NTNU
Norwegian University of
Science and Technology

Visual Pretraining for Deep Q-Learning

Torstein Sandven

Master of Science in Computer Science
Submission date: June 2016
Supervisor: Keith Downing, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Task Description

Reinforcement learning often converges slowly towards a solution, especially in cases with high-dimensional input data such as images. This slow convergence makes it expensive to use for real-world applications. This project attempts to reduce that training time by learning visual features in a base domain and apply transfer learning to speed up the learning process for the reinforcement learning problem. The algorithm will be tested in the domain of Atari 2600 games, a rich and diverse collection of challenges, some of which mirror central aspects of real-world problems.

1. Assignment given: 15 January 2016
2. Supervisor: Professor Keith Downing

Summary

Recent advances in reinforcement learning enable computers to learn human level polices for Atari 2600 games. This is done by training a convolutional neural network to play based on screenshots and in-game rewards. The network is referred to as a deep Q-network (DQN). The main disadvantage to this approach is a long training time. A computer will typically learn for approximately one week. In this time it processes 38 days of game play. This thesis explores the possibility of using visual pretraining to reduce the training time of DQN agents.

Visual pretraining is done by training an autoencoder (AE) to reduce the dimensionality of images. When learning dimensionality reduction, the AE learns visual features by recognizing the structure of the images. To test if the AE can learn general visual features, AEs are trained on different datasets. After the pretraining, transfer learning is used to initialize DQNs with weights from the AE. In order to run the experiments a training system was built using Theano.

The results generally show lower performance for cases with pretraining. This happens for all tested datasets. In fact, there is surprisingly little difference in the performance of AEs trained on different datasets. The lower performance most likely occurs because the trained AE focuses on large objects. Small moving objects are often not reconstructed correctly by the AE. These objects are often crucial to the reinforcement learning task. As a result, the image representation learnt by the AE is insufficient for the DQN agent. In addition, the weight magnitude is increased when AEs are trained. Since the parameters for the learning algorithm are tuned for smaller weights, it takes longer to correct the weights. In conclusion, the pretraining was harming the performance. Several possible solutions to this problem are discussed, e.g. increasing the network size, force the AE to focus on moving objects by weighting the loss function, and normalizing the AE.

Sammendrag

Nyere forskning har videreutviklet ”reinforcement learning” slik at datamaskiner kan lære å spille Atari 2600 spill på samme nivå som mennesker. Dette gjøres ved å trene et ”convolutional neural network” basert på skjermbilder og poeng i spillet. Dette nettverket heter ”deep Q-network” (DQN). Ulempen med denne teknikken er at treningstiden er lang. Vanlige kjøringer vil typisk ta 1 uke. I løpet av denne tiden vil datamaskinen prosessere 38 dager med erfaringer fra spillet. Denne hovedoppgaven prøver å redusere treningstiden med å lære DQNet visuelle kunnskaper før det trenes opp til å spille.

Visuelle forkunnskaper læres ved å trene en ”autoencoder” (AE) til å redusere dimensjonaliteten til bilder. Ved å redusere dimensjonen, vil AEen lære visuelle kunnskaper ved å gjenkjenne strukturer i bildene. Flere datasett er brukt for å teste om AEen lærer generelle visuelle kunnskaper som kan brukes i flere spill. Etter den visuelle treningen, brukes ”transfer learning” til å initialisere DQNet med vekter fra AEen. Et egentutviklet system er brukt til å utføre eksperimentene. Systemet er bygget på Theano.

Resultatene viser generelt redusert prestasjon når forhåndstreningen blir benyttet. Dette skjer for all datasett. Det er overraskende lite forskjell mellom datasettene. Den svake prestasjonen er sannsynligvis et resultat av at AEen fokuserer på store objekter. Små bevegelige objekter blir ofte rekonstruert feil. Disse objektene er meget viktige for ytelsen. Dette betyr at representasjonen av bildene som AEen har lært ignorer viktige detaljer. I tillegg øker størrelsen på vektene i nettverket når AEen blir trent. Siden læringsparameterene til treningsalgoritmen er optimalisert for mindre vekter, tar det mer tid å korrigere vektene. Alt i alt viser resultatene at forhåndstreningen er skadelig. Mulige løsninger på disse problemene blir diskutert. For eksempel å øke nettverksstørrelsen, fokusere AEen på områder med bevegelse ved å vekte kostfunksjonen, og normalisere AEen.

Preface

This thesis concludes my Master of Science education in Computer Science at The Norwegian University of Science and Technology (NTNU) in Trondheim. The thesis was performed throughout my 10^{th} semester, spring 2016, at the Department of Computer and Information Science.

I would like to thank my supervisors Professor Keith Downing, at the Department of Computer and Information Science, Norwegian University of Science and Technology.

Table of Contents

List of Tables

List of Figures

Abbreviations

AE	=	Autoencoder
ALE	=	Arcade Learning Environment
ANN	=	Artificial Neural Network
BRL	=	Batch Reinforcement learning
CNN	=	Convolutional Neural Network
CPU	=	Central Processing Unit
DERL	=	Data Efficient Reinforcement Learning
DFQ	=	Deep Fitted Q-Iterations
DQN	=	Deep Q-Learning
DRQN	=	Deep Recurrent Q-Network
ER	=	Experience Replay
FQI	=	Fitted Q-Iterations
GPU	=	Graphical Processing Unit
GSAE	=	Game Specific Autoencoder
INAE	=	ImageNet Autoencoder
LSTM	=	Long Short Term Memory
MDP	=	Markov Decision Process
MLP	=	Multilayer Perceptrons
POMDP	=	Partially Observable Markov Decision Process
RL	=	Reinforcement Learning
SSAE	=	Screenshot Autoencoder
TD	=	Temporal Difference
TL	=	Transfer Learning

Chapter 1

Introduction

1.1 Motivation

Machine learning is a growing industry with applications such as web search, identification of objects in images, speech recognition, and product recommendation. Most of these applications are based on supervised learning. Supervised learning depends on a labelled dataset with sample input and the correct output label. Problems with no correct label or answer is difficult to handle for supervised learning. This is where reinforcement learning (RL) can be applied. Examples are chess playing, stock trading, and control theory. However, RL has few practical examples, partly because of two drawbacks.

First, RL struggles in cases with large or continuous state spaces. For real world applications the state space is typically continuous and RL has therefore not been used. However, recent research has shown that a combination of RL and deep learning can make use of high dimensional images as sensory input, see Mnih et al. (2015).

Second, convergence time in RL is typically slow. In many RL applications there is a cost associated with interactions between an agent and environment. Since standard RL require many sample interactions to function optimally, the cost becomes a prohibitory factor. An example that illustrates the problem of convergence time is Mnih et al. (2015). This paper use 38 days of gameplay to train a computer to play Atari games. This algorithms can achieve impressive results for Atari games but many practical applications require computationally expensive simulations or even physical testing. Which means that this approach is infeasible.

Humans that play Atari games for the first time, learns to play much faster than computers. One of the reasons why humans solves these tasks faster, is that humans can reuse previously learnt knowledge. It is a well known fact that the visual system of

humans extract general visual features that are useful for diverse tasks. If computers could learn some of the same general visual features before learning a complex RL task, then the RL task could be solved faster.

The idea of learning knowledge in one domain and reusing it in another is not new. Transfer learning (TL) is a discipline of machine learning that attempts to learn knowledge in a base domain and transfer that knowledge into a target domain. A typically example of TL is a supervised task which has little training data, but there exists a related base domain. This base domain has plenty of training data and it is possible to learn knowledge that is useful for the target task in the base domain. By learning this knowledge in the base domain, the performance can be increased in the target domain.

1.2 Goals and Research Questions

Goal Statement:

Explore the effects of visual pretraining for deep reinforcement learning.

This thesis study the combination of deep RL and TL in a setting where the agent observe the world through visual observations. The goal is to learn useful visual features in a base domain and then transfer them to a RL agent. By learning visual features in the base domain, the RL task will hopefully be easier to solve. This should result in lower training time and possibly higher performance.

Specifically, agents will be trained to play Atari 2600 games. The agents will learn to act based on screenshots and the in-game rewards. The simulator that enables this is called the Arcade Learning Environment (ALE). ALE makes is possible to test the approach for diverse and challenging games.

Research Question 1:

Can autoencoders learn visual features that are useful for Deep Q-Network agents?

Potentially, any visual task, that trains a neural network, could be used as the base task. However, by choosing an unsupervised base task, the approach can work with any visual data, e.g. real images or Atari screenshots. Autoencoders (AE) are neural networks trained by unsupervised learning in order to reduce the dimensionality of the input data. AE learns a mapping from a high dimensional input space to a low dimensional representation of the input, also called encoding. For visual tasks, this mapping requires the AE to extract visual features from input images in order to encode the images efficiently. Given AEs ability to learn visual features without a supervised training set, they are ideal for this research.

The RL problem will be solved by using Q-learning to training a deep convolutional neural network (CNN). This network is referred to as deep Q-network (DQN). This approach has previously been shown to learn human level policies for Atari games.

Research Question 2:

How does the training set affect the autoencoders ability to learn general visual features?

As mentioned earlier, AEs can learn based on any image. Ideally one would select a training set so that the AE can learn general visual features that are useful for multiple games. On the other hand, if the AE is trained on screenshots from a single game, the AE could learn game specific features and thereby increase the benefits of the pretraining.

Examples of a game specific features are the position and movement direction of key objects such as the player, projectiles, and hostile entities. General features will not be that specific, they will typically focus on detecting edges at particular angles and motion.

1.3 Research Method

In order to answer the research questions, a training system was built using Theano. This system is able to train DQN agents and AEs. In addition it can perform TL between the domains. Evaluation of the system is mainly done by plotting the game score as a function of training steps for multiple Atari games. This allows comparison between standard runs and runs which utilize visual pretraining.

1.4 Main Literature

This work mainly builds on Mnih et al. (2015). The key difference is a focus on reduction in training time in this project. Background information on reinforcement learning is based on Sutton and Barto (1998, 2015). Bishop (1995) and Bengio and Courville (2016) have been used for background information about neural networks.

1.5 Thesis Structure

The rest of the thesis is organized as follows. Chapter 2 gives an introduction to RL and neural networks. That include the basics terminology, temporal difference learning, data efficient RL, convolutional neural networks, the application of neural networks as approximation functions and AEs are covered. Related works are treated in chapter 3. This chapter covers research into reinforcement learning, neural networks, and TL. The main focus of this chapter is on different versions of the DQN algorithm. Chapter 4 describes the DQN algorithm that is used for this work and details how TL is used to initialize DQN agents. Results and discussion are given in chapter 5. While the conclusion with feature work is covered in chapter 6.

Chapter 2

Background

2.1 Reinforcement Learning

Reinforcement learning (RL) is one of the main classes of machine learning. RL deals with situations where a software agent interacts with an environment by taking actions. Based on the actions the agent receives rewards. The goal for the agent is to learn which actions will yield the largest cumulative reward.

A simple example where RL can be applied, is a 2D map where the agent attempts to navigate to a goal state. When the agent reaches the goal state it will receive a reward. In this example the state is the coordinates of current position and the action set consists of four actions; up, down, left, and right.

2.1.1 General Framework

Basic Terms

In the standard reinforcement learning setting, one consider problems which can be formulated as Markov decision processes. In these processes agents can observe the state of the environment $s \in \mathcal{S}$. Based on the observations the agent performs an action $a \in \mathcal{A}$. This changes the state of the environment into a new state $s' \in \mathcal{S}$. In addition the agent receives a reward $r \in \mathbb{R}$. These interactions take place in discrete time steps $t \in \mathbb{N}$. To specify a state, action, or reward at a particular time step, the capital letter followed by t subscripted is used, S_t, A_t, R_t. Figure 2.1 illustrates how agents interact with the environment.

States in the RL framework can be ether finite or continuous. However it is important to note that some methods are not capable of handling continuous states. The same methods

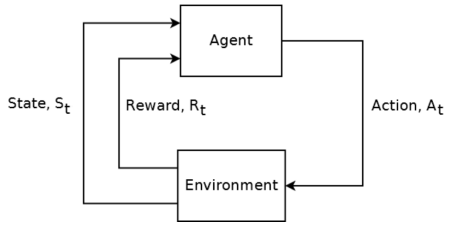


Figure 2.1: Shows the flow of information in typical RL.

have problems when handling large state spaces. A lot of effort has been put into working with large state spaces but this is still an active research area. Another important note is that the state variable does not have to describe the environment fully. This means that RL works on fully and partially observable environments.

The action set must be a finite set and the reward has to be a real number.

Environment Dynamics

As mentioned above, we are considering problems that can be formulated as a Markov decision processes. In a Markov decision process, the transition between states must satisfy the Markov property. This property states that the current state is conditionally independent of the sequence of states and actions leading up to the current state given the previous state and action. Mathematically it can be stated as

$$p(s', r) = Pr\left\{R_{t+1} = r, S_{t+1} = s' \mid S_0, A_0, S_1, A_1, ..., S_{t-1}, A_{t-1}, S_t, A_t\right\} \quad (2.1)$$

$$\downarrow Markov\ property$$

$$p(s', r \mid s, a) = Pr\left\{R_{t+1} = r, S_{t+1} = s' \mid S_t, A_t\right\} \quad (2.2)$$

Another way to understand the Markov property is that the transition between states is only affected by the current state and the action being performed. If there are two paths that lead to the same state, you can follow either one without affecting the future rewards. However, one path might yield higher rewards, when you are following it, than the other.

$p(s', r|s, a)$ gives the one-step dynamics of the environment. It specifies the probability distribution for the next state and reward, given a state action pair. This can be used to derive other properties of the system such as,

$$r(s, a) = \mathbb{E}\left[R_{t+1}|S_t = s, A_t = a\right] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s', r|s, a) \quad (2.3)$$

$$p(s'|s, a) = Pr\left\{S_{t+1} = s'|S_t = s, A_t = a\right\} = \sum_{r\in\mathcal{R}} p(s', r|s, a) \quad (2.4)$$

$$r(s, a, s') = \mathbb{E}\left[R_{t+1}|S_t = s, A_t = a, S_{t+1} = s'\right] = \frac{\sum_{r\in\mathcal{R}} rp(s', r|s, a)}{p(s'|s, a)} \quad (2.5)$$

2.3 is the expected reward of taking action a when in state s. 2.4 is the probability of transitioning into state s' when taking action a in state s. 2.5 is the expected reward of going from state s to state s' by taking action a.

Goal

In a Markov decision process the agent attempts to maximize its discounted future rewards. This can be formulated as

$$G_t = \sum_{t=t'}^{T} \gamma^{t'-t} r_{t'} \tag{2.6}$$

Here γ is the discounting factor. This parameter influences how far-sighted the agent thinks. A small value makes the agent prioritize instant rewards instead of attempting to work towards large future rewards. The only meaningful values are $\gamma \in [0, 1)$. With a negative γ the agent may attempt to avoid future rewards and if γ is 1 or larger the agent has no incentive to take rewards at this time step.

Policy

Another important concept in RL is the policy π. A policy maps from a state to an action. It can be deterministic $\pi(s)$ or stochastic $\pi(a|s)$. A policy is defined as optimal if it maximizes future rewards, see e.q. 2.6. Agents following the optimal policy will always make the best decision. In order to achieve the goal of maximizing feature rewards, RL attempts to learn the optimal policy.

Value Functions

When we have a policy we can start to talk about the state-value function $v_\pi(s)$. This is the expected reward of following policy π when standing in state s. Similarly we have the action-value function $q_\pi(s, a)$. This is the expected return of taking action a in state s and then following the policy π thereafter.

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_{t'=0}^{\infty} \gamma^{t'} R_{t+t'+1} | S_t = s \right] \tag{2.7}$$

$$q_\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t'=0}^{\infty} \gamma^{t'} R_{t+t'+1} | S_t = s, A_t = a \right] \tag{2.8}$$

The optimal policy is written as $v_*(s)$ and $q_*(s, a)$. Similarly when estimating 2.7 and 2.8 we write $V_t(s)$ and $Q_t(s, a)$. The capital letter indicates that this is a random variable. t is subscripted because this is the estimate at time t.

Value functions are typically represented in two ways; look up tables or approximation functions. Look up tables store one value for each state or state-action pair. With large state

spaces, this method requires much memory and is therefore impractical. In these situations it is common to use an approximation function. They reduces the memory footprint and enables generalization between states. Section 2.2.3 discusses how neural networks can be used as approximation functions in a RL setting. In this chapter, value functions will be represented as tables, because of its simplicity.

