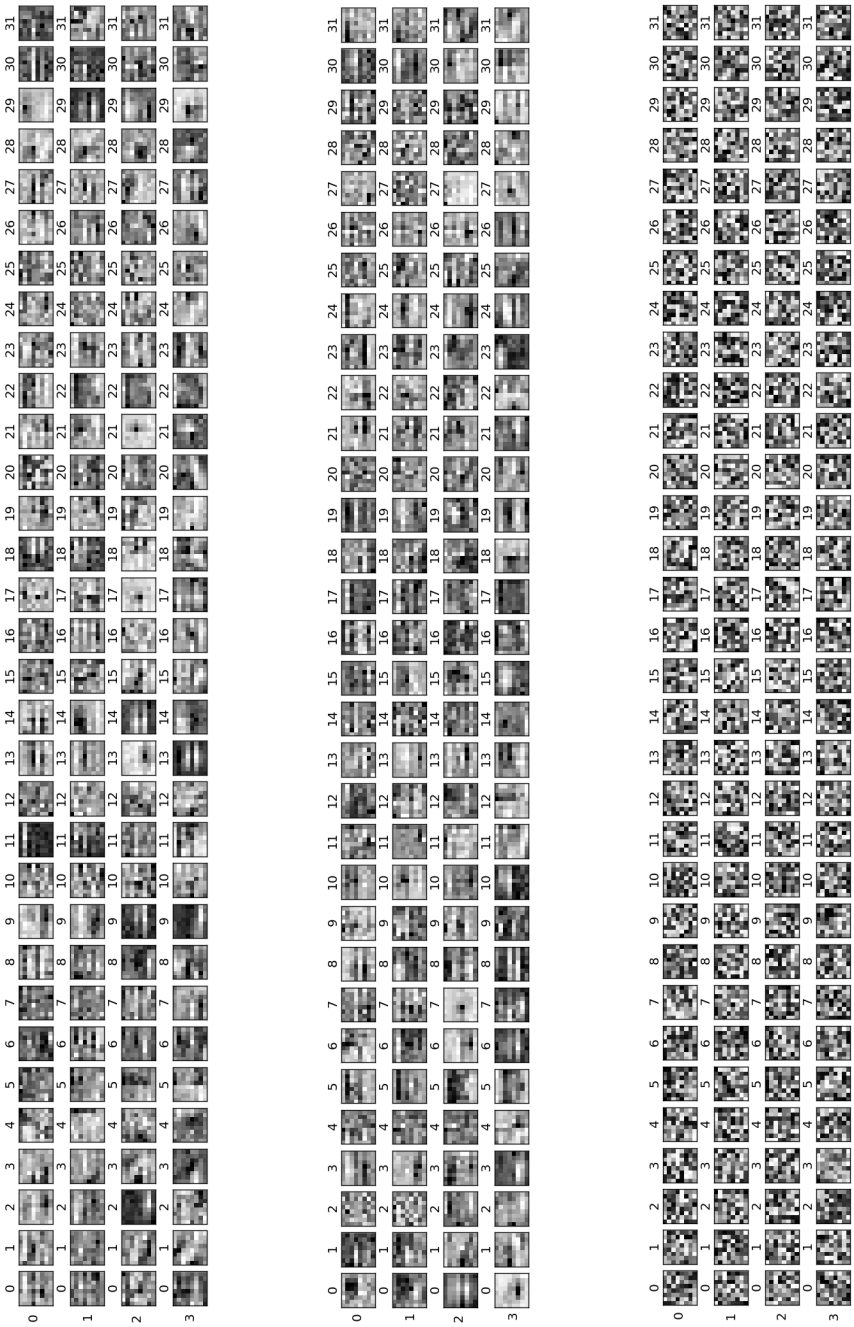
Observing the Pong filters, there may be some examples of vertical edge detectors. However, they are not very apparent. For instance, the top three filters in filter stack 19 in Figure 4.1a may be able to detect some vertical edges. Also, filter stack 22 in Figure 4.1b.

Despite having some filters being able to detect edges in Pong, they seem to be slightly specific to the edges found in the game. The Pong filters mentioned above seems like they would give a high response on the paddles controlled by the players. This is an example of situation specific filters. In addition, the fourth



(a) Pong_1 (b) Pong_2 (c) Random_1

Figure 4.1: The figure depicts the first layer from three different DQNs. Each layer consist of 32 stacks of filters. One stack consists of four filters which is used on the corresponding stack of input frames to the network. **(a)** The first layer in a trained Pong network. **(b)** The first layer in a different trained Pong network. **(c)** A randomly initialized untrained network. (b) is the result of training (c) to play Pong. (a) is the result of training Figure 4.2c to play Pong.



(a) Breakout$_1$ (b) Breakout$_2$ (c) Random$_2$

Figure 4.2: The figure depicts the first layer from three different DQNs. Each layer consist of 32 stacks of filters. One stack consists of four filters which is used on the corresponding stack of input frames to the network. **(a)** The first layer in a trained Breakout network. **(b)** The first layer in a different trained Breakout network. **(c)** A randomly initialized untrained network. (b) is the result of training (c) to play Breakout. (a) is the result of training Figure 4.1c to play Breakout

filter in filter stack 2 and 3 in Figure 4.1b both seems to give a high response when the ball is closing in on the paddle from left to right, the light pixels indicating the ball. Furthermore, the fourth filter in filter stack 12 in Figure 4.1a seems to give a high response when the ball is closing in on the paddle from right to left. These are two situations important for the agent to detect and recognize. Obviously, it is beneficial to know if the ball is approaching a paddle.

Investigating the Breakout filters, there are indeed examples of edge detectors. The filter in the fourth row in filter stack 13 in Figure 4.2a resembles a horizontal Gabor filter. This also applies to the second image in filter stack 29 in the same figure. Breakout_2 in Figure 4.2b has a horizontal Gabor filter in the fourth row in filter stack 24 as well. These filters resemble feature detectors for the horizontal paddle that the agent controls.

Although there are some examples of somewhat general edge detection filters i Breakout, there are indications that many of the filters are trained to give a high response on specific situations occurring during a game. For instance, the fourth filter in filter stack 0 in Figure 4.2b may detect holes, or openings in the layer of blocks at the top. There are several filters in both Breakout figures that work the same way, consisting of mostly light pixels with a darker dot in the middle. In addition, the fourth filter in filter stack 11 in Figure 4.2b has a gradient moving left to right. This may give a high response when the paddle is close to the wall, signaling that it is not possible to move further in the same direction. Another situation specific filter for Breakout is the second filter in filter stack 15 in Figure 4.2b. The light pixel may be the ball closing in on the blocks over it. It looks like this filter may give a high response when the ball is about to hit either a block or a hole in the block layer.

Section 3.6.2 predicted that filters being able to detect the ball in both Pong and Breakout should emerge. Intuitively, since the ball is an object both games have in common, this might be a feature that is general for both tasks. A filter able to detect a ball on the input frames should consist of high pixel values (light) surrounded by lower pixel values (dark). A filter with these properties will give a high response when it is slid over the location of the ball on the input frame during the convolution. There are some filters in Pong that may be a feature detector for the ball. Namely, the fourth filter in filter stack 19 in Figure 4.1a and the fourth filter in filter stack 22 in Figure 4.1b.