Bellman Equation

2.7 and 2.8 is the mathematical foundation for many reinforcement learning algorithms. However, in the current formulation they are difficult to use. The first step in transforming them into an algorithm is to make use of the Bellman equation. This equation states that a decision process can be defined recursively. Meaning that a decision process can be separated into two parts. An initial choice, followed by a new decision problem. Given the bellman equation the state-value and action-value function can be written,

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{2.9}$$

$$\begin{aligned} q_\pi(s, a) &= \sum_{s',r} p(s', r|s, a)[r + \gamma q_\pi(s', a')] \\ &= r(s, a) + \sum_{s'} p(s'|s, a)\gamma q_\pi(s', a') \end{aligned} \tag{2.10}$$

It is important to note that the Bellman equation only holds if we are modeling a Markov decision process.

In the case with deterministic policies and environments 2.9 and 2.10, can be simplified to more intuitive equations,

$$v_\pi(s) = r(s, \pi(s), s') + \gamma v_\pi(s') \tag{2.11}$$

$$q_\pi(s, a) = r(s, a, s') + \gamma q_\pi(s', a') \tag{2.12}$$

The reward for the initial choice is represented by the first term. The second term represents the remaining problem.

Model vs Model Free

In order to understand the difference between model and model free RL one must first understand what a model is and what it enables the agent to do. A model is a simulation of the environment. An agent with access to a model can simulate an action and observe the next state without performing the action in the environment.

This leads to one of the important differences between state- and action-value functions. A state-value function cannot in itself solve a decision problem. Agents needs a model to

simulate actions before it can estimate the value of actions. With action-value functions, the agent can loop through all actions and check which action yield the highest value estimate. Therefore it is possible to solve the decision problem without a model.

Since action-value functions are model free, they can be used in situations where the model is unknown. For real world application the model is often unknown. For this reason, model free RL with action-value functions, are used in the experiments. Section 2.1.2 will only address temporal difference learning with action-value functions.

Exploration vs Exploitation

When a policy determines an action, it must often choose between exploring new unknown areas of the state space or exploit already learnt knowledge. Exploration techniques are an important aspect of RL, because RL can guarantee to learn the optimal policy if all areas of the state space are adequately explored. In practice, computational limitations makes it impossible to visit every part of the state space. RL algorithms must therefore balance exploration with exploitation of previously learnt knowledge.

A simple policy which balances exploration and exploitation is the ϵ-greedy policy. ϵ is the probability for choosing the greedy option, otherwise a random action is chosen. The greedy option means to take the action recommended by the value function. The ϵ-greedy policy can guarantee convergence to the optimal policy if there is an infinite number of training cases.

2.1.2 Temporal Difference Learning

Temporal Difference (TD) is a technique used to learn value functions. There are other alternatives to TD, like dynamic programming and Monte Carlo. However, TD is the most common technique used in RL. TD is often viewed as a combination of dynamic programming and Monte Carlo.

TD, like Monte Carlo, is able to perform model free learning. This is done by exploring the state space with some policy and observing which state-action pairs receive rewards. The difference between TD and Monte Carlo is that Monte Carlo needs a complete episode before updating the value function. TD solves this problem by using bootstrapping. Bootstrapping refers to a technique where previously learnt action-values are used to estimate action-values. This technique is also used in dynamic programming. However dynamic programing requires a model of the environment in order to function.

By combining Monte Carlo and dynamic programming, TD is able to learn from sample interactions with the environment. These interactions are called transitions and are structured into quintuples; state, action, reward, next state, and next action (s, a, r, s', a'). When learning an action-value function, the value of a state-action pair should be equal to the reward plus an estimate of the next state-action pair, see equation 2.13. This formulation is possible because of the Bellman equation. SARSA and Q-learning are two

different TD methods. The difference between them lies in how the value of the next state is estimated.

$$Q(s, a) = r + \gamma \, \text{estimate}(s', a') \tag{2.13}$$

SARSA

SARSA takes its name from the quintuple (s, a, r, s', a'). All elements in this quintuple are used in the update rule,

$$Q(s, a) = Q(s, a) + \alpha \left(r + \gamma \, Q(s', a') - Q(s, a) \right) \tag{2.14}$$

Note that, 2.14 is for updating a look up table. $\gamma Q(s', a')$ is a discounted estimate of the next state's value, s'. SARSA uses the next action, a', in this estimate, therefore it is on-policy. This means that SARSA learns the action-value function for the policy being followed, q_π. Pseudo code for SARSA is given in algorithm 1.

Initialize $Q(s, a)$ arbitrarily
Set $Q(\text{terminal-state}, \cdot) = 0$
foreach *episode* **do**
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  foreach *step in episode* **do**
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) = Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))$
    $S \leftarrow S'$
    $A \leftarrow A'$
  end
end

Algorithm 1: Pseudo code for SARSA.

Q-Learning

Where SARSA use a' to estimate s' value, Q-learning ignores a' and rather chooses the action which maximizes s' value. Q-learning does not care about the policy which is performed, therefore it is off-policy. This enables Q-learning to directly approximate the optimal action-value function, q_*, independently of the policy being followed. A drawback is that this may lead to overestimations of the action-value. The update rule for Q-learning is given by,

$$Q(s, a) = Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right) \tag{2.15}$$

A description of Q-learning is given by algorithm 2.

Initialize $Q(s, a)$ arbitrarily
Set $Q(\text{terminal-state}, \cdot) = 0$
foreach *episode* **do**
  Initialize S
  foreach *step in episode* **do**
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) = Q(S, A) + \alpha(R + \gamma \max_a Q(S', a) - Q(S, A))$
    $S \leftarrow S'$
  end
end

Algorithm 2: Pseudo code for Q-learning.

An example which illustrates the difference between SARSA and Q-learning is cliff walking. In this example an agent moves around on a two-dimensional map, from a start position to a goal position. Every transition between states receives a reward of -1, thus the optimal policy will take the shortest path to the goal. Additionally there is a cliff region. Agents entering this area will be sent back to the start position and receive a reward of -100. If the policy being executed is the ϵ-greedy, with $\epsilon = 0.1$. SARSA will find a safe path away from the cliff while Q-learning will find the optimal path. This happens because SARSA learns the action-values of the policy that the agent follows. Since the policy executes random actions it is important to stay away from the cliff in order to minimize the chance of walking off the cliff. Q-learning on the other hand, learns the optimal policy independently of the policy being followed. During training, Q-learning receive a lower score than SARSA because it falls of the cliff more often. However if the trained agents are tested with a greedy policy, Q-learning will beat SARSA because it follows the optimal policy. See figure 2.2 for a graphical illustration.

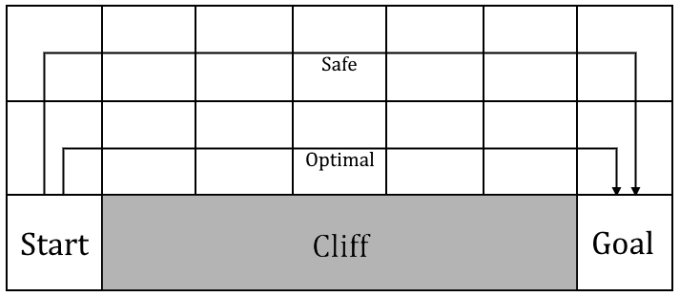


Figure 2.2: Shows show SARSA and Q-learning learn different policies for the cliff walking example.

2.1.3 Data Efficient Reinforcement Learning

Data efficiency is a term which says something about the amount of data needed to achieve a certain level of performance. If two algorithms achieves the same performance but the first uses less training data, then the first algorithm is more data efficient than the second algorithm. Data efficient reinforcement learning (DERL) are techniques that increase data efficiency by storing and reusing transitions. For Q-learning a transition consists of four elements: state, action, reward, and next state. This is all the information needed to perform an update of the value function. When a transition is reused, it contribute to multiple updates of the value function.

Generally DERL has several advantages over standard RL. First, data efficiency is increased because each transition is used multiple times. Executing actions in an environment are often computationally expensive. Thus, can be very beneficial to reduce the number of interactions as much as possible. Second, there are often strong correlations between subsequent transitions. This is a problem because many approximation functions, for instance neural networks, assume that the data is independent. With DERL it is possible to randomly sample from D and thereby mitigate the consequences of these correlations. Third, different areas of the state space can have ”local” optimal policies. With standard RL the learnt policy might oscillate between ”local” polices and thereby not converge. With DERL, transitions from all areas of the state space can be used in policy updates. This counteracts oscillation effects.

The next sections discuss two different DERL techniques, batch reinforcement learning and experience replay. They both store transitions in a dataset, D, but they use D differently.

$$D = \{(s_t, a_t, r_t, s_{t+1}) | t = 1, 2, ..., N - 1\} \tag{2.16}$$

Batch Reinforcement Learning

Batch reinforcement learning (BRL) has two phases, data collection and learning. BRL repeatedly goes through the two phases until a satisfactory policy has been found. During the data collection phase, transitions from the environment are generated by following an arbitrary policy. The policy is typically a previously learnt policy. In the learning phase, machine learning is used to learn the optimal policy for D and each transition can be used multiple times.

Note that during the learning phase, D can be converted to a supervised dataset. This allows supervised learning algorithms to learn the optimal policy for D. If D contains a sufficient amount of transitions, from all relevant areas of the state space, then a supervised algorithm can solve the original RL problem. See section 3.11 for details.

Initialize arbitrary policy π_0
N = 0
while *Stopping condition not reached* **do**
    Use pi_N to sample D_N from the environment
    Learn policy π_N based on D_N
    $N = N + 1$
end

Algorithm 3: Pseudo code for BRL, where previous learnt policy is used to generate D.

Experience Replay

Experience replay (ER), in contrast to BRL, does not separate data collection from learning, rather it simultaneously generate data and trains on this data. As for BRL, there are no special requirements for the policy which generates the transitions, but a near optimal policy will generate more relevant transitions than a random policy. When training, a subset of D is sampled and used to update the value function. A simplest way of doing this, is to uniformly sample D. However, there exists more sophisticated sampling methods which attempt to select the most important transitions. Schaul et al. (2015) introduces such a sampling method and it is discussed in section 3.5.

D can grow impractically large if all transitions are stored. The main problem is that D requires too much memory. In addition the stored transitions may be too old. This is a problem because the agent may have learnt all it can from the transitions or because transitions are sampled from areas of the state space which the current policy avoid. For these reasons it is common to implement a sliding window replay memory. With this technique, D is initialized to a fixed size. When D is full, new transitions overwrites the oldest transitions.

Another consideration to account for when using ER, is the balance of data generation versus training. If too much data is generated, than data efficiency is reduced. On the other hand, with insufficient amounts of data, the learning algorithm will only work with transitions it has observed before. This will reduce the learning speed and in the worst case scenario it causes overfitting. To balance this, one can tweak the frequency of training and the size of the subset used for updating the value function.

There are two drawbacks of ER. First, the memory requirements can be substantial. This is a problem even with sliding window. For example, the replay memory used in van Hasselt et al. (2015) requires approximately $26GB$. Second, ER only works with off-policy RL. ER cannot be used with on-policy RL because the transitions in D are generated by many different policies and it is not possible to learn one value function for all of them.

2.2 Artificial Neural Network

Artificial neural networks (ANN) are currently one of the most popular machine learning techniques. They achieve state of the art results on diverse tasks. Among the examples are image classification (Krizhevsky et al., 2012) and speech recognition (Graves et al., 2013)

This chapter starts by giving an introduction to neural networks (2.2.1), then moves on to convolutional neural network (2.2.2), which is a network architecture often used in visual computational problems. The last two sections discuss two applications of ANNs. First, how ANNs can be trained as action-value approximation functions (2.2.3). Second, unsupervised dimensionality reduction with AEs (2.2.4).

2.2.1 General Framework

ANNs are biologically inspired models capable of doing computations. Networks consists of artificial neurons and weighted connections between them. The neurons are capable of doing simple calculation, typically summing its inputs and calculating an output based on an activation function. The output is sent through connections, where it is multiplied with the connection weight, to neighboring neurons.

There are many variations of ANNs. In this section multilayer perceptrons (MLP) will be addressed. MLP is a common and simple way to structure a neural network, where neurons are arranged into layers. These layers are called dense or fully connected layers.

MLPs starts with one input layer which is followed by one or more hidden layers. The last layer is the output layer. Connection are only allowed from preceding to the succeeding layers. Mathematically, connections are expressed as a matrix W^k, representing the connections between layer $k-1$ and k. By multiplying the activation vector h^{k-1} by the matrix you get a vector representing the summed input for each neuron in layer k. Note that the input vector is the first hidden activation. $h^0 = x$. It is usual to add a bias vector, b^k, to the summed input. This increases the expressive power of neurons. To get the activation for layer k, the summed input plus the bias is sent through an activation function, ϕ. The input is propagated through the entire network, producing an output vector $\hat{y}$. Propagating the input to the output layer is known as the forward pass. 2.17 expresses the transition from one layer to the next. The complete forward pass is written as 2.18. Here θ represents all parameter for the network.

$$h^k = \phi(W^k h^{k-1} + b^k) \tag{2.17}$$

$$\hat{y} = f(x;\ \theta) \tag{2.18}$$

The activation function can be any function, but nonlinear functions make the network a universal function approximator. This means that a network with one hidden layer and sufficient hidden neurons can approximate any function. If the network is trained with backpropagation, then the activation function must be continuously differentiable.

Typically activation functions are the logistic function 2.19, tanh 2.20 and rectifier 2.21, these are nonlinear and continuously differentiable.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.19}$$

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1} \tag{2.20}$$

$$relu(x) = \max(0, x) \tag{2.21}$$

Training of MLPs are usually carried out by the backpropagation algorithm. The basic idea is to present the network with pairs of input and desired output, (x, y). The input is propagated through the network, resulting in an output vector, $\hat{y} = f(x; \theta)$. A cost or loss function calculates the difference between the target and the output vector, $J(y, \hat{y})$. This function is then partially differentiated with respect to the weights resulting in a gradient, see 2.22. This gradient is subtracted from the weights in order to reduce the value of the cost function, see 2.23. Propagating errors backwards is known as the backward pass. α is a learning rate.

$$\nabla_\theta J(y, \hat{y}) = \frac{\partial J(y, f(x; \theta))}{\partial \theta} \tag{2.22}$$

$$\theta_{i+1} = \theta_i - \alpha \nabla_\theta J(y, \hat{y}) \tag{2.23}$$

2.2.2 Convolutional Neural Network

Convolutional neural networks (CNN) are a type of ANNs which in recent years has pushed the boundaries of what is possible with machine learning. They work exceptionally well for visual task but are also used for other tasks such as audio and linguistic tasks.

The key difference between CNNs and MLPs is that convolution calculates the transition between one or more layers in CNNs. This transition is expressed in equation 2.24 and replaces the matrix multiplication from equation 2.17. By using convolution the number of parameters can be dramatically reduced which allows for deeper network architectures. In addition to convolutional layers, pooling layers are often used in CNN. Both will be discussed in this section.

$$h^k = \phi(h^{k-1} * W^k + b) \tag{2.24}$$

Convolutional Layer

Convolution is a mathematical operation in which two functions produce a third function, see equation 2.25.

$$s(t) = (x * w)(t) = \int x(a)w(t-a)\,da \tag{2.25}$$

Convolution can be viewed as the overlap between two functions when sliding one over the other. 2.25 is defined for continuous functions. In computer science most data are discrete therefore a discrete version is used, see 2.26.

$$s[t] = (x * w)[t] = \sum_{a=-\infty}^{\infty} x[a]w[t-a] \tag{2.26}$$

2.26 applies to one-dimensional data from $-\infty$ to ∞. Potential use cases for the equation is convolution between discrete audio waves. However most convolutional networks handle two-dimensional image data. 2.27 is the extension of 2.26 for two-dimensions. I is the image and K is the kernel or filter.

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[i-m, j-n]K[m,n] \tag{2.27}$$

Note that convolution can be extended to any number of dimensions.

Edge detection in image processing is an example of how convolution can extract meaningful features from an image. In image processing a common way to do edge detection is to convolve a Sobel filter with an image, see figure 2.3. This example illustrates how a small kernel convolved with an image can result in useful feature extraction.

$$K = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.28}$$

By applying a single kernel it is possible to extract one feature from an image. However in CNNs there is a need to extract multiple features. Typically this is done by applying multiple filters to the same image. The convolution operation applied in CNNs will therefore project an image into a three dimensional tensor. The tensor has two spatial dimensions and one dimension represents the filter dimension. Three dimensional tensors are also useful when handling color images, where the third dimension is the color space. In CNNs each layer has a collection of filters that is applied to the neurons in this layer. The collection is typically called a filter bank.

Another important element of the convolutional operation used in CNNs is the stride. This specifies how often the filters are applied to the input. With stride 1, the convolution operation will be applied to every neuron in the input. With stride 2, the convolution will be applied to ever second input neuron. This causes the output to be half the size of the input.

The use of convolution instead of matrix multiplication gives CNNs an advantage over MLPs. Some of the most important differences can be explained by these terms: sparse

(a) Original image **(b)** Edges

Figure 2.3: Shows how edges can be detected by convolving a Sobel filter, 2.28, with an image. The strongest response from the filter is when the left side is ”off” while the right side is ”on”. So the filter detects vertical edges.

interactions, parameter sharing, and invariance to location. These properties and how they benefit CNNs will be discussed in depth in the next paragraphs.

Neural networks are said to have sparse interactions if neurons in preceding layers are connected to a subset of the neurons in the succeeding layer. This is a property of CNN if the kernel is smaller than the input image. Kernels are typically several magnitudes smaller than the image. This ratio strongly reduce the number of parameters in a neural network, see figure 2.4. Benefits are a reduction in the computational cost and the memory footprint of the network. This will reduce the expressive power of the network. However this is typically not a problem, because neurons are indirectly connected, see figure 2.5.

Parameter sharing is a property of computational models where one parameter is used in several computations. This occurs in CNN because the same kernel is applied to all pixels in an image. Parameter sharing reduces the memory requirements of CNN further, while the computational complexity remains the same. Figure 2.3 illustrate why this is useful. The small three by three kernel is able to extract edge information in the entire image. If the same information should be extracted without parameter sharing you would need nine parameters for each pixels in the image. In addition to the increased memory use, each of the parameters have to be trained. More parameters, means an increased training time. Moreover the training set typically has to be larger, in order to determine the parameters. Also note that the Sobel filter can extract edge information from any image.

Invariance to location is a property which states that the position of an input signal does not affect the output signal. In the edge detection example, the input can be shifted by translating the image 20 pixels to the right. When edge detection is applied, the detected edges will be identical to the original edges only shifted 20 pixels to the right. This is a property of convolution. As a result, CNNs are very robust in regards to translation in

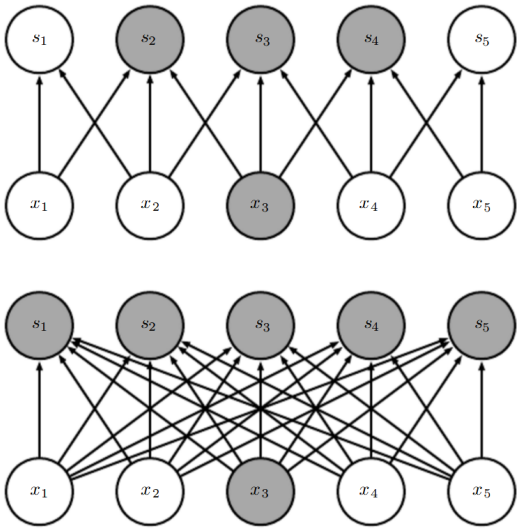


Figure 2.4: Shows the difference between sparsely connected and fully connected hidden layers. The upper figure shows a sparsely connected hidden layer. Here neuron x_3 is directly connected with three neurons. The bottom figure shows a fully connected hidden layer. Here, x_3 is direct connections to the entire hidden layer. The figure is taken from Bengio and Courville (2016)

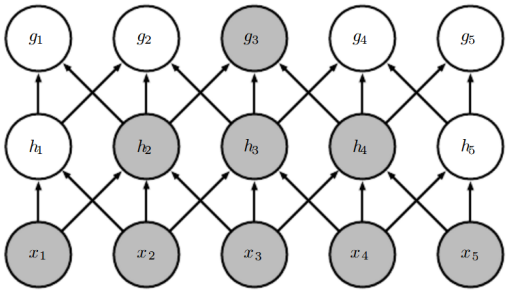


Figure 2.5: Shows how neuron g_3 is indirectly affected by all neuron in the input layer. The figure is taken from Bengio and Courville (2016)

images.

Pooling Layer

Pooling layers are another type of layers often used in CNNs. It further improves the invariance with respect to translation. This type of layers should be used if recognizing that a feature exists is more important than knowing the exact position of this feature. For instance, a network can attempt to classify images of people into two classes, male and female. The network could then detect whether the hair is short or long. Exactly where this feature is positioned in the image does not matter, only whether it is present or not.

Pooling layers replace the response of a neuron with a statistical summary of neighboring neurons. Examples of pooling layers are max and averaging pooling layers. These layers look at the neighboring neurons and either calculate the max or average.

2.2.3 Neural Networks as Approximation Functions

Section 2.1 considered RL with look up tables, but for real RL problems the state space is often too large to store a value estimate of each state. In such cases an approximation function can be used to compute an estimate of the value. Another advantage of approximation functions is that they can generalize between states, which means that not all states must be explored in order to learn a policy. A problem with nonlinear approximation functions are that they can be unstable.

Neural networks have been used as approximation functions for a long time (e.g., see Tesauro, 1995). It has been an active research area for two decades (e.g., see Riedmiller, 2005; Lange and Riedmiller, 2010). Currently deep neural networks are able to achieve state of the art results in reinforcement learning problems (e.g., see Mnih et al., 2015).

Training

In order to train the network, an input and target output must be defined. For Q-learning, the input is defined as the state-action pair, see eq. 2.29. The target is defined as the received reward plus a discounted estimate of the next states value, see eq. 2.30. This equation is referred to as the target equation and it is based on the Q-learning update rule, see eq. 2.15.

$$x_t = (s_t,\, a_t) \tag{2.29}$$

$$y_t = r_t + \gamma \max_a Q(s_{t+1},\, a;\, \theta) \tag{2.30}$$

$\hat{y}_t$ is the action-value estimate for a state-action pair are computed based on the neural network, see eq. 2.31.

$$\hat{y}_t = Q(s_t,\, a_t;\, \theta) \tag{2.31}$$

Training is done based on the cost function described by equation 2.32. This is the mean squared error loss between the target and the action-value estimate.

$$\begin{aligned} J(s_t,\, a_t,\, r_t,\, s_{t+1};\, \theta) &= (y_t - \hat{y}_t)^2 \\ &= (r_t + \gamma \max_a Q(s_{t+1},\, a;\, \theta) - Q(s_t,\, a_t;\, \theta))^2 \end{aligned} \tag{2.32}$$

The next sections discusses alternative network architectures used to approximate action-value functions.

State-Action Input with Single Action-Value Output

Action-value functions are defined to take a state action pair as input and should return a single value which estimates the action value. This approach simply encodes the state action pair and pass it to the network as input. The output is a single neuron which is the estimate. An advantage of this approach is its simplicity. The drawback is that multiple feed forward passes are required to select an action, see equation 2.33.

$$a_t = \operatorname*{argmax}_a Q(s_t,\, a;\, \theta) \tag{2.33}$$

State Input with Multiple Action-Value Output

With this architecture, only the state is passed to the network. Each action is represented with a neuron in the output layer and the network computes a vector of action values. This architecture makes it possible to calculate all action-values for a state with a single feed forward pass. Thus the computational cost of action selection, eq. 2.33, is decreased. However, an additional step must be performed in order to convert the vector into a single value when computing $Q(s_t,\, a_t;\, \theta)$. This is done by equation 2.34. The equation selects the specific action, a, from the output vector $f(s_t;\, \theta)$. When backpropagating through this function all output neurons, except the one representing the chosen action, are ignored.

$$Q(s_t,\, a_t;\, \theta) = f(s_t;\, \theta)_a \tag{2.34}$$

2.2.4 Autoencoders

Autoencoders (AE) are neural networks which take some input and attempts to replicate it in the output layer. The input is encoded to a hidden representation $h = f(x)$. By choosing the dimensionality of the hidden representation to be smaller than the input

dimensionality, the AE must learn to extract features and encode them efficiently. The hidden representation is used to reconstructed the original input $r = d(h)$. For optimal AEs, the reconstruction is the same as the input, $r = x$. Mean squared error is typically used as the loss function when training AEs, see eq. 2.35.

$$L(x, r) = (x - r)^2 \tag{2.35}$$

When training AEs, it is possible that they learn just to copy data from the input layer to the output layer. In such cases the AEs has learnt the identity function and they are typically not useful. As mentioned before, the dimensionality of the hidden representation can be chosen so that the AE will not learn the identity function. There are other techniques which mitigates the risks of learning the identity function. One such technique corrupts the input before it is passed to the network. With this technique, the AE cannot copy the data because the input data is not the correct output. Rather it must learn the structure in the input space in order to reconstruct the original data. AEs trained with this technique are called denoising AEs.

2.3 Summary

This chapter attempts to give introduction to the field of RL and ANN. By combining these fields, very impressive algorithms can be created. The DQN algorithm is an example of this. This algorithm use Q-learning (section 2.1.2) to train a deep CNN (section 2.2.2) to estimate action-values (section 2.1.1). In addition, ER (section 2.1.3) is used to stabilize the learning process. When all of these elements are combined, human level performance can be achieved on Atari games.

The next chapter focuses on the DQN algorithm and improvements to it. These improvements are achieved by for instance refining the target generation from section 2.2.3 or prioritizing the sampling of the replay memory. A firm understanding of the background is needed to understand how these improvements work.

Chapter 3

Related Work

The first section of this chapter covers TD-gammon. This is a computer program that plays backgammon and is an early example of how RL combined with ANNs can achieve high level of performance on games. Section 3.2 presents the ALE which is a simulation tool that allows computers to play Atari games. The DQN algorithm is introduced in section 3.3. This algorithm improves state-of-the-art performance on Atari games. Section 3.4 to 3.8 discuss work which builds on and improves the DQN algorithm. Section 3.9 and 3.10 present two papers discussing how visual features in CNNs tend to be general. In addition they show how CNNs can be used for TL and feature extraction. Section 3.11 covers some examples of batch RL.

3.1 TD-gammon

There is a long history of applying RL to games. The most famous example of this is Tesauro (1995). Tesauro's paper describers a computer system that plays backgammon. It was able to compete against top ranked backgammon players and discovered several new strategies that were adopted by human players. This is an early example of RL outperforming other approaches like supervised learning, brute force search, and heuristic positional judgment.

Tesauro used a version of temporal difference learning. His approach uses an ANN as a state-value function. The function is trained to approximate the value of a board state. During game play, a model that knows the rules of backgammon simulates all legal moves. The network evaluates all resulting board states and the action leading to the highest scoring is selected. Simulations of the board is needed because a state-value function cannot be directly used to generate a policy. If an action-value function is utilised, simulations are avoided because the agent is able to learn the rule of the game while playing.

Several properties of backgammon were discussed in an attempt to understand why the learning algorithm was so successful.

First is the stochasticity of the environment. Stochasticity helps with exploration, prohibits cycles, and smooths out the state-value function. Exploration is improved because the same state-action pair can lead to multiple states and rewards. Cycles in the board position will at some point stop because of the randomness. This ensures that the game will terminate at some point, which again ensures that the final reward signal can be propagated backwards. The smoothness of the state-value function, which the neural network is approximating, mitigates the risk of back-propagation converging to a local minimum.

Second, there are simple linear concepts in backgammon that can easily be learnt by neural networks. By using simple concepts, a simple policy can be learnt. Concepts that are more complex can gradually be constructed from simple concepts. When the network recognizes complex concepts, it will be able to use a more sophisticated strategy. A simple concept could be used to recognize a useful board configuration and give high value estimates for states containing this configuration. If this board configuration can become a liability in the end game, a more complex concept can be constructed from the simple concept. This complex concept will account for the time dependency. Multi layered neural networks typically detect simple concepts in the first layer and from them construct complex concepts in the following layers.

This paper gives insight into which Atari 2600 games RL will achieve high performance. The games are expected to have an element of stochasticity and simple concepts which can form an initial policy.

3.2 Arcade Learning Environment

The Arcade Learning Environment (ALE) is a platform for testing control algorithms. ALE is an simulator for the Atari 2600 game console and it has a simple interface that enables software agents to play Atari 2600 games. ALE enables agents to observe the environment, take actions and receive rewards from games. The observations can be screen shots or the internal state of the game.

The goal of ALE is to introduce a standard platform for testing RL algorithms. By introducing a standard it is possible to compare results between different approaches. There has been other standard RL problems (e.g. pole balancing and mountaincar) and libraries but none have the rich variety of problems as ALE. ALE's focus on high dimensional images is also a unique quality.

Bellemare et al. (2013) is written as an introduction to the ALE. Bellemare's paper illustrates the potential use of ALE by having several different algorithms play 55 different games and then compare the game score. Among the algorithms are $SARSA(\lambda)$, as described in Sutton and Barto (1998). There were five different feature construction algorithms used to generate then inputs to $SARSA(\lambda)$. The features did not

contain any domain specific information and could therefore be applied to all the games. The empirical results from these experiments demonstrate the performance of standard RL.

Exact training time is not reported in Bellemare et al. (2013), however there is a limit. In their experiments, $SARSA(\lambda)$ was trained on 5 000 episodes, each one lasting 5 minutes (18 000 frames) or until the game ended. Combined this is over 17 days of game play which is the same as 90 million transitions. 17 days is a long time and this illustrates how slowly standard RL converges on a policy. In comparison humans can understand the game and develop a decent policy within 5 minutes of game play.

3.3 DQN

A more resent paper that uses the ALE is Mnih et al. (2015). This paper demonstrates their RL algorithm's performance by comparing it with a professional human player. In 29 out of 49 games, their agent performed comparably to the professional human player. A comparison of the performance of this algorithm and other improvements is given in table 3.1.

The paper uses a version of Q-learning with a CNN as the action-value approximation function, $Q(s,\ s;\ \theta)$, where θ is the parameters of the network. They named their network, deep Q-network (DQN). The CNN is passed screenshots from the ALE and outputs action-values. This incorporates 'end-to-end' training, which makes use of feature learning. This means that only features that are useful for the problem at hand are learned.

The DQN architecture consists of three convolutional layers followed by a dense layer before the final output layer. A complete description can be found in section 4.1.2. By using a CNN it is possible to use high dimensional images as input for Q-learning. This makes the algorithm more applicable to real world problems which often uses high dimensional data.

Their algorithm is based on Q-learning but it differs from the standard Q-learning in two ways. First, they use experience replay, see section 2.1.3. This technique store transitions in a memory bank, D. Training of the action-value function is based on mini-batches which are uniformly sampled from D. Second, targets are generated by a separate network. This network uses weights θ^- and is referred to as the target network, $\hat{Q}(s,\ s;\ \theta^-)$. Its parameters are copied from the online network every τ steps, so that $\theta^- = \theta$. By introducing the target network, the target equation which is given by equation 2.30, is altered. The new target equation is given by equation 3.1. Both of these changes attempt to improve the stability and mitigate the chance of divergence.

$$y_t = r_t + \gamma \max_a \hat{Q}(s_{t+1},\ a;\ \theta_t^-) \tag{3.1}$$

The most interesting elements of this paper is the fact that they used minimal prior knowledge. Minimal prior knowledge means that they did not customize hyper-parameters for each game, but rather found one set of parameters for all games.

This shows that the method is robust and can be used on diverse tasks. Examples of hyper-parameters are the network architecture and learning rate.

The results of the DQN algorithm is very impressive when compared to a professional human player, but it should be noted that some of the reported human scores are low for a professional human player. I tested breakout and after 5 minutes of training, I achieved scores of 32, 63, 84, 62, and 78 which gives a mean of 63.8. The human score from Mnih et al. (2015) is 31.8. So, I was able to achieve double the score of the professional human player. By searching the web, it seems like humans should be able to achieve scores of approximately 400 points or more. This is the same as the DQN algorithm.

3.4 Double DQN

Further improvements to the DQN algorithm is proposed in van Hasselt et al. (2015). This work builds on Mnih et al. (2015) but focuses on the effects of overestimations in action value functions. Overestimation is a well-known property of Q-learning. However, questions like: How common are overestimates? Can overestimations be harmful to learnt polices? And can overestimations be prevented? is unanswered. van Hasselt et al. (2015) answer these questions and report increased performance on Atari games with their improved version of the DQN algorithm. This algorithm attempts to reduce overestimations and is referred to as Double DQN.

To answer the first question, the paper gives a mathematically prof showing that value functions will overestimate the value of actions. The prof is valid if the function has some estimation errors but on average is correct. Furthermore, they analyse DQN's action values for 49 Atari games. The analysis reveal varying amounts of overestimation in all games. This demonstrates that overestimation is a common occurrence for the DQN algorithm.