For Breakout, it is hard to detect any filters with the potential of being a general feature detector for the ball. An example of this is the filters in filter stack 29 in Figure 4.2a. The light pixel areas may be the ball, but there are other, darker

artifacts in the filters suggesting the filters are situation specific. Perhaps the filters give a high response when the ball is closing in on the wall or the blocks in a certain manner.

It is hard to find features that might be similar in both networks. In addition to being specific to certain situations, a reason for this may be that a combination of the filters in the same stack is needed to extract a feature for each input to the network. As we know, the input to the network consists of four frames. Each filter in the first layer is applied to one of the frames in the input stack. This may be a reason why it is difficult to visually compare the different layers. In addition, the filters may have co-adapted. This is a phenomenon occurring when training neural networks without distinct regularization techniques. By having co-adapted filters, they depend on each other to be able to extract meaningful features. This is not a good thing, and often leads to overfitting and low generality. A regularization technique often used to cope with this, is *dropout*, proposed by Srivastava et al. [2014]. The technique involves randomly dropping different neurons from the network during training. This prevents strong dependencies between them and reduces the co-adaptation. Dropout is not used in the DQN, but it would be interesting to see if incorporating some sort of regularization would result in less situation specific filters.

When taking a closer look at the Pong layers, it seems like some of them have remained unadjusted during the training. By looking at Figure 4.1b and the corresponding filters in Figure 4.1c, filter stacks 4, 5, 16, 17, 18, 25 and 29 seems to not have been adjusted at all since initialization. This is also the case with filter stacks 2, 5, 6, 8, 10, 22, 24 and 30 when comparing Figure 4.1a and Figure 4.2c. A reason for this may be that the filters are poorly initialized. Recall that the activation function used is the rectifier described in Section 3.3.2. If the filter response when performing the convolution is negative, the rectifier function will simply clip it to 0. If the filters are initialized in a way that results in them always giving a zero response, they may not get adjusted by the backpropagation algorithm during training.

Another reason may be that the network architecture is too large. From Figure 3.1 it is clear that the only objects that is crucial for the agent to understand in Pong, are the two paddles on the sides and the ball. The rest of the screen is not very important as it will not affect the score in any way. The current model architecture of the DQN may be too excessive for the game of Pong. The game may not be complicated enough as a task for all the filters to be needed. This may result in the filters giving the least response when starting training, simply not being adjusted. The result of this is that many of the filters will give a zero

response most of the time. Nevertheless, the agent seems to perform well without using all the filters in the first convolutional layer of the network.

In Breakout, this is not the case. From Figure 4.2 it is clear that all the filters are adjusted. Breakout is harder and more complex to play. Since the task of the agent is to break all the blocks, they must all be considered. In addition, the horizontal paddle and the ball is crucial for the agent to keep track of (see Figure 3.2). The game pace is also higher. In other words, there are more elements to keep track of and less time to react when playing Breakout. This may be an explanation why all the filters in the first layer of the Breakout networks are used.

4.3 E1 - Low-Level Features - Difference Score

This section presents the results from the second part of Experiment E1 regarding the computation of the difference score. This is used to compare the first convolutional layers between trained networks and will help identify similarities between the low-level features in networks trained on similar tasks. Then, the results are discussed and analyzed.

4.3.1 Results

The results from finding the difference score between the networks are shown in Table 4.1. It consists of all the computed scores describing the difference between the first convolutional layers in the networks. The leftmost column presents the different networks used as DQN_A, while the topmost row presents the different networks used as DQN_B. Because the comparison is done both ways, all the networks are used as both DQN_A and DQN_B. The name of the networks in the table corresponds to the names found in Figure 4.1 and Figure 4.2.

To see the results more clearly, Table 4.2, is constructed by averaging the corresponding comparisons. For instance, take the comparison between Breakout_1 and Pong_1. In Table 4.1 there is one difference score for each comparison, the first using Pong and the second using Breakout as DQN_A respectively. The resulting scores are 0.2058 and 0.1964. In the sparse Table 4.2, these numbers are averaged, giving a difference score of 0.2011.

The results made bold in Table 4.2, are not as expected. It appears that in both networks trained on Pong, several of the filter stacks remain unchanged throughout training. The filter stacks are equal to the corresponding filter stacks in the untrained randomly initialized network in which they are based on. By

Table 4.1: Difference score when comparing the various networks. Both two-way comparisons are present in this table.

$\mathbf{DQN_A} \downarrow \mathbf{DQN_B} \rightarrow$	$\mathbf{Pong_1}$	$\mathbf{Pong_2}$	$\mathbf{Breakout_1}$	$\mathbf{Breakout_2}$	$\mathbf{Random_1}$	$\mathbf{Random_2}$
$\mathbf{Pong_1}$	0.0000	0.1967	0.2058	0.2093	0.2524	0.1696
$\mathbf{Pong_2}$	0.1955	0.0000	0.2062	0.2082	0.1771	0.2509
$\mathbf{Breakout_1}$	0.1964	0.1961	0.0000	0.1663	0.2495	0.2456
$\mathbf{Breakout_2}$	0.1980	0.1980	0.1680	0.0000	0.2486	0.2444
$\mathbf{Random_1}$	0.2535	0.1782	0.2519	0.2508	0.0000	0.2702
$\mathbf{Random_2}$	0.1714	0.2530	0.2501	0.2504	0.2705	0.0000

Table 4.2: Difference score when comparing the various networks, based on numbers from Table 4.1. Scores for the corresponding two-way comparisons are averaged for the table to be more sparse and the result easier to read.

	$\mathbf{Pong_1}$	$\mathbf{Pong_2}$	$\mathbf{Breakout_1}$	$\mathbf{Breakout_2}$	$\mathbf{Random_1}$	$\mathbf{Random_2}$
$\mathbf{Pong_1}$	0.0000					
$\mathbf{Pong_2}$	**0.1961**	0.0000				
$\mathbf{Breakout_1}$	0.2011	0.2011	0.0000			
$\mathbf{Breakout_2}$	0.2036	0.2031	0.1671	0.0000		
$\mathbf{Random_1}$	0.2529	**0.1776**	0.2507	0.2497	0.0000	
$\mathbf{Random_2}$	**0.1705**	0.2519	0.2478	0.2474	0.2703	0.0000

extracting the unadjusted filters from the Pong networks and computing the difference score one more time, the results are closer to what was expected. These results are presented in Table 4.3. The cells made bold in the table are the difference scores that changes after extracting the unadjusted filters. The rest of the scores are the same as in Table 4.1. Section 4.3.2 will discuss the results found in this experiment.

Table 4.3: Difference scores after extracting the unadjusted filters in both Pong networks. The cells made bold indicate that the score has changed after extracting the unadjusted filters. The other scores are the same as in Table 4.1.

$\mathbf{DQN_A} \downarrow \mathbf{DQN_B} \rightarrow$	$\mathbf{Pong_1}$	$\mathbf{Pong_2}$	$\mathbf{Breakout_1}$	$\mathbf{Breakout_2}$	$\mathbf{Random_1}$	$\mathbf{Random_2}$
$\mathbf{Pong_1}$	0.0000	**0.1779**	**0.1905**	**0.1956**	**0.2464**	**0.2235**
$\mathbf{Pong_2}$	**0.1787**	0.0000	**0.1936**	**0.1962**	**0.2263**	**0.2453**
$\mathbf{Breakout_1}$	0.1964	0.1961	0.0000	0.1663	0.2495	0.2456
$\mathbf{Breakout_2}$	0.1980	0.1980	0.1680	0.0000	0.2486	0.2444
$\mathbf{Random_1}$	**0.2541**	**0.2332**	0.2519	0.2508	0.0000	0.2702
$\mathbf{Random_2}$	**0.2331**	**0.2531**	0.2501	0.2504	0.2705	0.0000

4.3.2 Analysis and Discussion

Looking at Table 4.2 there are some expected results, but also unexpected results. The unexpected results are made bold. Comparing the two Breakout networks gives a low difference score, around 0.16. This is expected. However, comparing the two Pong networks results in a higher difference score, around 0.19. This is almost as high as the difference between a Pong and a Breakout network. The reason for this is the unadjusted filters in the Pong networks, first mentioned in Section 4.2.2. Since as many as eight of the filters in Pong_1 and seven of the filters in Pong_2 (as seen in Figure 4.1) can be regarded as random even after training, it makes an impact on the difference score. The fact that comparing anything to a random filter tends to give a high difference score supports this suspicion. The unadjusted filters effectively raise the difference score when comparing the two Pong networks.

We see that the unadjusted filters in both Pong networks push the difference score up when comparing the two. This is also the reason for the relatively low difference between Pong_2 and Random_1. The Random_1 network is simply a randomly initialized untrained network and the Pong_2 network is the result of

training the Random_1 network to play Pong. Since several of the filter stacks in Pong_2 remains unchanged during training, they are equal to the corresponding filter stacks in the Random_1 network. This is likely the reason why the difference score between the Pong_2 network and the Random_1 network is so low. The same reasoning applies to the unexpected low difference score between the Pong_1 network and the Random_2 network. The Pong_1 network is the result of training the Random_2 network to play Pong.

Another observation that supports this reasoning is found in Table 4.1. If we look at the comparisons between Breakout and Pong. When using Pong as DQN_A, the numbers are slightly higher than when Breakout is used as DQN_A, going from around 0.19 to 0.20. This is also presumably due to the unadjusted filters in the Pong networks. As mentioned in Section 3.6.3, when using a network as DQN_A all its filters will be compared to their most similar counterpart in DQN_B. This means that when Pong is used as DQN_A, the unadjusted filters are forced to be compared. However, when a Breakout network is used as DQN_A the unadjusted filters in Pong may be ignored as there are other filters that are more similar. This may be the reason for the difference scores being slightly different.

To confirm the suspicion that the unadjusted filters in the two Pong networks affect the resulting difference scores, the comparisons were redone. This time, the unadjusted filters were skipped in the comparison process. The results are presented in Table 4.3.

Comparing Table 4.1 and Table 4.3 the numbers indicate that it is indeed the unadjusted filters in the Pong networks that affect the initial results. From Table 4.3 we see that the difference score between the two Pong networks has decreased, as expected. We can also see that the difference score between the Pong networks and the Random networks has increased and better correspond with the expectation. When using Breakout as DQN_A, the difference scores remain unchanged. This does indeed indicate that the unadjusted filters in the Pong networks are too different from any of the filters in the Breakout networks to be used in the original comparison. Additionally, we can see that when using Pong as DQN_A, the difference score between Breakout has slightly decreased from around 0.20 to 0.19 when extracting the unadjusted filters. These numbers correspond better with the difference scores found when using Breakout as DQN_A.

The rest of the difference scores are as expected. We can see that comparing two random networks yield a high difference of around 0.27. After extracting the unadjusted filters, comparing either Breakout or Pong to a random network, results in a relatively high difference of between 0.23 and 0.25. Comparing Pong

to Breakout yields a lower score of around 0.19. The first layers in the two Breakout networks have a lower difference at around 0.16. This is also the case when comparing the two Pong networks after the unadjusted filters are extracted from the comparison, about 0.17.

The difference scores are as expected, suggesting that the first convolutional layer in the Pong and Breakout networks may have more in common than a randomly initialized network, but not as much in common as a network trained on an equal task.

4.4 E2 - Cutback in Training Time - Pong as Target

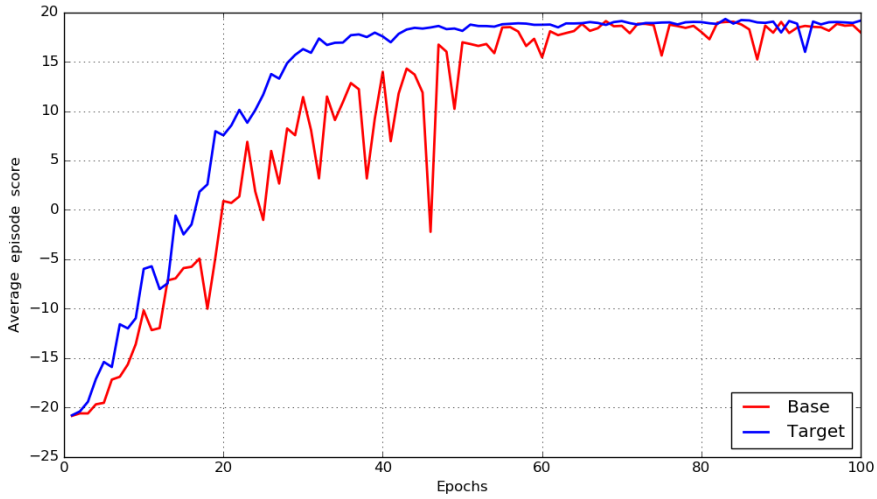
This section presents the results from Experiment E2, using Pong as target task when investigating cutback in training time. Then, the results are discussed and analyzed.