Double DQN reduces overestimation by using more accurate targets while training. Standard DQN generate targets with 3.1. The max operation used in this equation, both selects and evaluates actions based on one set of parameters, θ^-. This is somewhat subtle but by rewriting 3.1 into 3.2 this is highlighted. In this equation, selection is explicitly done by the argmax operator, while the outer action-value function only evaluates the value of actions. Since the same parameters are used twice, estimation errors are magnified. Double DQN decouple selection and evaluation by using the online network to select actions while evaluation is done by the target network. The new target equation used by Double DQN is given by 3.3.

$$y_t = r_t + \gamma \hat{Q}(s_{t+1}, \underset{a}{\text{argmax}}\, \hat{Q}(s_{t+1}, a; \theta_t^-); \theta_t^-) \tag{3.2}$$

$$y_t^{\text{DoubleDQN}} = r_t + \gamma \hat{Q}(s_{t+1}, \underset{a}{\text{argmax}}\, Q(s_{t+1}, a; \theta_t); \theta_t^-) \tag{3.3}$$

This simple trick reduces overestimations in DQN and improves the performance in numerous Atari games. This illustrates that overestimations can impair policies and by

reducing overestimations policies can be improved.

3.5 Prioritized Experience Replay

Schaul et al. (2015) improve the performance of the DQN algorithm by prioritizing the sampling of the replay memory. Previously the sampling is done uniformly. By identifying transitions which are important for the policy, the computations resources can be focus on these transitions and thereby reducing the training time.

It is not possible to directly calculate the importance of samples, but the magnitude of TD errors can be used as a heuristic. The magnitude shows whether or not the network was able to predict the correct value estimate. Large magnitudes means that the network incorrectly predicted the value, meaning that the network does not understand this transition therefore this sample is important. Similarly, for small magnitudes, the network is correct in its predictions therefore this samples contains less useful information. If sampling is done greedily based on TD errors, the system becomes prone to over-fitting. To address this problem the paper computes a probability distribution based on the importance and sample from it.

Prioritized experience replay boost the performance of the DQN algorithm and it outperforms standard DQN in 41 of 49 games. In addition, it increases the learning speed. It is currently the best performing algorithm on Atari games, see table 3.1. Note that these improvements are complementary to those of Double Q-learning.

3.6 DRQN

Hausknecht and Stone (2015) investigate how long short term memory (LSTM) helps DQN agents in partially observable markov decision processes (POMDP). A POMDP is a MDP where the agent cannot observe the state directly but has access to observations which incompletely describes the state of the environment. A LSTM network is a recurrent neural network which enables the network to remember information from previous time steps. By using a LSTM network in a POMDP, agents can internally model the environment and use this model when taking actions, thereby actions does not solely depend on imperfect observations.

The method, which is named deep recurrent Q-network (DRQN), generally follows that of Mnih et al. (2015). However, there are two important differences. First, the CNN architecture is changed by swapping the dense layer with a LSTM layer. Second, the random sampling of the replay memory becomes more challenging because LSTM networks contains a state which is used in updates. Therefore, steps must be taken in order to generate an appropriate internal state. In this paper, the problem is solved by first zeroing out the state, then generate it by letting the agent experience n frames leading up to the time step for the update. At this point backpropagation can be used to update the model.

Most Atari games are MDPs if the observed state is a stack of four frames. To test their algorithm they convert the games into POMDPs by blacking out the screen with a certain probability. By controlling this probability, it is possible to control the degree of which the environment behaves like a POMDP or MDP. If this probability is high, then the environment is a POMDP. With a low probability the environment becomes a MDP.

The findings are that DRQN is able to learn proficient policies, in MDPs, with similar performance as DQN, but nether technique is systematically better than the other. In addition, DRQN is better at POMDP even if the DQN agent observers 10 frames. Furthermore, DRQN is more robust to changes in the blackout probability. If training is done in a POMDP but evaluation is done in a MDP, then DRQN performs better than DQN. Similarly, if agents are trained in a MDP and evaluated in a POMDP, then DRQN generalize better. This can be vital for real world applications where the quality of observations might change over time.

3.7 Asynchronous Methods

A new framework for deep RL is introduced by Mnih et al. (2016). The framework allows for on/off-policy and value/policy-based RL while at same time reduce the computational costs of training. The key idea is to asynchronously simulate multiple environments rather than using replay memory. Each environment has an agent which learns from interactions with an instance of the environment. This agent is referred to as a learner and it generates gradients which are used to update a shared model. With asynchronous learners, which simultaneous update the same model, there is a need to synchronize updates. Synchronization is solved with an algorithm called Hogwild! which allows for lock-free updates.

One of the main reasons for using replay memory is to break correlations between samples. Since each learner interact with an independent instance of the environment, then the updates to the shared model is based on data with few correlations. Therefore the need for replay memory is alleviated and it is possible to remove it completely. The new framework significantly reduces the memory requirements by discarding the replay memory. However, replay memory can still be used to increase the data efficiency and could be beneficial if interacting with the environment is expensive.

Another important benefit is that the framework runs on a central processing unit (CPU) instead of a GPU. This means that there is no need for specialized hardware to run the algorithm. Furthermore, when training for the DQN task, it needs half the training time of previous DQN algorithms. It should however be noted that the CPU used for the experiments has 16 cores, which offers considerably more processing power than average consumer grade hardware.

In addition to the asynchronous training framework, the paper experiment with two techniques which differs from the standard DQN algorithms from Mnih et al. (2015). These techniques are LSTM and separate exploration rates for each learner. A LSTM network is, as mentioned in section 3.6, a recurrent neural network which enables the

network to remember information from previous time steps. Mnih et al. (2016) changes the standard DQN architecture, introduced in Mnih et al. (2015), by adding a LSTM layer with 256 neurons between the dense layer and the output layer. As seen in section 3.6, LSTM-networks combined with replay memory is challenging because the internal state of the network must be generated. With this framework this problem is avoided because observations are experienced sequentially rather than randomly sampled. As a result, the internal state is always correct. Separate exploration rates are implemented by initializing each learner its own exploration rate. This means that one learner can focus on exploration while other focus on exploitation. This makes the approach more robust for different problems.

Mnih et al. (2016) tested their framework with four standard RL methods: one-step Q-learning, one-step SARSA, n-step Q-learning, and advantage actor-critic. All these methods was able to learn policies for the DQN and other RL tasks. These methods are described in Sutton and Barto (2015). However, advantage actor-critic stand out as the best performing technique. This method is tested against other DQN techniques and the results can be seen in table 3.1. In the table advantage actor-critic is referred to as A3C which stands from asynchronous advantage actor-critic.

Table 3.1: The table is taken from Mnih et al. (2016) and compares different DQN methods. The scores are normalized to that of a human player. So if the algorithm achieves a normalized score of 100% for one specific game, then it got the same score as a professional human player. Mean and median is based on scores from 57 Atari games. A3C, FF is a feed forward neural network while A3C, LSTM is a recurrent neural network. Gorila and Dueling Double DQN is not discussed in this thesis.

Method	**Paper**	**Mean**	**Median**
DQN	Mnih et al. (2015)	121.9%	47.5%
Gorila	Nair et al. (2015)	215.2%	71.3%
Double DQN	van Hasselt et al. (2015)	332.9%	110.9%
Dueling Double DQN	Wang et al. (2015)	343.8%	117.1%
Prioritized DQN	Schaul et al. (2015)	463.6%	*127.6%*
A3C, FF	Mnih et al. (2016)	496.8%	116.6%
A3C, LSTM	Mnih et al. (2016)	*623.0%*	112.6%

3.8 Action-Conditional Video Prediction

Oh et al. (2015) trains a CNN to predict the next frame based on the previous frames and the current action, for Atari games. This prediction task requires a network with similar architecture to a AE. The CNN start, like an AE, by encoding the input. But rather than just encoding the input, the network combines the encoding with the action and predicts the next frame. Then the CNN attempts to reconstruct the next frame. The encoder consists of 4 convolutional layers and 2 dense layers. The decoder is symmetrical to the encoder, with one exception. It reconstructs a single frame rather than a stack of frames. The CNN is

depicted in figure 3.1. In addition to this feed forward network, the paper experiment with a LSTM network. This paper uses the original ALE frames as input for the CNN. Other DQN techniques typically use preprocessed frames rather than the original ALE frames. Training is done based on a dataset which is generated by a DQN agent.

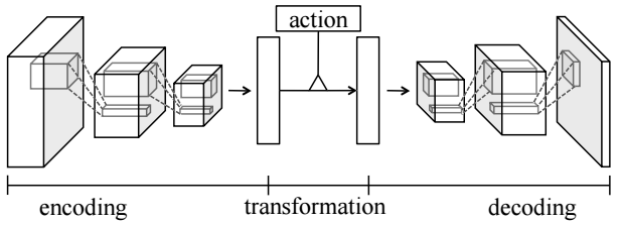


Figure 3.1: Shows the network architecture used by Oh et al. (2015). The figure is take from the paper.

After training, the CNN is capable of predicting accurate screenshots up to 100 steps into the future. In order to do this, the network has to anticipate movement of controlled and uncontrolled objects as well as the interactions between them, such as collisions. This require high understanding of the environment dynamics and by solving it the network demonstrate it has learnt vital information about the game. However the technique struggles with a few cases such as stochastic environments and small object. Stochastic environments are obviously difficult because objects can spawn and move nondeterministically. When it comes to small objects, they are more challenging than large objects, simply because their errors are negligible when they are reconstructed incorrectly.

The paper propose a new exploration technique which depends on the ability to predict the next state. If the agent maintain a collection of previous states, it can check which actions leads to states not in the collection. Thus the agent is able to choose actions leading to unexplored areas of the state space. This exploration technique was tested on several games and produced marginally better performance in most games. However, the exploration lead to over a doubling in score for the game Q*bert. A significant problem with this technique is the supervised training set which is needed to train the prediction network. This set is produced by an agent which was trained with random exploration. So random exploration is still needed. Furthermore, the prediction network might have problems generalizing to new areas of the state space which the random exploration did not explore. In these areas it is particularly important that the exploration technique performs well.

3.9 CNN transfer learning

TL is a technique which attempts to learn knowledge in a base domain and transfer it to a new target domain. For neural networks, one can transfer knowledge by training a network

on the base task and then initialize a new network, for the target task, with weights copied from the base network.

In visual tasks, where a CNN is trained to solve the problem, there are often strong similarities between Gabor filters and filters from the first layer in the CNN. These similarities occur independent of which visual task the CNN solves. Gabor filters are mathematically derived filters designed to detected edges and they are used in image processing. Furthermore, neurons which behaves similarly to Gabor filters can be found in the visual system of humans. Combined, this shows that there are certain general low-level features which are useful in many visual tasks. By using TL it should be possible to learn these features in a base domain and reuse them for different target domains. When it comes to higher levels of visual systems, it is not clear if they extract general features like the first or if features are task dependent.

Yosinski et al. (2014) carry out experiments which investigates how transferable features are in CNNs. In the experiments, an eight layered CNN is trained for image classification on the ImageNet dataset (Deng et al., 2009). ImageNet is spilt into two subset, A and B and each subset represents a different domain. In addition to testing TL between domains, they test cases where the base domain and target domain is the same. Meaning that a network is trained on dataset, e.g. B, then the upper layers are randomized before training restarts on dataset B. To do a thorough investigation they vary the number of copied layers and analyse the difference when freezing versus fine-tuning copied weights. The performance of different networks are shown in figure 3.2.

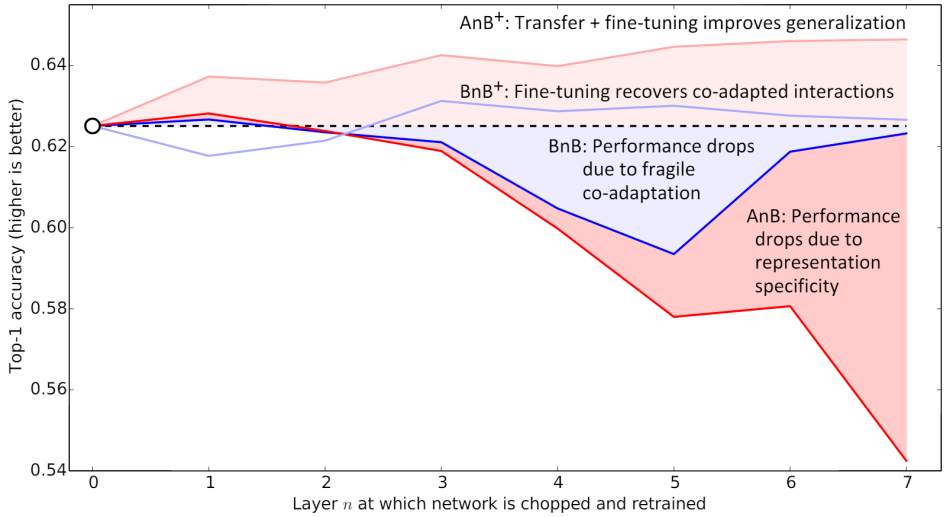


Figure 3.2: Plot of the accuracy of classification on ImageNet. AnB is a run where TL goes from domain A to B. The BnB run is the case where the base and target is the same. n denotes how many of layers are copied. For instance, the three first layers are copied when $n = 3$. The $+$ in AnB^+ and BnB^+ indicates that weights were fine-tuned. The figure is from Yosinski et al. (2014).

Their main finding are that:

First, the first layers contain general features which can easily be transferred by coping. This can be observed in figure 3.2 by looking at the AnB line with $n = 1, 2$. Here the first layers are copied and frozen. Meaning that the same weights are used in both domains. The performance is similar to the base line. Hence the features must be general.

Second, the middle layers contains co-adapted features. Co-adapted layers are subsequent layers which depends on each other. If one layer is randomized and the other is frozen, then performance will drop because the randomized layer cannot learn the co-adopted features by itself. The performance drop in $BnB, n = 4, 5$ is explained by this phenomenon. Note that the co-adopted neurons can be recovered by fine-tuning the weights, BnB^+ and neither the first nor last layers shows signs of co-adopted features.

Third, the last layers contains task dependent features. This can be seen in plot $AnB, n = 6, 7$ where performance drop by nearly 10%. This happens because the last layer detects features that is specialized towards the base task and consequently is not useful for the target task.

Finally, TL can boost performance. This claim is supported by the AnB^+ plot which shows an increase in performance in comparison to BnB^+. These plots are trained for the same amount of time and the only difference is the base domain.

3.10 CNN feature extraction

Recent research has suggested that CNNs can extract generic hige level features from images. These features can be used for a wide variety of visual computational tasks. Razavian et al. (2014) test this hypothesis by using a pretrained CNN for feature extraction and use these features to achieve state-of-the-art results on tasks such as object classification, object detection, attribute detection, and visual instance retrieval. For all of these tasks, the CNN extracts a feature vector and passes it to a task specific machine learning algorithm, e.g. a linear SVM.

The pretrained CNN is the OverFeat network. This is a freely available CNN which was trained for object classification. The feature vector used in these experiments, is the output from the second last layer of the OverFeat network.

Training large CNNs typically requires enormous amounts of training data and computational resources. This paper shows that CNNs can extract general features that are useful for many different tasks. By using feature vectors from CNNs, deep learning can be used in cases with little training data and limited computational resources.

3.11 Batch Reinforcement Learning

Fitted Q-Iteration

An algorithm named fitted Q-iteration (FQI) was introduced by Ernst et al. (2005). This is a batch reinforcement learning algorithm (BRL) where the main idea is to convert RL problems into supervised problems. This enable the use of standard supervised algorithms. FQI can be seen as a function taking two parameters, a set of transitions D (see section 2.1.3) and a supervised regression algorithm. FQI returns a trained regression algorithm that functions as an action-value approximation function.

From the set of transitions, a supervised training set is defined with the state and action as the input. The target is the observed reward plus an estimate of the next state's value. Any supervised regression learner can now be applied to the training set. After the supervised algorithm finishes, targets will be updated with a new estimate of the next state's value. Iteratively the regression learner will be trained and the targets updated until some stopping condition is reached. A typical stopping condition is a max number of iterations. The pseudo code for FQI is given in algorithm 4.

Ernst applied FQI to a series of standard RL problem and achieved excellent results. In his experiments tree based regression algorithms were used.