4.4.1 Results

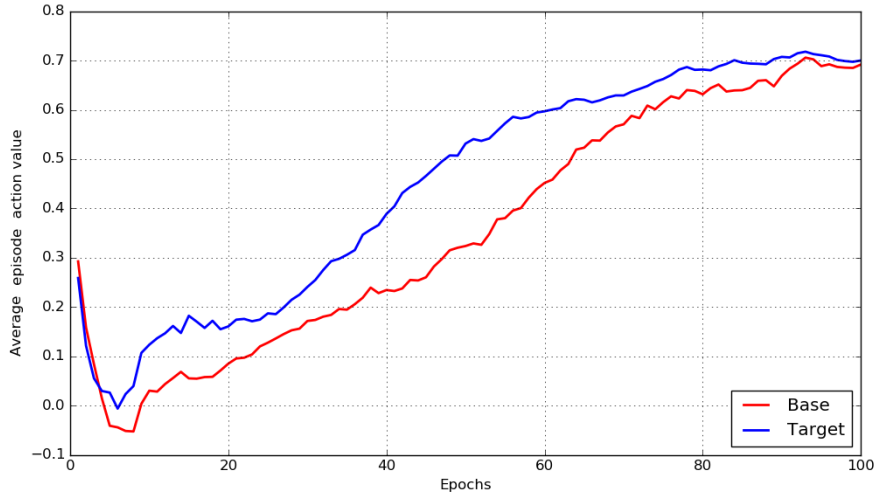
The results from Experiment E2 using Pong as target network are depicted as graphs in Figure 4.3. As mentioned, the experiment is performed three times with identical settings and averaged to mitigate statistical anomalies. Figure 4.3a plots the average episode score per epoch, gathered during the testing epochs. Figure 4.3b plots the average episode action value per epoch, also collected during the testing epochs. Figure 4.3c plots a moving average of the mean loss per game episode. The loss is gathered during the training epochs. All graphs plot data of both the Pong target network and the Pong base network for comparison.

4.4.2 Analysis and Discussion

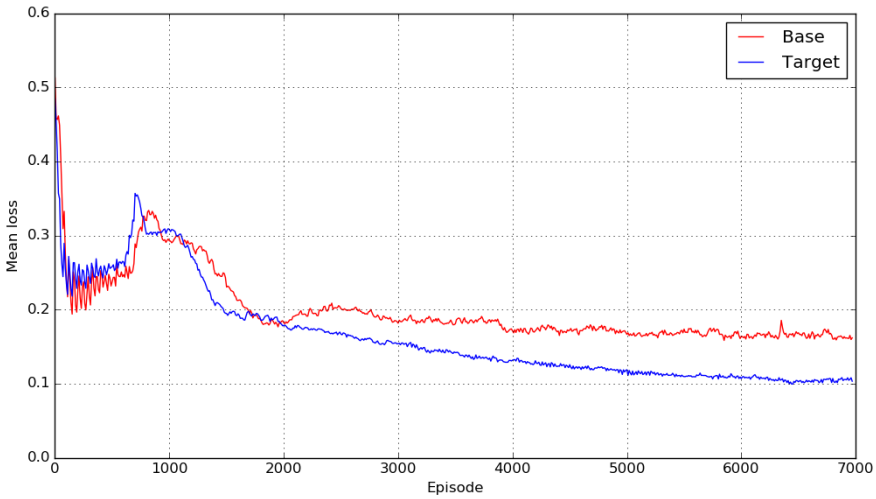
As seen in Figure 4.3a, when playing Pong, the target network does get a higher average score faster than the base network and it looks like it has a higher score for the duration of the training until around epoch 70. It is also interesting to see that the score seems to be somewhat more stable for the target network, not fluctuating as much as the base network. Both networks seem to converge towards an average score of 18-19. This means that on average, the agent scores 21 points by deflecting the ball past the opponent (and thereby winning the game), while only conceding 2-3 balls itself. Both the base network and the target network seems to reach the same score over time.



(a) Average episode score per epoch.



(b) Average episode action value per epoch.



(c) Mean loss moving average per episode.

Figure 4.3: Results when using Pong as target task. Best viewed in colors.

In terms of cutback in training time, it seems like the target network has converged at around epoch 40-45. The base network needs at least 60-70 epochs before getting the same score and converging. This is a cutback of 15-20 epochs, or up to 5 million less frames. This cutback amounts to around a day of training time with the hardware setup used for this experiment. It is hard to say why the target network achieves a more stable score throughout the training. The score graph may indicate that knowledge from the pre-trained first layer makes the target network perform more stable. In addition, the fact that the target network achieves a higher score faster than the base network may suggest that there are some features from Breakout that the Pong network can take advantage of.

Figure 4.3b plots the average action value per epoch. The action value is the reward the agent expects at any given time. Since the agent is rewarded one point for deflecting the ball past the opponent, it is clear that the highest reward possible for the agent to expect by doing an action at any time is 1. We can see from the graph that the action value for the target network is generally higher than for the base network. Towards the end of training, the two networks have more similar values, but from epoch 40-70, the target network does expect the reward for each action it takes to be higher than the base network. At around

epoch 50, the target network expects almost 20% higher reward for the actions it takes in general.

The action value and the score are tightly coupled. Since the average score climbs faster for the target network, it is only natural that the action value, or expected reward, will be higher on average for the target network as well. One peculiar observation about the action value graph is the drop during the first few epochs. A possible explanation for this is that the final fully connected layer before the output layer in the model architecture needs some time to calibrate. It is this final fully connected layer that maps the features found by the convolutional layers to actual action values for each valid action. They both start out with randomly initialized weights which will give inaccurate action values for the different valid actions. After a few epochs, the initial adjustment of the weights creates the base for the understanding of how the action values are connected with the different actions.

Figure 4.3c plots a moving average of the mean loss per game episode. The loss is recorded for training epochs. The data points are down-sampled by a factor of 10 with a moving average. This is done to get a more clear plot while still being an accurate representation of the mean loss per episode. As mentioned in Section 3.3.2, the mean loss is used as a learning signal for the network during training. Higher loss means that there is a larger distance from the action predictions the network is currently doing and what the target network, $\hat{Q}$ (see Section 3.3.2), considers the "correct" action prediction for that state. Since the "correct" prediction is dependent on the weights of the network, this loss can not be interpreted the same way a supervised loss signal would. However, it is still what the network uses for learning.

The mean loss for both the target network and the base network are mostly the same until around episode 2,000. From this point on the target loss continues to drop while the base loss seems to flatten out. The sudden mean loss increase right before episode 1,000 is an unexpected observation. Generally, a loss graph is expected to drop constantly and flatten out over time if the network is properly configured. This kind of behavior might indicate that something is off.

One reason for this increase may be the exploration factor, ϵ, explained in Section 3.3.4. The ϵ value represents the exploration probability of the agent. The value starts at 1.0 and is decreased to 0.1 over the first million frames. This happens to be around episode 1,000 where the mean loss is about to decrease again. This is probably the best explanation for this hump in the graph. There is still a decent probability that the agent will choose a random action instead of the action pro-

viding the highest action value. If the agent somehow chooses the random action many times in a row, it would certainly give a higher mean loss, as the actions it is currently taking is further from the targets used to calculate the loss.

Nevertheless, an odd-looking training loss graph does not provide much information in itself. The score and action value are more valuable metrics to evaluate the performance of the agent during training by. As long as the agent seems to learn and acts in an intelligent manner on the tasks, the training loss is of less importance.

The results from using Pong as target task seems promising. There are indications that the knowledge contained in the first convolutional layer in a base network trained to play Breakout does have some general knowledge that can be transferred to the task of playing Pong. However, as the results are only averaged over three experiment runs, it is not possible to make any conclusive statements.

4.5 E3 - Cutback in Training Time - Breakout as Target

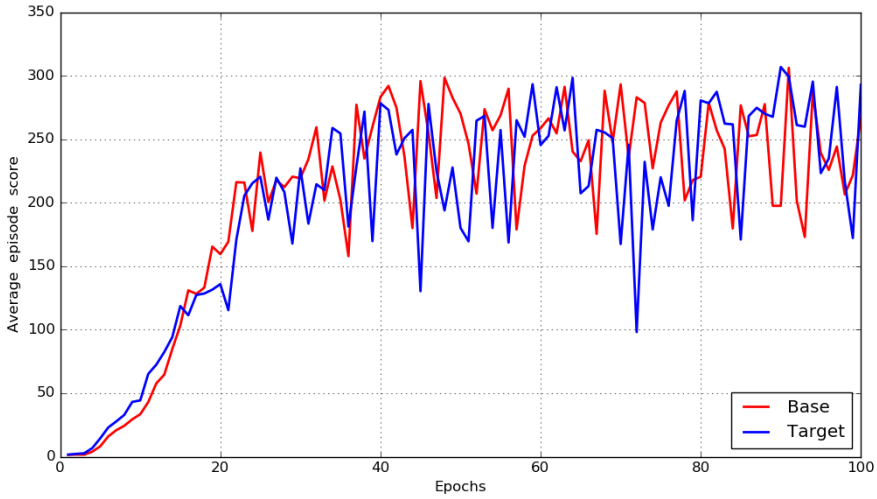
This section presents the results from Experiment E3 using Breakout as target task when investigating cutback in training time. Then, the results are discussed and analyzed.

4.5.1 Results

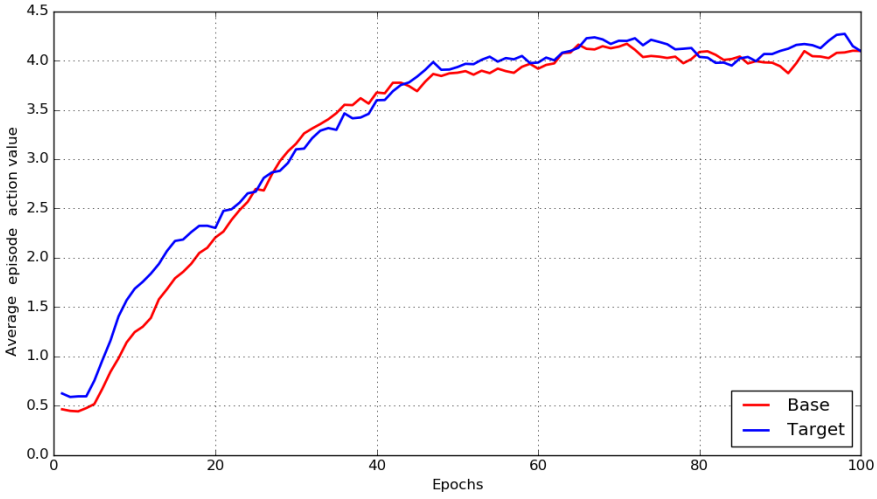
The results from Experiment E3 using Breakout as target network are depicted as graphs in Figure 4.4. As mentioned earlier, the experiment is performed three times with identical settings and averaged to mitigate statistical anomalies. Figure 4.4a plots the average episode score per epoch, collected during the testing epochs. Figure 4.4b plots the average episode action value per epoch, also gathered during the testing epochs. Figure 4.4c plots a moving average of the mean loss per game episode. The loss is gathered during the training epochs. The graphs plot the collected data for both the Breakout target network and the Breakout base network for comparison.

4.5.2 Analysis and Discussion

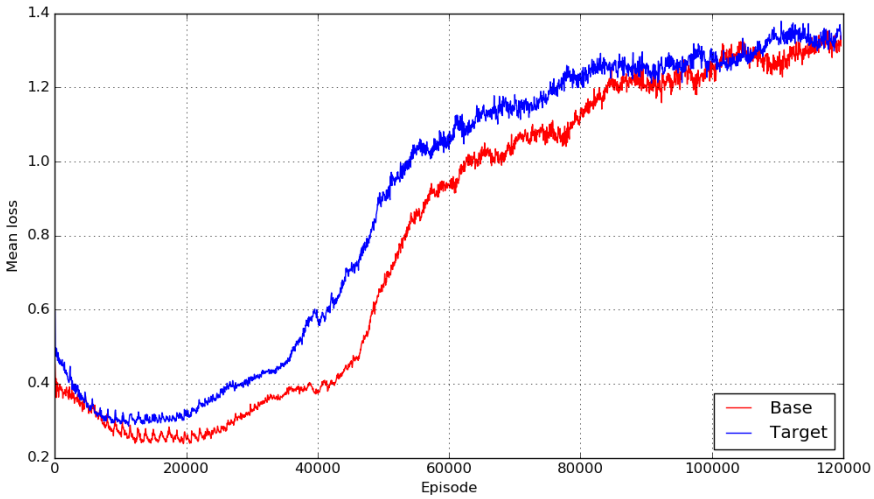
Figure 4.4a plots the average episode score per epoch. As seen in this figure, both the base network and the target network seem to perform equally. Based on the graph, it is hard to determine whether the target network achieves a better score



(a) Average episode score per epoch.



(b) Average episode action value per epoch.



(c) Mean loss moving average per episode.

Figure 4.4: Results when using Breakout as target task. Best viewed in colors.

than the base network. Even though the graph shows the score averaged over 3 runs, the score seems to fluctuate and vary a lot compared to the score from the Pong experiment in Section 4.4.2. An explanation for this may be that doing a random *explorative* move can influence the score more drastically in Breakout than Pong. During a testing epoch, which takes place after every training epoch, the explorative factor, ϵ, is fixed at 0.05. In other words, there is a 5 % chance of the agent choosing a random action. This is done due to the Atari emulator depending on player input to generate randomness, as described in Section 3.3.4. Although the chance of doing a random action is the same for both Pong and Breakout, making a random (and most likely sub-optimal) move in Breakout may influence the score more since the game pace is higher. There is much less time to correct bad choices.

Another reason why the Breakout score seems more variable than the Pong score, may be that it is distributed over several hundred points. In Pong, the score is always between -21 and 21. In Breakout, the score can be between 0 and several hundred. The explanation for the varying score may simply be the nature of the game itself and how it is designed to be played.

Figure 4.4b plots the average episode action value per epoch. Neither the tar-

get network nor the base network seems to achieve a higher action value than the other. Both the target and the base network seem to increase steadily until around epoch 70, where they both stabilize at an action value at about 4. The keen reader may have noticed that the action value is higher for Breakout than for Pong. This is because of the scoring system in the respective games. When a Breakout episode starts there are 6 layers of breakable blocks at the top of the screen. The blocks in the two lowest layers earn the agent one point when hit. The blocks in the third and fourth layer earn the agent four points when hit. The two top layers consist of blocks worth seven points when hit. This means that the highest reward the agent can expect from doing an action at any point in time is seven. An average action value over 4 seems reasonable for a well-performing agent, as it is impossible to hit the blocks yielding maximum points with every deflection of the ball.