Input: set of sample interactions D and a regression algorithm ;
Initialize: $N = 0$ and $\hat{Q}_N(\cdot, \cdot) = 0$;
while *Stopping condition not reached* **do**
  $N \leftarrow N + 1$;
  $TS \leftarrow \{ (i^l, o^l) \mid l = 1, 2, ..., \#D \}$;
    $i^l = (s_t^l, a_t^l)$;
    $o^l = r_t^l + \gamma \underset{a}{max} \hat{Q}_{N-1}(s_{t+1}, a)$;
  Use regression algorithm to train $\hat{Q}_N$ with TS ;
end
Return: $\hat{Q}_N$;

Algorithm 4: Pseudo code for fitted Q-Iteration.

Neural Fitted Q-Iterations

Riedmiller tests the FQI framework by using neural networks as the regression algorithm in Riedmiller (2005). Neural networks fit nicely into the FQI because weight from the previous network ($\hat{Q}_{N-1}$) can be reused when training the new network ($\hat{Q}_N$). The specialization of FQI was called neural fitted Q-iteration. Again, the FQI framework shows its ability to achieve excellent results on RL benchmark tasks.

Deep Fitted Q-Iterations

Deep fitted Q-iterations (DFQ) is a framework that is introduced by Lange and Riedmiller (2010). The purpose of this framework is it to improve results on high dimensional sensory input. The main idea is to use an AE to reduce the dimensionality of the data before RL is applied, meaning that the RL algorithm will handle low dimensionality data instead of high dimensional data.

DFQ can be used with any RL algorithm. However, dimensionality reduction requires a set of state observations which is used to train the AE. In batch algorithms, this set is already stored and there is no additional memory requirements introduced by DFQ. Therefore batch algorithms like FQI is preferred.

DFQ was applied to grid-world task where the task is to navigate a maze, from a start position to a goal position. The sensory input is a noisy image of the grid-world. The AE is presented with many screenshots and trained to reduce the dimensionality. After the AE is trained, the RL task is solved by using BRL. A near optimal control policy was learned autonomously without the need for hand crafted feature extraction. Without the dimensionality reduction, the RL problem would be more difficult to solve.

3.12 Structured Literature Review

3.12.1 Search Procedure

When searching for relevant work the reference list of Mnih et al. (2015) was as a starting point. Another way of finding relevant paper was to visit the author's websites and look for further developments to the DQN algorithm.

In addition, the search engine Google scholar was used to look for relevant papers. The search terms was 'Atari 2600', 'DQN', 'transfer learning' and 'reinforcement learning'.

3.12.2 Selection criteria

When selecting paper, there were no strict selection criteria. Rather, several criteria was weight up against each other.

- Published date
- Number of situations
- Trusted publishers (IEEE, nature, ACM)
- Trusted research company (DeepMind)

The date is important because there have been many resonate developments in the last years. This means that some of the older work is not directly relevant for this thesis. The large majority of the reviewed papers is less than 5 years old. The number of situations,

the publisher, and the research company are all used to give an indication about the quality of the work.

3.13 Summary

This chapter reviews multiple versions of the DQN algorithm. Ideas from different paper are combined in the next chapter to form the algorithm that is used in the experiments.

The chapter also covers a few examples which illustrate that the CNNs can learn general visual features. These features can be used for different visual tasks, see section 3.9 and 3.10. Section 3.9 also show how knowledge, learnt by CNNs, can be transferred to a new domain by simply coping weights.

Chapter 4

Method

This chapter starts by describing the DQN algorithm. In short, the DQN algorithm is a model-free and off-policy RL algorithm which is able to learn human level policies for many different Atari games. This is done by training a deep CNN to estimate action-values based on Atari screenshots and the rewards in the game.

There exists numerous different versions of the DQN algorithm. Section 4.1 describes the version of the DQN algorithm used in this project. The algorithm is similar to the version used by Mnih et al. (2015) but elements such as the target generation and exploration is taken from van Hasselt et al. (2015) and Mnih et al. (2016) respectively.

Section 4.2 discuss the training of AE and how TL is used to initialize DQNs with weights from a pretrained AE. The evaluation technique and a description of each test game is given in section 4.3.

4.1 DQN

This section starts by describing the preprocessing steps performed on Atari screenshots before they are used in the DQN algorithm. Section 4.1.2 gives a detailed description of the CNN architecture. The exploration strategy is discussed in section 4.1.3. Section 4.1.4 and 4.1.5 discuss two of the key elements of the DQN algorithm: the use of experience replay and the target generation. The training time is discussed in section 4.1.6. Section 4.1.7 lists the remaining details that are needed to implement the system.

4.1.1 Preprocessing

State observations from ALE are RGB images with size 210×160. The high dimension of the images makes it difficult to learn policies. Therefore a series of preprocessing steps is used in an attempt to reduce the dimensionality without losing crucial information. The first step is to convert the image to gray scale. Then the image is scaleed to the size 84×84. After this, the max of the current frame and the previous frame is selected. This is useful because some games only render foreground objects every other frame, so without this step the observations would flicker. The last step is to stack the m most recent frames, $m = 4$ for the experiments. This makes it possible to detect motion which can be useful for policies. All of the preprocessing steps combined is referred to as the function ϕ and is used for all state observations which are passed to the learning algorithm.

4.1.2 Network Architecture

The network take a state as input and outputs a vector of action values. This architecture is discussed in more details in section 2.2.3, but in short this architecture takes the state as input and computes action-values for all possible actions. This makes it possible to select actions with a single forward pass.

The network is composed of three convolutional layers, one dense layer, and the output layer. The first convolutional layer uses 32 filters of size 8×8 with stride 4. The second uses 64 filters of size 4×4 with stride 2. The last convolutional layer uses 64 filters of size 3×3 with stride 1. All convolutional layers uses one bias node for each filter and ReLU as the activation function. A dense hidden layer with 512 neurons follows the convolutional layers. Again, ReLU is used as the activation function. The output layer uses the identity as the action function and has a single output for each action in the minimal action set. The minimal action set is defined by ALE and has between 4 and 18 actions. Bias nodes are used for the hidden layer and the output layer. The network architecture is depicted in figure 4.1.

4.1.3 Exploration

Exploration of the state space is done by a ϵ-greedy policy. ϵ is sampled from a discrete probability distribution before each episode. The probability distribution is given by table 4.1. In addition, ϵ starts at 1 and falls linearly to the sampled value over the first million steps. So in the beginning exploration is done completely randomly then slowly transitions to exploiting the learnt knowledge.

The idea of sampling ϵ from a distribution has previously been used in Mnih et al. (2016) and the intention for varying ϵ is that different exploration rates will explore different areas of the state space. This should be more robust because one ϵ might be too high. A high ϵ may prevent the agent from building on learnt knowledge. However, for the next episode a new, and potentially lower, ϵ is sampled. Also, different games may have different optimal ϵ.

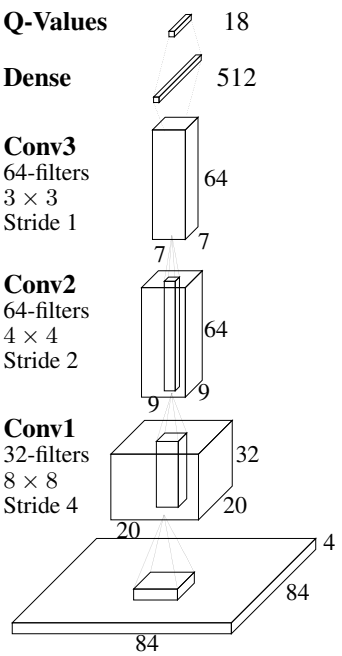


Figure 4.1: The figure depicts the network architecture and is based on a figure from Hausknecht and Stone (2015). Note that the number of output nodes vary based on the game. It can be between 4 and 18.

Note that for many Atari games a specific key must be pressed in order to continue the episode. An example of such a game is Breakout. This game will pause after losing a life and the player must press the ”fire” button in order to continue. So to guarantee the episode will end at some point, ϵ cannot be zero.

Table 4.1: Probability distribution of ϵ.

ϵ	$P(\epsilon)$
0.01	0.3
0.1	0.4
0.5	0.3

4.1.4 Experience Replay

Experience replay is a form of data efficient RL and has been covered in section 2.1.3. The basic idea is to store interactions with the environment in a memory bank, D. When learning, D is sampled rather than directly using interactions with the environment. The sampling is done uniformly and selects a mini-batch of size 32.

Following the parameters from Mnih et al. (2015), D contains 250000 transitions. Other papers have used a larger replay memory, e.g. van Hasselt et al. (2015) used 1000000 while Hausknecht and Stone (2015) used 400000. Each transition is stored in D requires $4 \times 84 \times 84 \times 1\text{B} \approx 28.2\text{KB}$ of memory. With 250000 transitions, the total memory requirement is approximately 6.57GB.

4.1.5 Target Network

Standard Q-learning use one approximation function to select actions (e.q. 2.33) and generate targets (e.q. 2.30). In many cases this causes instabilities. Mnih et al. (2015) separate the action selection and target generation by using two separate networks. One online network that selects actions, $Q(s,\ a;\ \theta)$, and one target network which generates targets, $\hat{Q}(s,\ a;\ \theta^-)$. The target network is a copy of the online network and is copied every τ steps. By introducing a target network, the target equation is altered slightly, see e.q. 3.1.

$$a_t = \underset{a}{\text{argmax}}\, Q(s_t,\ a;\ \theta_t) \tag{2.33 revisited}$$

$$y_t = r_t + \gamma \max_a Q(s_{t+1},\ a;\ \theta_t) \tag{2.30 revisited}$$

$$y_t = r_t + \gamma \max_a \hat{Q}(s_{t+1},\ a;\ \theta_t^-) \tag{3.1 revisited}$$

Further improvements to the target generation is introduced by van Hasselt et al. (2015). This paper recognizes that when targets are generated, the target network both selects an action and evaluates it. Equation 3.2 is a reformulation of 3.1 which shows how selection and evaluation is done by the same network. When the online network selects the action and the target network only evaluates the action, the performance of the DQN algorithm can be boosted. This way of generating targets is called Double DQN and the new target equation is given by equation 3.3.

$$y_t = r_t + \gamma\, \hat{Q}(s_{t+1}, \underset{a}{\text{argmax}}\, \hat{Q}(s_{t+1}, a; \theta_t^-); \theta_t^-) \tag{3.2 revisited}$$

$$y_t^{\text{DoubleDQN}} = r_t + \gamma\, \hat{Q}(s_{t+1}, \underset{a}{\text{argmax}}\, Q(s_{t+1}, a; \theta_t); \theta_t^-) \tag{3.3 revisited}$$

Targets in this thesis is generated by the Double DQN target equation and the target network is updated every $\tau = 30000$ steps. Note that the related work chapter cover this topic in more details, see section 3.3 and 3.4.

4.1.6 Training Time

The experiments run for 10 million steps. This is lower than other related work discussed in chapter 3. Therefore the reported performance is lower for this work. However the

research in this thesis is mostly concerned with the early stages of the learning process so the reduction in training steps is of little importance.

When using a graphical processing unit (GPU) to train the network, one experiment takes approximately one day to perform. However, it is possible to perform multiple experiments on one GPU without significantly increasing the training time.

4.1.7 Training details

This section list several implementation details that increases the performance of the DQN algorithm. These details are omitted from algorithm 5 in order to increase readability.

Optimization

The implementation uses the same optimization algorithm as Mnih et al. (2015), that is RMSprop combined with momentum. This algorithm requires three parameters in addition to the global learning rate: the RMSprop decay ρ, RMSprop stabilization constant δ, and momentum parameter α. The value of the parameters are $\rho = 0.95$, $\delta = 0.01$, and $\alpha = 0.95$.

Action Repeat

A speed improvement is to repeat actions multiple times. This works because it takes much more time to select an action than to execute the action in the ALE. In the experiments, actions are repeated 4 times. ALE runs with a frequency of 60 Hz, so when actions are repeated 4 times, agents are able to select 15 actions per second. Humans can select roughly 10 actions per second, so repeating the actions should not hamper the performance of the agent. Throughout the project, taking 1 step in the DQN algorithm means that the action is repeated 4 times in ALE.

Training frequency

The online network is updated every forth step of the algorithm. Since updates are based on mini-batches of size 32, each transition will on average contribute to 8 updates. This makes the agent learn slower but chances of diverging is reduced.

Reward Clipping

Rewards in ALE vary by several orders of magnitude between games. Therefore it is difficult to use the same learning rate for all games. To address this problem, the rewards from ALE is clipped between -1 and 1. This change makes it possible to use the same

learning rate for all games, but the learning algorithm is no longer able to distinguish between cases where the reward magnitude is larger than 1.

TD error Clipping

In addition to clipping the reward, the TD error is also clipped to be between -1 and 1. This stabilizes the learning process.

Image Normalization

The input images are dividing by 255 in order to normalize them to be between 0 and 1.

Replay Memory Initialization

The replay memory, D, is initialized with 50 000 transitions before training starts. These transitions are generated by a policy which chooses random actions.

Weight Initialization

Weights are initialized uniformly between a lower and upper bound. The magnitude of the lower and upper bound is the same. The bound is calculated based on the layer shape. For convolutional layers the bound is calculated by equation 4.1 and dense layers are calculated by equation 4.2. The same bound is used for weights and bias nodes.

$$\text{convBound} = \frac{1}{\sqrt{\text{numFeaturesIn} \times \text{filterWidth} \times \text{filterHeight}}} \tag{4.1}$$

$$\text{denseBound} = \frac{1}{\sqrt{\text{numNeuronsIn}}} \tag{4.2}$$

GPU programming

Theano is used to reduce the training time. This is done by moving the computations from the CPU to the GPU. Without this tool the training time would be impractically long.

4.2 Visual Pretraining

This thesis explores the possibility of using an AE to learn visual features and then apply TL to speed up the learning process and possibly improve the performance of DQN agents. Intuitively this should work because there are two key concepts that the DQN algorithm

Initialize replay memory D with capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
Initialize emulator
Sample ϵ
Observe state s_1
Preprocess $\phi_1 = \phi(s_1)$
foreach *t in steps* **do**
  if ϕ_t *is terminal* **then**
    Reset emulator
    Sample ϵ
  end
  Choose a_t from ϕ_t using ϵ-greedy policy derived from $Q(\phi_t,\ a;\ \theta)$
  Execute action a_t, observe reward r_t and observe state s_t
  Preprocess $\phi_{t+1} = \phi(s_{t+1})$
  Store $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
  Sample mini-batch $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
  Set $y_j = \begin{cases} r_j & \text{if } \phi_{j+1} \text{is terminal} \\ r_j + \gamma\, \hat{Q}(\phi_{j+1}, \underset{a}{\text{argmax}}\ Q(\phi_{j+1}, a;\ \theta);\ \theta^-) & \text{else} \end{cases}$
  Perform gradient decent on $(y_j - Q(\phi_j,\ a_j;\ \theta)^2)$ with respect to θ
  Every C step reset $\hat{Q}$ by $\theta^- = \theta$
end

Algorithm 5: Pseudo code for deep Q-learning.

must learn in order to perform well: the visual feature extraction and the game specific strategy. If it is possible to learn visual features that are useful for DQN agents before the RL task starts, then the training time could be reduced.

4.2.1 Base Domain Training

As discussed in section 2.2.4, an AE takes an input and encodes it to a hidden representation before decoding it with the goal of reconstructing the original input. For this thesis the AE takes preprocessed images as input. The AE learns visual features because it must recognize structure in the images in order to efficiently code it. The different datasets that are used to train the AEs are introduced in section 4.2.3.

Network architecture of AE

The network architecture of the AE needs to be similar to the architecture used by DQN because weights will be directly copied between the networks. In fact, the encoder has the same structure up to the output layer. The output layer is swapped with another dense layer

which has 18 neurons. This layer outputs the encoding of the input image. The decoder uses a symmetrical structure to the encoder. The weights of the encoder and decoder are independent.

Training Details of AE

The AE is trained with the mean squared difference as the loss function. The difference is calculated between the original image and the reconstructed image. Optimization is done with RMSprop. The RMSprop hyper-parameters are: RMSprop decay $\rho = 0.9$ and RMSprop stabilization constant $\delta = 1 \times 10^{-5}$. The global learning rate is 0.001. The AE is trained as a denoising AE with corruption probability of 0.3. Training is done over 30 epochs.

4.2.2 Transfer Learning

As seen in section 3.9, it is possible to transfer knowledge between neural networks by copying weights. So by training an AE and then initialize the DQN with weights from the AE, the DQN agent should have a better starting point in comparison to the standard approach which initializes with random weights.

Multiple experiments with TL are performed in chapter 5. The main difference between the experiments are the number of weights which are copied. For some experiment, only weights from the first convolutional layer is copied. For other experiments, all layers up to and including the dense layer is copied. In addition, the experiments are run with both frozen and fine-tuned weights. The intention of these experiments is to give some insight to which degree the AE is able to extract visual features which are useful to the DQN agent.