Figure 4.4c plots a moving average of the mean loss per episode. For Breakout, the data is down-sampled by a factor of 50. The reason for this is the same as for Pong, that the loss is sampled at every end of an episode during the training epochs. In Breakout, the episodes are much shorter than Pong. As we can see by comparing the x-axis in Figure 4.4c and Figure 4.3c there are a lot more episodes for Breakout than for Pong. The reason for this is that Breakout is a more fast paced game and the duration of an episode is shorter on average.

The loss graph seems a little strange. Generally, a training loss signal will start with a high value and decrease as training progresses, converging at a low value close to 0. The training loss signal for Breakout starts out as expected, decreasing over the first 20,000 episodes. However, from this point on it starts to increase. As mentioned in Section 3.3.3, the loss describes how far the current predicted Q-value is from the target Q-value. Based on this, the weights are adjusted accordingly. A higher loss indicates that the current Q-value is further from the target Q-values.

In supervised learning, a climbing *test loss* signal may indicate that the network is overfitting. It is harder to draw the same conclusion with the training loss. The DQN does not have a testing set with unseen examples that can judge how well the agent has generalized to the task, as supervised learning methods require. In contrast to the labels used as targets in supervised learning, the targets in the DQN system are dependent on the weights of the target network, $\hat{Q}$, which is a copy of the current DQN some iterations ago (see Section 3.3.3). Because of this, the loss signal is only based on what the network itself considers good targets according to the knowledge it possessed some time ago. Unlike supervised labels, these targets are not necessarily the optimal or "correct" targets.

However, the figure illustrates that the network regards the predicted actions at the end of training as further away from what it considers "correct" behavior given its knowledge at that point during training. Why this is the case is hard to determine, especially since both the score and the action value are as expected and the agent appears to behave reasonably. Recall that the training in this thesis is done over 25 million frames rather than the 50 million frames done by Mnih et al. [2015]. It could be interesting to see how the loss would evolve over the next 25 million frames. Generally, as mentioned in the analysis of Pong in Section 4.4.2, the training loss does not provide very valuable information in itself. A strange training loss does not necessarily mean that anything is wrong.

Training Breakout as target network with its first layer transferred from a trained Pong base network did not seem to affect the result much. Both the base network and the target network progressed equally during training. A reason for this may be that Pong requires less skill to achieve a good score. In Pong it is sufficient to avoid conceding the ball, in addition, the ball and the pace of the game are slower. The Pong base network may not have knowledge general enough that the Breakout target network can take advantage of it. This may be why the results are better when using Pong as target network compared to using Breakout as target network. Breakout is harder to play and requires more skill. The results from the other experiments support this claim as well. For instance, the unadjusted weights found in the Pong networks. There are simply more task specific knowledge contained in the Pong networks than the Breakout networks.

When choosing tasks for these experiments, the difficulty was not considered. As seen from the results, this may be a weakness with the experimental design. By choosing tasks more equal in terms of difficulty, the results might be improved.

The knowledge contained in the first layer of a fully trained Breakout base network seems to be more general and transferable than the knowledge contained in the first layer of a fully trained Pong base network. At least when transferring between these two tasks.

4.6 Performance of the DQN

The purpose of this section is to discuss the performance achieved with the DQN. Although not critical to the research questions or goal of this thesis, it can be interesting to compare the performance with that of Mnih et al. [2015] to confirm that the agent does acquire knowledge.

To measure the agent's performance, Mnih et al. [2015] use the evaluation from the test epochs after every training epoch, just as this thesis does. Their test epochs last 135,000 validation frames, as opposed to 125,000 in this thesis. In addition, their agent is trained on 50 million frames, as opposed to 25 million in this thesis. The highest achieved average score from their testing epochs is used.

In Pong, Mnih et al. [2015] achieved a score of 18.9($\pm$1.3). This is very similar to the score the agent in this thesis achieved. From the graph in Figure 4.3a, we can see that the score the agent achieves is around 18 to 19. However, the highest achieved average score during a single test epoch is 19.2. It seems like the performance of the agent playing Pong is similar to that of Mnih et al. [2015].

In Breakout, Mnih et al. [2015] achieved a score of 401.2($\pm$26.9). Compared to the score the agent in this thesis achieved it is a bit higher. From the graph in Figure 4.4a, we can see that the agent rarely achieves a score higher than 300. However, the highest achieved average score during a single test epoch is 369.7. Although not as high as that of Mnih et al. [2015], the score confirms that the agent does acquire knowledge.

To give an idea of how the skills of the agent develops during training, some videos have been made to demonstrate this. How these can be watched and their descriptions are provided in Appendix A.

4.7 Results Summary

This chapter has presented the experimental results. In addition, they have been analyzed and discussed.

In the experiments regarding similarities between the low-level features in networks trained to play Pong and Breakout, the results were mixed. There were signs of general features, especially in Breakout. However, there were filters from both games resembling task specific feature detectors.

In addition to the visual comparison between the tasks, a difference score was found between the first layer of the different networks. These results were more promising, indicating that the difference between the first layers in Pong and Breakout was less than their difference to a randomly initialized network. This may suggest some level of generality between the low-level features.

For the experiments regarding cutback in training time, the results were mixed. Training Pong as target network with its first layer transferred from a fully trained

Breakout base network gave promising results. Both the score and the action value increased faster and converged earlier for the target network than the Pong base network during training.

However, training Breakout as target network with its first layer transferred from a fully trained Pong base network did not seem to result in faster convergence. Both the target network and the Breakout base network progressed in a similar manner during training and converged at around the same epoch. The fact that Breakout is more challenging to learn compared to Pong could be the reason why we are not seeing a cutback in training time. Pong seemed to be less challenging, this was supported by the fact that the trained Pong networks contained a considerable amount of unadjusted filters.

Although the results are inconclusive, there are indications worth exploring. The next chapter will evaluate the results in terms of the goal and research questions. In addition, it will outline possible future work and how to explore these subjects further.

Chapter 5

Conclusion

5.1 Overview

This chapter begins by giving a summary of the thesis in Section 5.2. Section 5.3 evaluates the results in terms of the defined goal and research questions. Contributions are described in Section 5.4. Finally, Section 5.5 describes how the work done in this thesis can be built upon and my thoughts on interesting paths for this field of research.

5.2 Thesis Summary

Chapter 2 gave an introduction to the field of deep learning, CNNs and reinforcement learning. A literature review resulted in the various papers presented as related work which provided an insight into the current state of the field of deep reinforcement learning. It focused on how to further build on the DQN proposed by Mnih et al. [2015]. Especially, how to compress the knowledge from several DQNs into a single more general DQN. The related works revealed that a more direct approach to investigating how general the knowledge contained in deep CNNs could be a valid contribution to the field.

Chapter 3 contained detailed information about the system, domain and technology used to perform the experiments in this thesis. A DQN, first proposed by Mnih et al. [2015], was implemented based on a recreation by Sprague [2015]. Three experiments were designed according to the research questions. Part one of experiment E1 was concerned with visually comparing the lower layer features in various DQNs trained on similar tasks. The second part quantified the similarity between the lower level features in various DQNs trained on similar tasks.

Experiment E2 transferred the Breakout base network to a Pong target network to investigate how training time could be reduced. Finally, Experiment E3 did the same as E2, only using Pong as base network and Breakout as target network.