4.2.3 Dataset

The choice of dataset may affect the AEs ability to learn useful features for the DQN agent. Three different alternatives are listed in the following sections.

ImageNet

ImageNet is a database containing a large collection of images for supervised visual computational task, see Deng et al. (2009) for more details. The images are taken of a wide variety of real objects with no limitations on background and camera angle. These images are much more complex than the input for the DQN algorithm. If the AE is able to extract useful features from these images, it might be able to generalize to screenshots from ALE.

There are too many images in the ImageNet database to train the AE on all of them, so training is done on a subset. Specifically 50 000 images from the 2012 validation set. The images are preprocessed in order to fit the network input shape. The preprocessing first reshapes images to the shape 84×84 then stacks four copies of the image. The result is a tensor with the shape $4 \times 84 \times 84$.

The main limitation of this dataset, is that there is no moving object in the images. The velocity and direction of moving objects are important features for many different games. This mean that the AE cannot learn all useful visual features.

AEs trained with this dataset is referred to as ImageNet autoencoders (INAE).

Screenshots

This dataset is composed of preprocessed screenshots from multiple different games. The screenshots are generated by DQN agents which have been trained by the standard DQN algorithm, 5. Screenshots are preprocessed as described in section 4.1.1 so that the image and network input shape are equal. The games used for this dataset is Breakout, Double Dunk, Q*bert, Road Runner, and Space Invaders. Each game contribute with 50 000 preprocessed screenshots.

By using screenshots from Atari games, the AE learns to work with images with the same style as the network will encounter when training the DQN agent. Also, the images contains moving objects. This is something that is missing from the ImageNet dataset.

Since screenshots are taken from specific Atari games, the AE may extract features that are only relevant for the games within the dataset. To test this, one can compare the performance on games within and outside the dataset. This is done in section 5.3.

AEs trained on this dataset is referred to as screenshot autoencoders (SSAE).

Game Specific

Another option for the dataset is to use screenshots from one game. This dataset is a subset of the screenshot dataset. This means that a unique AE must be trained for each game.

AEs trained on this dataset may learn task specific features and not just general visual features. These feature will most likely be more useful for the DQN algorithm. For instance, an AE trained specifically for Breakout may learn the position and velocity of the ball. This is key high level features that the DQN must recognize in order to learn a good policy for Breakout.

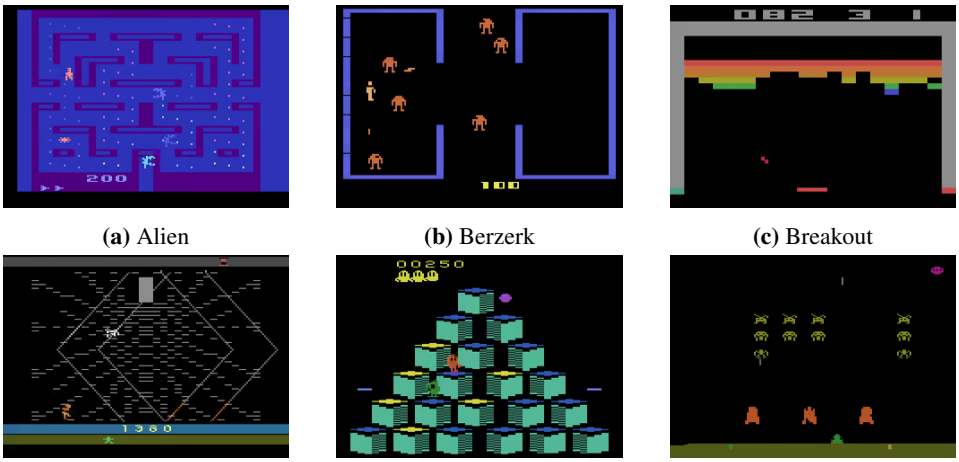
It is worth noting that, the experimental setup is very similar to that of Lange and Riedmiller (2010), when this dataset is used. This paper is discussed in section 3.11. Both approaches train an AE on observation of the state space. Lange and Riedmiller (2010) argues that by reducing the dimensionality of the state space, the RL task becomes easier.

AEs trained on this dataset is referred to as game specific autoencoders (GSAE).

4.3 Evaluation

4.3.1 Test Games

This section gives a short introduction to the games used in the experiments. Screenshots from the different games is presented in figure 4.2.



(a) Alien **(b)** Berzerk **(c)** Breakout

(d) Krull **(e)** Q*bert **(f)** Space Invaders

Figure 4.2: Screenshots of different Atari games used in the experiments.

Alien

The player runs around on the board and attempts to avoid hostile aliens. If the player and alien touch, the player looses a life. The player collects points by piking up alien eggs which is scattered around the board. The player can kill the aliens with a short range flame thrower.

Berzerk

Berzerk is a shooting game where the player walks around in a 2D world and shoots computer controlled enemies. The enemies shoots back and if the player is hit, he looses a life. The player will also loose a life if he touches an enemy or the blue wall. Once all enemies are killed, the player moves to another screen with new enemies.

Breakout

In Breakout, the goal is to clear all block at the top of the screen. Blocks disappear when hit by a moving ball. The ball moves in straight lines and ricochet of blocks and walls. At the bottom of the screen, the player controls a platform. This platform is must be placed underneath the ball to prevent it from moving outside the screen. If this happens, the player looses a life. When the ball ricochets of the platform, the reflected direction is affected by where the ball hit the platform. This makes it possible to aim at specific blocks.

Krull

Krull is an adventure game. In this game there are three different screens. Each screen has its own objective. In the first screen, enemies attack the player and his bride. The player tries to protect the bride by punching enemies. The next screen is depicted in figure 4.2d. To complete this screen the player has to jump between moving threads of web which pushes him back. At the same time the player must avoid a spider. The goal is to reach a window at the top of the screen. Once this is done, the player progresses into the last screen. Here the player must break through a barrier in order to save his bride while avoiding a boss who throws fireballs.

Q*bert

Q*bert is a game where the player jumps around on cubes. The goal is to change the color of all cubes into a target color. Cubes change color when the player jumps on top of them. The rules for how the color changes varies from level to level. For the first level, the cube change color to the target color when the player visit the cube. In later levels, the difficulty is increased by for instance having the player visit every cube twice. In addition, there are a few different hostile entities that either reverts the color of visited cubes or kills the player on contact.

Space Invaders

Space Invaders is a shooting game where the player moves horizontally at the bottom of the screen. Alien space ships shoots directly down at the player who have to dodge the projectiles while shooting back. The ships move downwards and if they reach the bottom of the screen, the player looses.

4.3.2 Evaluation Procedure

Evaluation of trained agents are done by having them play 30 episodes of the game. Each episode can last up to 4500 steps which is five minutes of gameplay. During testing, agents

follow a ϵ-greedy policy where $\epsilon = 0.05$. ϵ cannot be zero because some games require specific actions to be performed in order to continue. Testing is done every 125000 steps.

The limit of 4500 test steps affects the score for Krull. In this game, the agent can play passively and thereby stay alive for a long time. For the other games, the limit is not reached because passive agents will quickly loose lives.

Chapter 5

Result and Discussion

The chapter starts with section 5.1 which compares the performance between Double DQN, TL with the ImageNet autoencoder (INAE), and TL with the screenshot autoencoder (SSAE). In the TL cases all weights are fine-tuned. Section 5.2 performs the same experiments, the only difference is that the transferred weights are frozen. Section 5.3 looks into the SSAE's ability to generalize to games not in the screenshot dataset. The last section, 5.4, looks at the performance for game specific autoencoders (GSAE).

Each section plots the performance for each game. Performance is measured with the game score and it is plotted as a function of training steps. The scores are calculated by following the evaluation method described in section 4.3.2. Additionally, each plot is averaged over 3 runs and moving average is used to smooth the plots over 5 points. This is necessary because the raw game scores oscillate, see the appendex.

Each plot is labeled with an abbreviation. Double DQN refers to the standard run without TL. INAE, SSAE, and GSAE are used to label the different AEs. The abbreviations refer to the dataset which is used to train the AE and they stand for ImageNet AE, Screenshot AE and game specific AE respectively. The TL cases are also marked either with "First" or "All". This refers to how many layers are copied from the AE to the DQN. The output layer in the DQN does not have a corresponding layer in the AE. So when "All" layers are copied, the output layer of the DQN still has random weights. The experiments which freezes weights are marked with [F].

5.1 Double DQN vs Fine-Tuned TL

5.1.1 Results

Figure 5.1 plots the game score for Double DQN and 4 different cases of TL for Breakout, Q*bert, and Space Invaders. In all of the TL cases, the weights are fine-tuned.

The main goal of applying TL is to reduce the training time. None of the plots shows that TL is able to speedup the learning. For Breakout the learning speed is lower for every TL case. While for Q*bert and Space Invaders, the learning speed follows that of Double DQN when only the first layer is copied. When all layers is copied, the learning is significantly slower for both Q*bert and Space Invaders.

When looking at the max performance for each run, it is only Q*bert that benefits from TL. The highest performing policy for Q*bert is the TL case where only the first layer is copied from ether SSAE or INAE. In Breakout, Double DQN beats every TL case because the learning speed is slower for the TL cases. For Space Invaders, all TL cases achieve similar max performance as Double DQN. An except to this is the case where all layers are copied from SSAE. For this run the performance slows down and seems to stagnate before reaching the same performance as other approaches.

Section 5.1.2 discuss different factors that may explain why the TL cases perform worse than Double DQN. Other observations which does not directly relate to the performance of TL is discussed in section 5.1.3.

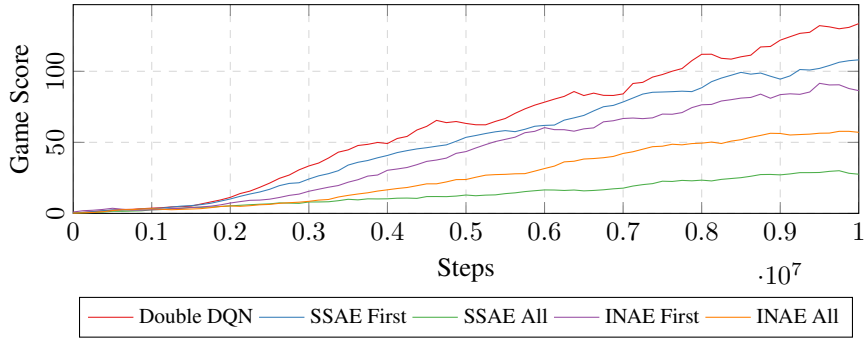
5.1.2 Explanations for Lower Performance

Specialized towards the Base task

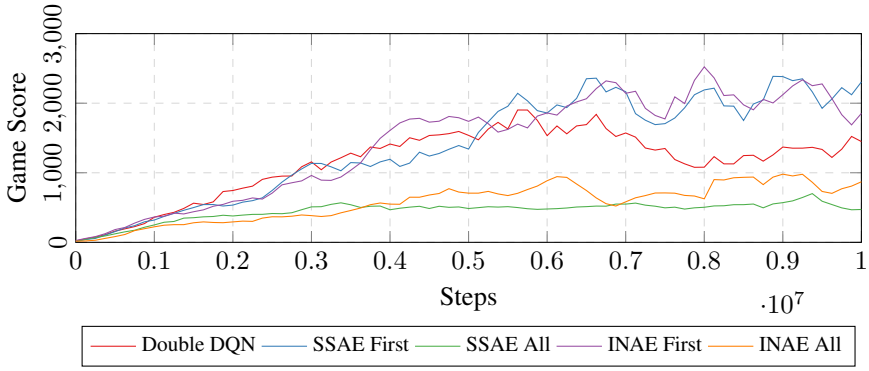
The most likely explanation to why there is no speed improvement or performance boost for TL cases is that the weights are task dependent, meaning that they are specialized towards the base task. With different base and target tasks, TL can negatively affect the performance as shown by Yosinski et al. (2014).

The cost function for AEs will focus on large object and background objects, because they are responsible for the largest reconstruction error. Small variations in the images might be mistaken for noise and consequently ignored. These small variations can be the placement of small objects, their velocity or direction. If the AE suppresses these features, the DQN algorithm may work with a feature representation that is worse than randomly initialized features.

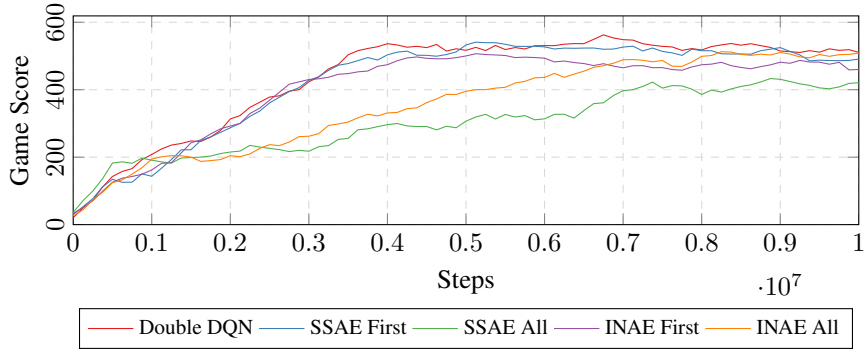
To get an indication of how well the AE learns useful image representations, one can look at original images and there reconstructed counterparts. This is done in figure 5.2. In the reconstructed images, objects that are obviously important for the policy have disappeared. Examples of objects that disappear are the ball in Breakout, the player in Q*bert, and projectile in Space Invaders. This suggest that the image representation learnt by the AE



(a) Breakout



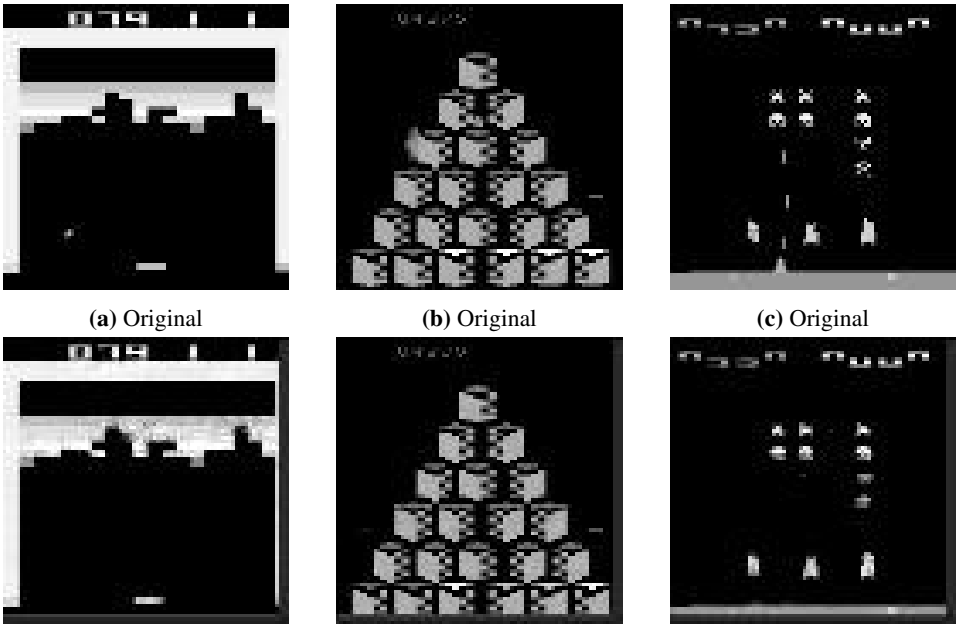
(b) Q*Bert



(c) Space Invaders

Figure 5.1: Plots of the game score as a function of training steps. For all experiments the weights are fine-tuned.

suppresses crucial details in images. In it self this does not prove that the AE suppress details. It could be that details are accounted for in the encoding but lost in the decoding process.



(a) Original **(b)** Original **(c)** Original

(d) Reconstruction **(e)** Reconstruction **(f)** Reconstruction

Figure 5.2: Shows examples of how the SSAE is unable to reconstruct details in ALE frames. For Breakout, the ball disappear. The player vanishes from the board in Q*bert. This can be difficult to see because the player appears as a diffuse cloud. The player is located in the middle left of the pyramid. For Space Invaders, the player and the projectiles are removed. The original images are black and white because they have gone through the preprocessing steps from section 4.1.1 and the reconstructions are black and white because the AE outputs black and white images.

If the problem lies in that the AE ignores details, it could be beneficial to increase the network size and work with original images from ALE. Section 6.3.1 explores this idea further.

Weight magnitudes are larger

When the AE learns to encode images, the weights in the network grow. This is depicted in figure 5.3. There is a significant difference in magnitude for weights that are randomly initialized and trained by the AE. Parameters for the DQN algorithm is optimized for weights with similar magnitude to the randomly initialized weights. When the magnitude change, it is likely that the optimal value for these parameters will also change.

The magnitude difference between screenshots and ImageNet can be explained by the number of samples in each dataset. The screenshot dataset is 5 times larger than the ImageNet dataset. Consequently, the screenshot AE is trained 5 time longer. This means that the weights have more time to grow.

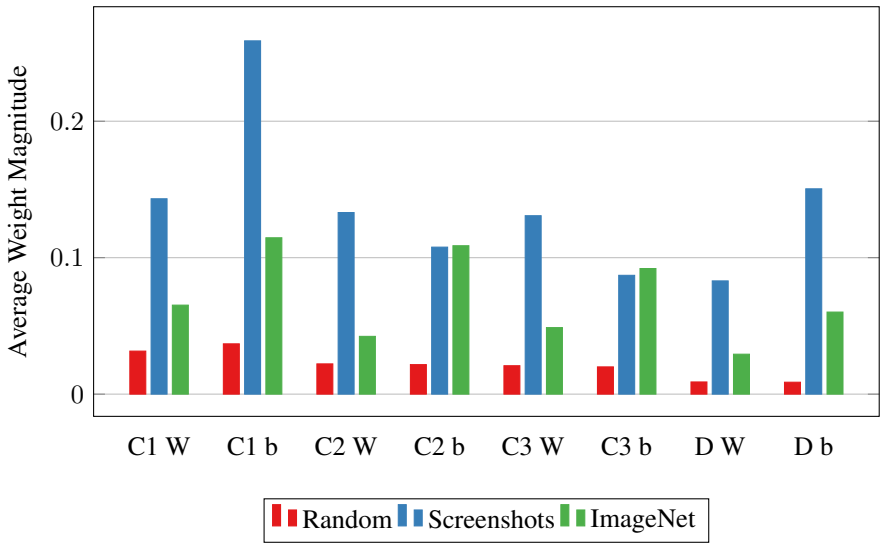


Figure 5.3: Shows the average weight magnitude for different staring points of the DQN algorithm. Random is the standard weight initialization method. Screenshots and ImageNet is the trained AE weights which are copied to the DQN network. 'C1 W' plots the bar chart for filters in the first convolutional layer, 'C1 b' is the biases for the first layer.

Large magnitudes can explain why many TL cases learns slower. Since weights are larger, it will take more time for the DQN algorithm to change them. This means that the learning process will take more time. It should be noted that the weights are not so large that the DQN algorithm cannot significantly change them. The DQN algorithm must be able to change them because the performance is much higher when fine-tuning weights rather then freezing them. This can be seen when comparing the performance between figure 5.1 and 5.4 for the TL cases.

Large magnitudes can also explain the improved performance for Q*bert in the case where the first layer is copied. The performance of Double DQN peaks at approximately 6.5 million steps then it falls. In the TL cases the performance continue to increase after 6.5 million steps. With larger weights the network may be more robust to the diverging problems often observed in RL problems.

Easy visual features

The degree to which TL helps may be limited because visual objects in Atari games are very simple. Most objects in Atari games cannot be rotated, scaled, or change color, they are a fixed structure of pixels which is moved around on the screen. Consequently it is very easy to recognize them. This means that the visual feature extraction require little training time. Which again means that the DQN algorithm use most of its training time on game specific strategies rather than visual features. Since the AE cannot learn game

specific knowledge, it gives no benefits. If Atari games took place in a complex 3D world, where more of the training time is dedicated to learn visual features, the agent may benefit more from TL.

The experiment in section 5.2.1 to some degree confirms this hypothesis. It shows that its possible to freeze random weights in the first layer and still learn to play Breakout. Not surprisingly this significantly reduces the performance but nevertheless the performance is well over random play.

Imperfect Screenshots Dataset

The screenshot dataset is generated by DQN agents that do not visit every part of the state space. For instance in Breakout, the agent which generated the screenshots, is not able to clear all the blocks. This means the the AE learns from images which span a limited part of the state space. This can limit the performance in the TL cases.

5.1.3 Other Observations

Difference between Screenshots and ImageNet

For Q*bert and Space Invaders there is only a small difference between the performance for the SSAE and the INAE. This small difference can easily be explained by random fluctuations in the performance. However, for Breakout the difference is larger. This difference is not huge and can be explained by fluctuations in the performance. Nevertheless, it seems like the SSAE performs better when the first layer is copied and the INAE performs better when the complete network is copied.

The SSAE may work better when copying the first layer for two reasons. First, real images is quite different from Atari screenshots. Second, there is no movement in ImageNet images. Both of these factors may make the SSAE learn more relevant low-level features.

The simplest explanation to why the INAE performs better when all layers are copied, is that both AEs have learnt features that does not help the DQN agent. The difference is that the SSAE has larger magnitudes. With larger magnitudes it takes more time for the DQN algorithm to correct the weights. The magnitudes are depicted in figure 5.3.

Stable learning in Breakout

The learning process for Breakout is stable compared to Q*bert and Space Invaders. This is very apparent in plots that do not calculate a moving average, see appendix. For Q*bert and Space Invaders, the performance oscillate while for Breakout, the performance increase almost linearly. This could be caused by the fact that Breakout has a very simple and good policy; move the platform under the ball. Further training can perfect this policy by

making accurate predictions of where the ball will be and thereby increase the probability of hitting the ball.

For Q*bert and Space Invader, there exist no simple policy. For these games multiple elements must be combined in order to learn a decent policy. For instance, in Q*bert, the agent must not jump of the edge, move towards unvisited platforms, and avoid the hostile computer controlled agent. These are distinct objectives that the agent must learn to recognize and prioritize correctly. This is more complicated than the simple policy for Breakout. As a consequence, the performance in Q*bert will oscillate more.

This observation gives rise to higher confidence that the results from Breakout is statistically significant. Given the a small performance difference for Q*Bert or Space Invaders the result will most likely not be statistically significant because the variance it to high. However a small difference in game score for two Breakout experiments may be statistically significant because of the low variance.

Stagnation for Q*bert and Space Invaders

The improvements to the policy for Q*bert and Space Invaders stagnates early in the learning process for Double DQN. This does not occur in other papers, e.g. Mnih et al. (2015) and van Hasselt et al. (2015). One explanation can be that in these papers they run the DQN algorithm five times longer than the experiments in this section. With more time the performance may increase.

Another explanation is that the parameters used in the experiments causes the stagnation. The parameter that is not shared between these experiments, Mnih et al. (2015), and van Hasselt et al. (2015) is the target generation, the target network update frequency, the replay memory size, the exploration rate, and the test exploration rate. The value of each parameter can be seen in table 5.1. The parameter combination used in this thesis may reduce the performance.

Table 5.1: Shows the value of parameters not shared by these experiments, Mnih et al. (2015), and van Hasselt et al. (2015).

	These experiments	Mnih et al. (2015)	van Hasselt et al. (2015)
Target Generation	Double DQN	Single DQN	Double DQN
Target Network Update Rate	30 000	10 000	30 000
Replay Memory	250 000	250 000	1 000 000
Exploration Rate	[0.5, 0.1, 0.01]	0.1	0.01
Test Exploration Rate	0.05	0.05	0.001

5.2 Double DQN versus Frozen TL

When AE weights are frozen in the DQN it gives some insight into whether or not the image representation, learnt by the AE, describes the screen sufficiently for a policy. Since the weights are unchanging, the DQN algorithm must learn a policy based on the AE's features.

The plots show lower performance for all cases when comparing frozen experiments to fine-tuned experiments. Lower performance is expected because the AE features are not be optimized for the DQN task and the number of free parameters is reduced. However, the performance drop is so significant that it is likely the feature representation learnt by the AE does not give any benefits.

Low-Level Features

If the DQN algorithm is able to learn a high performing policy in the case where the first layer is frozen, it means that the low-level feature representation learnt by the AE extracts most of the needed information from the screenshots.

The plots show lower performance both in comparison to the Double DQN run and the TL cases where the weights are fine-tuned. This indicates that there are some key features that the DQN algorithm need which the AE have not learnt. The difference is most striking in Breakout. In this game, the DQN algorithm is barely able to improve the performance at all when the first layer is frozen. For breakout low-level features must be able to accurately detect the direction of the moving ball. Since the performance is so poor, it is likely that the low-level feature representation learnt by the AE ignores this motion.

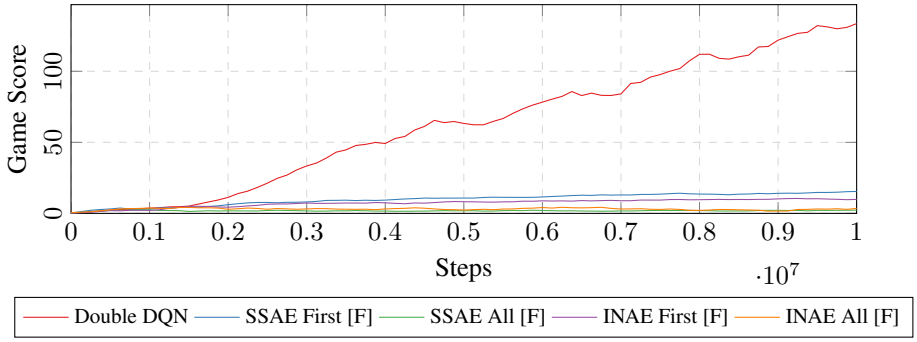
High-level Features

When all layers, except the output layer, have been frozen, the DQN algorithm must construct a linear policy based on high-level features learnt by the AE. The plots show little to no improvement over random play for all combinations of games and dataset. This is not surprising given that the low-level features does not extract enough information to learn a high performing policy. These experiments reveal that the high level features learnt by the AE has very little useful information for the DQN task and that there is a big difference between the high level features in an AE and a DQN.

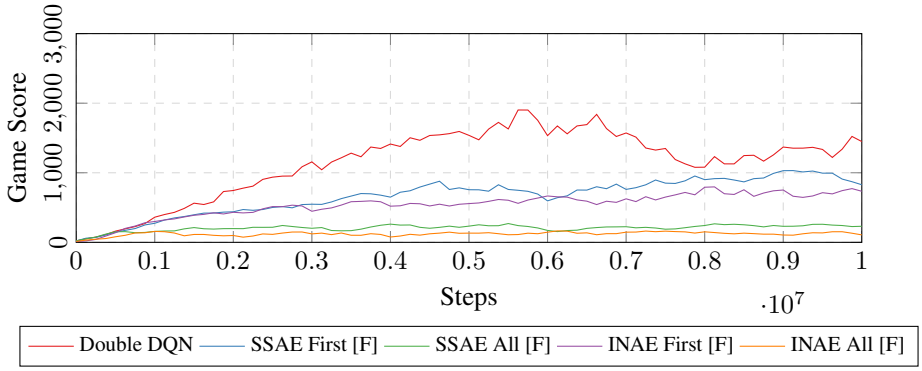
Since the learnt policy is linear, there might be some useful features that only a nonlinear policy could use. However it is unlikely.

5.2.1 Frozen Random Weights

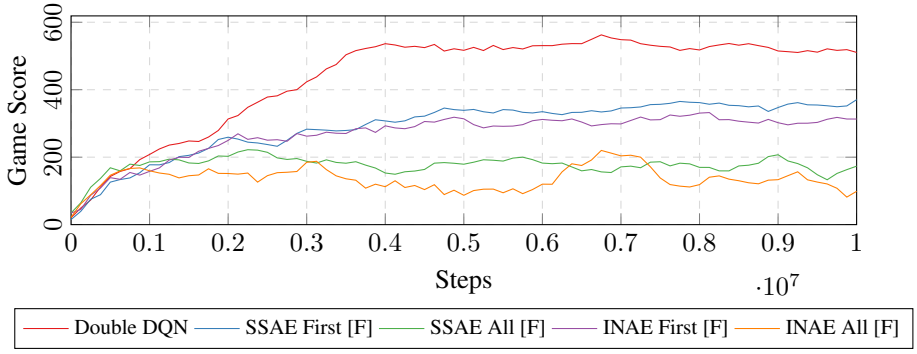
Given the exceptionally low performance for Breakout in figure 5.4 and the removal of small objects in the reconstructed screenshots in figure 5.2 it is likely that the pretraining



(a) Breakout



(b) Q*bert



(c) Space Invaders

Figure 5.4: Plotting the game score as a function of training steps. All transferred weights are frozen in the DQN.

is harming the performance. To back this up, another experiment, where random weights are frozen in the DQN, is performed. The result can be seen in figure 5.5.

The plots show that freezing random weights in the first layer will give higher performance

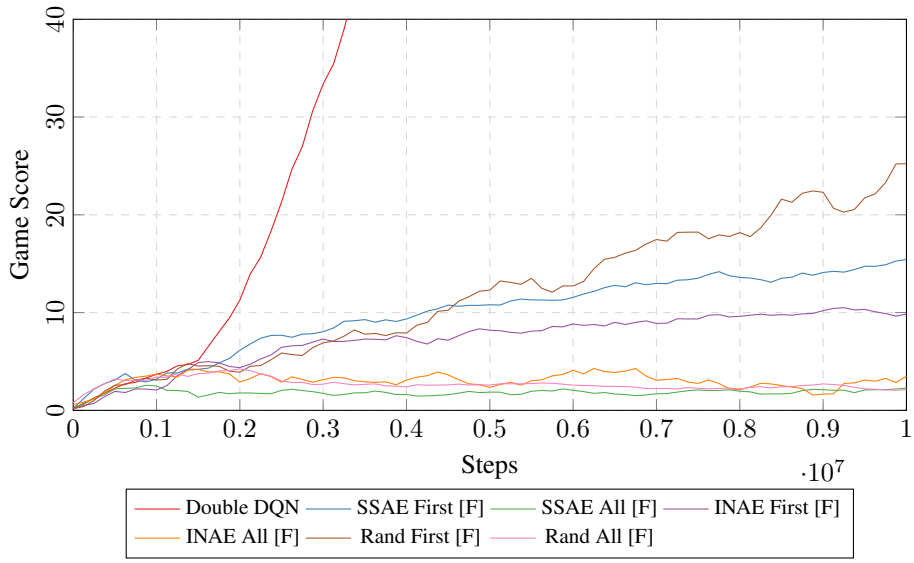


Figure 5.5: Plotting the performance, when freezing weights, for Breakout. The weights are either randomly initialized or taken from the INAE or SSAE. Note that the scale on the y-axis has changed from previous plots.

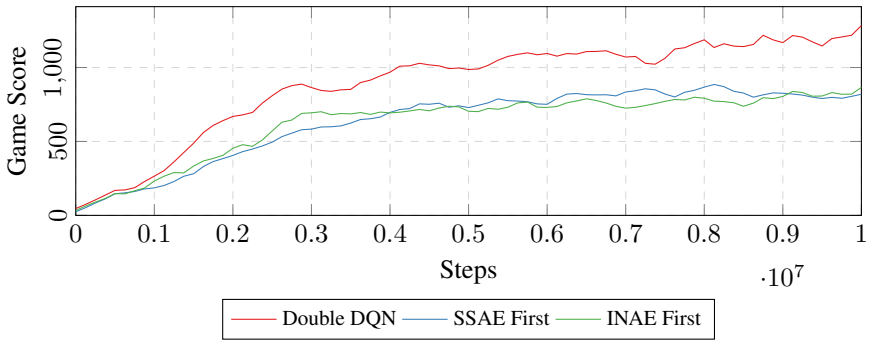
than freezing weights from either INAE or SSAE. The difference is not huge and again it should be noted the the result may not be statistically significant. For the case where the complete network is copied, the random weights does not enable the DQN agent to improve the policy over random play. This is the same as with the AE weights.

5.3 Generalization with Screenshots

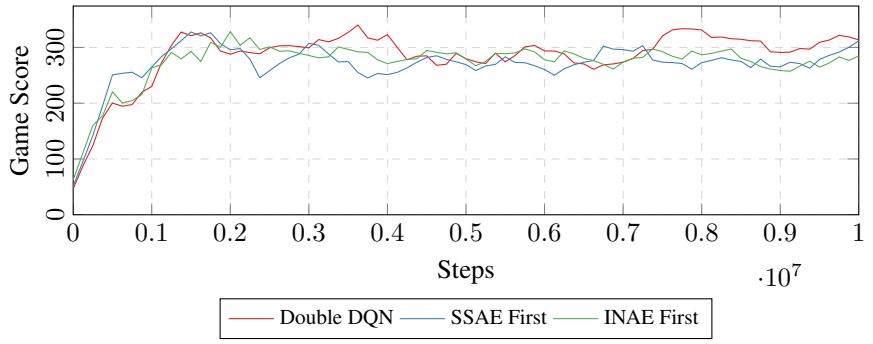
This section investigate how the use of screenshots affects the generalization ability of the AEs. Previously, in section 5.1, the SSAE is tested on games that appear in the screenshot dataset. By testing the performance on games not in the dataset, it may be possible to say something about the SSAEs ability to generalize.