Chapter 4 described, analyzed and discussed the results from the experiments. First, part one of Experiment E1 performed a visual comparison between the first convolutional layer in a DQN trained on the games Pong and Breakout. There were indications that the networks had some limited knowledge that was general to the two tasks, but most filters seemed to be specific to the task. Second, the quantified difference between lower level features in part two of Experiment E1 was calculated. The results were positive in that the tasks Breakout and Pong seemed to have some similarities in their lower level features. Experiment E2 and E3 transferred the first layer of a pre-trained DQN and used it to train a target network. In Experiment E2, the results showed that Pong did perform better using pre-trained weights from Breakout. However, in Experiment E3, Breakout did not and the performance was equal to a network without pre-trained weights.

5.3 Goal Evaluation

The goal of this thesis was to investigate how general the knowledge contained in a DNN is. The motivation behind investigating the generality came from Mnih et al. [2015], their DQN and their statement of it being a general solution. Because of this, it was natural to use the DQN for the experiments in this thesis. The DQN was used to perform experiments aimed at addressing the research questions. This section discusses the results achieved with the research questions in mind.

Research question 1 *Will there be any similarities between the lower level features in DNNs trained on similar tasks?*

For research done with CNNs applied to image detection and image recognition, we know that the lower level features in CNNs often correspond to known image processing filters, like edge detection filters such as Gabor filters. These filters can be regarded as general features, as most images contain edges. Edge detection is applicable in most systems aimed at somehow interpreting images, including the DQN.

The first experiment, E1, was designed with this research question in mind. By training different networks to play Pong and Breakout, the first convolutional layer could be visualized and interpreted. The interpretation was first done manually by visually comparing the filters in the first layers of the networks. This

was done to identify similar filters that might have been optimized independently of the task. Second, a difference score was computed between the different layers. The difference score served as a quantified measure of the difference between the layers.

The results from E1 when visually comparing the different filters were divided. In the Breakout networks, horizontal edge detection filters could clearly be identified. The Pong networks on the other hand, evolved slightly specific vertical edge detection filters. The edge detection filters were expected since the respective games mostly consist of horizontal or vertical objects. However, since the games seem to only include one of the filter types, they might not be very general in respect to each other. Both Pong and Breakout had some task specific filters. Generally, it was hard to visually identify similarities between networks trained on the two games.

The results of computing difference scores were more promising and showed that the first convolutional layer in networks trained to play Pong and Breakout were more similar than when comparing either of them to a randomly initialized network. It is not possible to draw any conclusions based solely on this experiment. But as an answer to the research question, there are indications that DQNs are developing similar lower level features for tasks that are related.

Research question 2 *Will there be a significant cutback on the training time for a network already trained on a similar task? How significant?*

The process of training DNNs is computationally expensive and require modern hardware to be feasible to carry out. Lately, the task of reducing training time of DNNs has been in focus for research in this field.

The second experiment was designed with this research question in mind. DQNs were trained on Pong and Breakout and named base networks. The trained weights from the first convolutional layer in the Pong base network were used as initial weights in the Breakout target network. The target network with the modified first layer was then trained. For the Pong target network, a Breakout base network was used as initial weights. By comparing the score, action value and mean loss during training of both the base networks and the target networks, research question 2 could be addressed.

The results when using Pong as target network were promising. During training, both the score and the action value of the target network converged at an earlier point compared to the base network. Around halfway through training, the target network had a 20 % higher action value than the base network. The average

score was also higher for the target network. For the base network to achieve the same numbers, it needed 10-20 more epochs of training, or up to 5 million more frames to train on. In terms of training time, this amounts to a cutback of around 24 hours with the hardware used in this experiment.

The results when using Breakout as target network were not as promising. The score and action value for both the base network and the target network developed at an equal pace and converged at the same time. Although the performance of the agent trained on Breakout was adequate, the training loss signal resulted in a peculiar plot. Based on the results from Breakout alone, there were no indications that initializing the first convolutional layer with a layer pre-trained on a similar task gave advantages in terms of reduced training time. At least with the current experimental setup.

Although the results when using Pong as target network were promising, looking at the results combined, no definitive conclusions can be drawn in terms of cutback in training time. More work should be done to confirm the results from Pong. One suggestion is to choose tasks that are more equal in terms of difficulty. Since Pong requires less skill and knowledge, it may not contribute as much to the target Breakout network.

5.4 Contributions

The generality of DNNs has been explored. It has been shown that there are indications that the lower level features in CNNs do have similarities and may contain some general knowledge. This has already received some attention in regards to object recognition in images, but has, to my knowledge, not been done with DQNs in the domain of Atari games before. The contributions to the field are also done by supporting earlier found evidence.

To my knowledge, there have not been any previous efforts concerned with investigating how pre-trained layers in a DQN can give a cutback in training time. This thesis has shown that there are indications that cutback in training time can be achieved by using knowledge previously obtained as a starting point when learning to do new and similar tasks. However, this thesis can only be regarded as exploratory research of this subject. More experimentation is needed in this area before any conclusions can be made.

5.5 Future Work

To mitigate the statistical anomalies in the results, they should be averaged over more experiments. This is at least true for the experiments targeted at cutting down on training time. In this thesis, the results presented are only averaged over three runs due to limits in computational power. Also, training the DQNs could be done for longer, training for 50 million frames instead of 25 million frames like Mnih et al. [2015].

Additionally, the same experiments should be performed on the other convolutional layers in the DQNs. An experiment similar to Yosinski et al. [2014], described in Section 2.6.2 could be done to quantify the generality of each layer. This could pinpoint the convolutional layer in which the knowledge becomes too specific to be used in transferring.

In addition, the experiments should be conducted involving more tasks. In this thesis, only the games Pong and Breakout were used as tasks. Other games being both more similar and less similar should be used. By doing the same experiments on a wide range of games with different levels of similarity, it could be possible to determine how similar the tasks need to be for a potential cutback in the training time to take effect.

Recall that using Pong as base network did not result in cutback in training time on a target Breakout network. The reason may be that Pong is an easier game to learn and less knowledge may be required to play it compared to Breakout. It would be interesting doing the same experiments using tasks that are considered to be at the same level of difficulty. One approach could be to make Pong more difficult by increasing the length of the opponent's paddle or placing a block in the middle of the screen to increase the unpredictability of the ball. This would force the agent to aim better. Furthermore, the game pace could be increased making the ball move faster. This could possibly achieve better results.

Furthermore, it could also be interesting to implement a regularization technique like dropout [Srivastava et al., 2014] in the DQN. This could potentially reduce the co-adaptive neurons in the layers and make them more general. In addition, noise could be introduced in the input data to increase its domain size. This could also result in more general knowledge.

An important aspect of reducing training time is to be able to train a network sufficiently with less data. An interesting thought that takes the concepts from this thesis a little further, is to build a library of pre-trained networks with gen-

eral knowledge. A network from this library can then be used as a base to further fine-tune other networks to more specific tasks, effectively reducing training time. This could possibly be done better with techniques like multitask learning or distillation as mentioned in Section 2.6.

Bibliography

(1999). Atari 2600 history. `http://www.atariage.com/2600/`. Accessed: 2016-04-07.

Ascher, D., Dubois, P. F., Hinsen, K., Hugunin, J., Oliphant, T., et al. (2001). Numerical python.

Azizpour, H., Razavian, A. S., Sullivan, J., Maki, A., and Carlsson, S. (2015). Factors of transferability for a generic convnet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I. J., Bergeron, A., Bouchard, N., and Bengio, Y. (2012). Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop.

Battenberg, E., Dieleman, S., Nouri, D., Olson, E., van den Oord, A., Raffel, C., Schluter, J., and Sonderby, S. K. (2014–2015). Lasagne. `http://lasagne.readthedocs.org/`.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279.

Bengio, Y. (2012). Deep learning of representations for unsupervised and transfer learning. *Journal of Machine Learning Research (JMLR)*, 27:17–36.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation.

Caruana, R. (1997). Multitask learning. *Machine Learning*, 28(1):41–75.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 160–167, New York, NY, USA. ACM.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. *Journal of Machine Learning Research (JMLR)*, 15:315–323.

Goertzel, B. and Pennachin, C. (2007). *Artificial General Intelligence*. Springer-Verlag.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.

Hausknecht, M. J. and Stone, P. (2015). Deep recurrent q-learning for partially observable mdps. *CoRR*, abs/1507.06527.

Hecht-Nielsen, R. (1989). Theory of the backpropagation neural network. In *Neural Networks, 1989. IJCNN., International Joint Conference on*, pages 593–605. IEEE.

Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554.

Hinton, G. E. and Salakhutdinov, R. R. (2006). Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507.

Hinton, G. E., Srivastava, and Swersky (2012). Overview of mini-batch gradient descent. `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C., Bottou, L., and Weinberger, K., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

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

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg,

S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

Parisotto, E., Ba, L. J., and Salakhutdinov, R. (2016). Actor-mimic: Deep multitask and transfer reinforcement learning. *International Conference on Learning Representations (ICLR)*.

Russel, S. and Norvig, P. (2010). *Artificial Intelligence A Modern Approach*. Pearson Education.

Rusu, A. A., Colmenarejo, S. G., Gülçehre, Ç., Desjardins, G., Kirkpatrick, J., Pascanu, R., Mnih, V., Kavukcuoglu, K., and Hadsell, R. (2016). Policy distillation. *International Conference on Learning Representations (ICLR)*.

Sprague, N. (2015). Parameter selection for the deep q-learning algorithm. In *Proceedings of the Multidisciplinary Conference on Reinforcement Learning and Decision Making (RLDM)*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. Cambridge: MIT Press.

Tzeng, E., Hoffman, J., Darrell, T., and Saenko, K. (2015). Simultaneous Deep Transfer Across Domains and Tasks. *ArXiv e-prints*.

Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3):279–292.

Yosinski, J., Clune, J., Bengio, Y., and Lipson, H. (2014). How transferable are features in deep neural networks? In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N., and Weinberger, K., editors, *Advances in Neural Information Processing Systems 27*, pages 3320–3328. Curran Associates, Inc.

Appendices

A Videos - Agent Training Progression

There are three videos available to watch on YouTube. These are provided for the reader to get a grasp of how the training progression of the agent is.

Video 1: Pong - Training progression
URL: `https://www.youtube.com/watch?v=4KzE3C7Q0mE`
This video shows how the skills of the DQN agent progresses throughout training. First, an untrained agent plays Pong. The movements are clearly random and the performance poor. Second, an agent trained for 50 epochs, or approximately 12 million frames plays. We can clearly see that the agent is starting to understand the concept of the game. However, it is not until after 100 epochs, or 25 million frames of training, that the agent performs very well. We can see that it has found an optimal angle to hit the ball which makes the opponent unable to reach it.

Video 2: Breakout - Training progression
URL: `https://www.youtube.com/watch?v=tyHyvDg0yoc`
This video shows how the skills of the DQN agent progresses throughout training. First, an untrained agent plays Breakout. The movements are clearly random and the performance poor. Second, an agent trained for 50 epochs, or approximately 12 million frames plays. As for Pong, we can see that the Breakout agent is starting to grasp the concept of the game. After 100 epochs, or 25 million frames of training, the agent performs very well. An interesting observation is that the fully trained agent has learned the advantageous strategy of digging a tunnel on one of the sides of the blocks. By deflecting the ball through the tunnel, the ball will bounce on top of the blocks, generating an abundance of points.

Video 3: Pong - Base network vs Target network
URL: `https://www.youtube.com/watch?v=CWC6RFN84PY`

This video shows the difference between a base network and a target network playing Pong. The video is recorded halfway through training, i.e after 50 epochs and approximately 12 million frames. At this stage in training, the target network has started to converge, while the base network still has around 10-20 epochs left before converging. The difference in agent behavior is subtle, but if we look closely, we can see that the base network has not yet found that "optimal" deflection angle. The result of this is rather lengthy rallies. The target network seems to perform as good as the fully trained agent from Video 1.

Pong is the only game with a video comparing the base network and the target network play. The reason is that in Breakout, there is no visible difference in terms of agent performance.

B Source Code - Environment Overview

The source code is available online at `https://github.com/torgeha/dqn`.

Dependencies:

- Python 2.7
- Numpy 1.11
- Theano 0.7
- Lasagne 0.1
- Arcade Learning Environment (ALE) 0.5.1
- OpenCV 3.0.0
- matplotlib 1.5.1

The source code is implemented and run in Ubuntu 14.04. Installing the dependencies should be straight forward using the Linux package manager *APT* (apt-get) and the Python installation manager *Pip*. The environment has not been tested on Windows.

C Source Code - System Overview

There are two kinds of scripts in the GitHub repository mentioned in Appendix B; system scripts and tool scripts. The system scripts are the scripts implementing the DQN and can be found in the */src* folder. The tool scripts are tools implemented to transfer layers, visualize weights, calculate difference score and so on. The tool scripts can be found in the */tools* folder. In addition, scripts used to plot graphs can be found in the */plotting* folder.

Following is a brief overview of the different scripts.

DQN:

- *run.py* - Static class where the hyperparameters can be adjusted.
- *launcher.py* - Initialize all components with correct hyperparameters and start training.
- *ale_experiment.py* - Handles logic for training and agent.
- *ale_agent.py* - Agent wrapping the Q-network.
- *q_network.py* - The DQN implementation.
- *ale_data_set.py* - Replay memory.
- *updates.py* - RMSProp.

Tools:

- *difference_score.py* - Calculate the difference score between a specific layer in two networks.
- *check_similarity.py* - Verify if two networks are equal, layer by layer.
- *visualize_weights.py* - Visualize the convolutional filters in a specific layer in a network.
- *swap_layers.py* - Transfer a specific layer from one network to another.
- *wath_gameplay.py* - Watch an agent play an Atari game by providing a model.

Plotting:

- *plot_loss.py* - Plot the loss signal from result file produced during training.
- *plot_results.py* - Plot the score and action value based on result file produced during training.