The experiments in figure 5.6 shows that the SSAE performs equally well the INAE. The same happened in section 5.1, for Q*bert and Space Invaders. This indicates that the first layer of the SSAE is not specialized towards games in the screenshot dataset. However, since the AE seems not to learn useful features, there may be little that separates them. As a result, the outcome of the experiment may change if the AEs for some reason is able to learn useful features. If the AEs are tested on a different set of games, it may learn useful features. Also, the future work section, 6.3, has multiple suggestions that may enable the AEs to learn useful features.

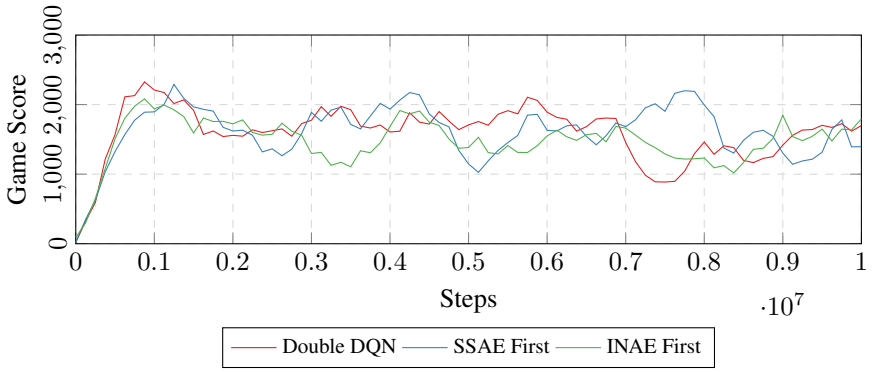
The performance plot for the game Alien, is another example of how TL impedes the



(a) Alien



(b) Berzerk



(c) Krull

Figure 5.6: Plotting the games score as a function of training steps. The games does not appear in the screenshot dataset. For all experiments the weights are fine-tuned.

performance of the DQN algorithm. This is happens for both AEs and therefore it does not offer any insight to which dataset will generalize best.

5.4 Specialization with Screenshots

Given the low performance for the different TL cases in previous sections, it is possible that the AEs are to ambitious. If the AE is trained on screenshots from only one game rather than multiple games or ImageNet, the performance may be boosted. This is tested in figure 5.7. For these experiments, an AE is trained specifically for each game. The plots are compared to the results from section 5.1.

5.4.1 Results

Figure 5.7 shows similar performance for the GSAE and the AEs trained on general datasets. Given that the performance is similar to the other AEs it is likely that the GSAE suffer from some of the same problems. For instance the inability to account for small details and the increased weight magnitudes.

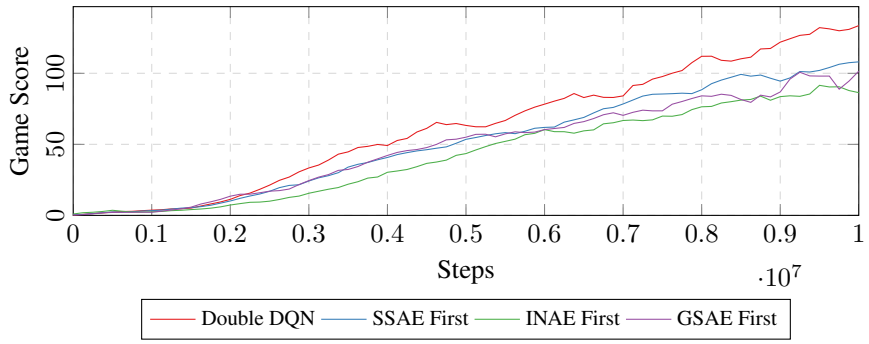
To verify the problems, the same figures as in section 5.1 have been created. Figure 5.8 shows that the GSAE is unable to reconstruct details for Breakout and Space Invaders. When it comes to Q*bert, the GSAE can reconstruct some of the details but not all, see figure 5.9. A plot of the average weight magnitude is included in the appendix. This plot confirms that the GSAEs weights are much larger than randomly initialized weights.

5.4.2 Correct Reconstructions for Q*bert

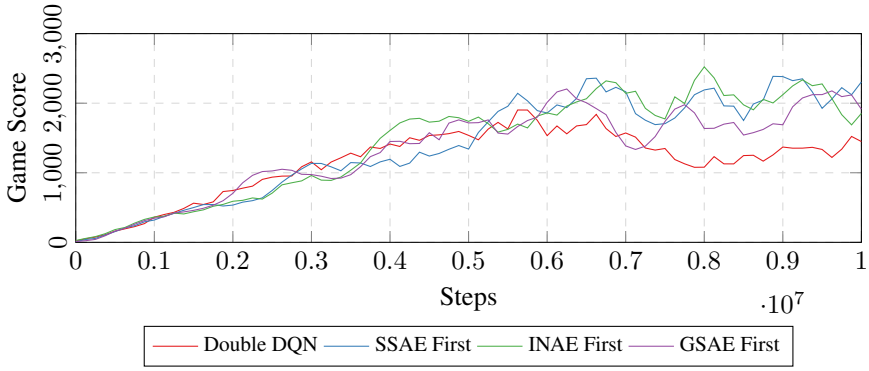
The Q*bert results raise two questions. First, why is the GSAE able to reconstruct details in the image when the SSAE is not? Second, why is the GSAE able to reconstruct details for Q*bert bot not Breakout or Space Invaders?

The first question is easy to answer. The SSAE must reconstruct screenshots from 5 different games. Consequently five different games must be accounted for in the encoding in the middle of the AE. For the GSAE the entire encoding is dedicated to Q*bert. So more details can be stored.

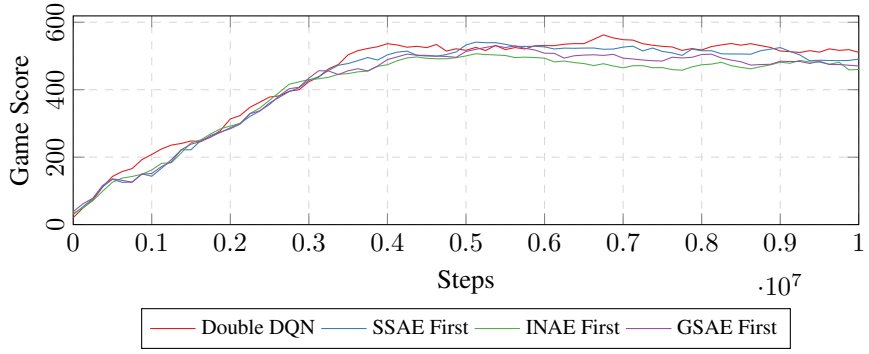
There is two factors that affects the answer for question two. First, when comparing the size of the player in Q*bert with the ball in Breakout or projectiles in Space Invaders, the player in Q*bert is larger. This means that the player will generate a larger reconstruction error if it is not reconstructed correctly. Second, the player may remain stationary for some time before jumping to a new platform. Therefore the dataset contains multiple images where the player is in the same position. This may prevent the AE mistake the player for noise. In figure 5.9a the play stands on a platform and the GSAE is able to reconstruct this image. While the GSAE is unable to reconstruct the player in figure 5.9b. In this figure the player is moving between platforms.



(a) Breakout



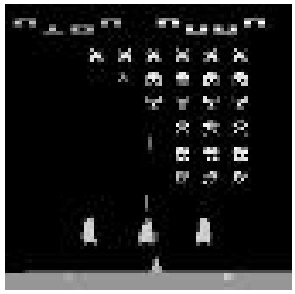
(b) Q*bert



(c) Space Invaders

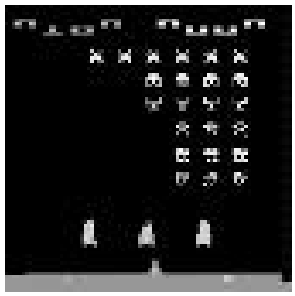
Figure 5.7: Compares the performance between INAE, SSAE and GSAE when copying the first layer from the AE to the DQN. GSAE has been trained based on screenshots from the game the DQN agent plays afterwards.

(a) Original



(b) Original

(c) Reconstruction

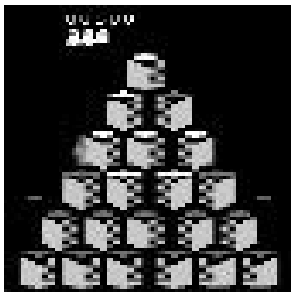


(d) Reconstruction

Figure 5.8: Shows that the GSAE is not able to reconstruct details for Breakout and Space Invaders. The ball disappear in Breakout, while projectile disappear from Space Invaders.

5.4.3 No Performance Boost for Q*bert

Given that the GSAE is able to reconstruct some of the details for Q*bert, one would expect the DQN algorithm to benefit from TL from the GSAE. However, performance plots for Q*bert show almost no difference between the AEs. This is somewhat surprising and it seem like the image representation learnt by the GSAE is still inadequate for the DQN algorithm. It is also worth to repeat the argument about optimizing the hyper-parameters for the DQN algorithm. By changing the parameters of the DQN algorithm, it may work better with larger magnitudes. This may help the DQN algorithm may benefit from the pretraining.



(a) Original

(b) Original

(c) Reconstruction

(d) Reconstruction

Figure 5.9: Shows that the GSAE is able to reconstruct some of the details for Q*bert. In both figures the player is located in the middle left of the pyramid. In sub figure 5.9a the GSAE is able to reconstruct the player. However in figure 5.9b the player have moved slightly to the left and the GSAE is not able to reconstruct the player.

Chapter 6

Conclusion

This chapter starts by giving a summary of the previous chapters. Then the goals are evaluated in section 6.2. The final section comes with suggestions for future work.

6.1 Summary

Chapter 2 gives an introduction to the underlying technique that this thesis is built on. The first section of this chapter goes through reinforcement learning (RL). The focus is on understanding the basic concepts, such as the problem setting, the goal and training technique. RL problems take place in Markov decision processes and RL attempts to learn a policy which maximizes discounted future rewards. One way of doing this is to learn action-value functions. These functions estimate the value of taking an action in a state. For this thesis, action-value functions are trained by q-learning which is a temporal difference learning method. The section finishes with introducing data efficient RL. After the background on RL, the chapter treats artificial neural networks (ANN). The focus is on convolutional neural networks (CNN) which is a special type of ANNs that is often used in visual tasks. The section also presents two applications. First, how ANNs can approximate action-value functions. Second, how ANNs can learn visual features by learning to reduce the dimensionality of input images. ANNs that perform this task are called autoencoders (AE).

Chapter 3 focuses on RL with the Deep Q-Network (DQN) algorithm. This approach combines the main topics from chapter 2: RL and ANN. This combination is able to learn human level policies for Atari games, based on screenshots and rewards from Atari games. Recently, several improvements to the DQN algorithm has been published. These improvements are the main focus of this chapter. In addition, examples of transfer learning (TL) for deep neural networks are reviewed.

Chapter 4 describes the version of the DQN algorithm used for this theses. In addition, it presents the technique for visual pretraining with AEs and how TL is used to initialize DQN with weight from the AE. The experimentation with visual pretraining is what separates this study from previous work in the domain.

Chapter 5 plots and discusses the performance of different experiments. These plots reveal in general lower performance when the pretraining is utilized. In order to explain the lower performance, two figures, which show the effect of pretraining, are examined. First, details disappear from the screenshots, see figure 5.2. Second, the magnitudes of the weights are increased, see figure 5.3.

6.2 Goal Evaluation

The goal of this thesis is to explore the effects of visual pretraining for deep reinforcement learning. To achieve this goal, experiments are carried out where an AE learns visual features before TL transfers the learnt feature to a DQN. These experiments are designed so that they may answer the research questions in section 1.2.

Research Question 1:

Can autoencoders learn visual features that are useful for Deep Q-Network agents?

In general the experiments indicated that the features learnt by AEs are not useful for DQN agents. If the AEs did learn useful features, one would expect the learning process to take less time. This did not happen in any of the experiments. In addition, the experiments from section 5.2 show that when freezing the AEs weights, the performance significantly drops. This demonstrates that the feature representation learnt by the AE does not describe the screen sufficiently for a high performing policy. Furthermore, section 5.2.1 demonstrates that frozen random weights are better than the trained AE weights for the game Breakout.

The only experiment that may suggest that the features learnt by the AE is better than randomly initialized weights is for Q*bert. TL was able to increase the performance when only the first layer was copied and then fine-tuned, see figure 5.1. However this can be explained by larger magnitudes which causes the network to be more robust to divergence tendencies and not because the AE has learnt useful features.

The most likely explanation for why the AE does not learn useful features is that the AE focuses on larger objects while the DQN task needs small details from the screen. Figure 5.2 shows that the screenshot autoencoder (SSAE) is not able to reconstruct small objects. Presumably, the AE believes the details are noise which should be ignored. If details are suppressed by the network, the RL task is more challenging. In addition, the average weight magnitude is larger in the trained AEs than in the randomly initialized DQN. As a consequence, it is more demanding to change the weights, especially since parameters like the learning rate is optimized for smaller weights.

Even though the AEs in these experiments do not learn useful features, there may be cases where AEs are useful. For instance, if games contain large visual objects which the AE can

account for. Also, the future work section mostly focuses on ways to make the AE account for smaller details. If any of the proposed techniques solves this problem, AEs may learn useful features in games where the important objects are too small for the current AE to detect them.

Research Question 2:

How does the training set affect the autoencoders ability to learn general visual features?

Section 5.3 and 5.4 attempts to generate results that may answer this question. This is done by checking if the SSAE can generalize to games outside of the training set and comparing the performance of SSAE and INAE to game specific autoencoders (GSAE). The performance plots show surprisingly little difference between the different datasets, see figure 5.6 and 5.7. This can be connected with the conclusion from research question 1. Since none of the AEs seems to learn useful features, there is little that separates them.

It is worth noting that the GSAE is able to reconstruct some objects for Q*bert, see figure 5.9. This is an improvement over the SSAE. However, the DQN algorithm is still unable to boost performance with this feature representation. Also, the GSAE is unable to reconstruct key objects for Breakout and Space Invaders.

The only game where the dataset seems to have some effect on the performance is Breakout. This is discussed in section 5.1.3. It seems like the SSAE learns better low-level features while the ImageNet autoencoder (INAE) learns better high level features. The SSAE may learn better low-level features simply because the dataset contains similar images to what the DQN algorithm works with. The smaller magnitudes in the INAE gives this AE an advantage when the complete network is copied because the DQN algorithm can more easily change the weights.

All in all, it is difficult to present a conclusion without having additional experiments that show a significant difference between the datasets. However, given that the GSAE seems unable to learn useful features, it seems unlikely that AEs that attempt to learn general visual features will help the DQN algorithm. Future studies may want to focus on approaches which use game specific visual pretraining.

6.3 Future Work

Throughout chapter 5 it has been shown that the proposed pretraining reduces the performance. This section suggest some changes that may alleviate some of the observed problems. Section 6.3.1 and 6.3.2 attempt to solve the problem that AE seems to ignore details in the input. Section 6.3.3 addresses the problem of increased weight magnitudes in the trained AE. Section 6.3.4 consider the use of multi-task learning rather than TL. Section 6.3.5 looks at two alternative tasks that could replace dimensionality reduction with AE as the base task for TL.

6.3.1 Increase Network Size

One of the problems using AEs is that details in the images are ignored. A simple solution is to increase the network size. Doing so, more details can be accounted for by the AE. Another way to increase the network size, is to use the original screenshots from the arcade learning environment. The original images are larger and have color. This could make it easier to learn the small details.

As discussed in section 3.8, Oh et al. (2015) are able to accurately predict screenshots 100 time steps into the feature. This is done with a network that is larger than the standard DQN and uses original ALE images as input. This shows that it is possible to train networks that can account for the game dynamics of Atari games.

6.3.2 Weight AE Loss Function

The AE in this thesis use mean squared error as the loss function, see eq 2.35. With this loss, the AE struggles with the reconstruction of small moving objects. By adding a weight term to the loss function it may be possible to force the AE to focus on these objects. If x is the stack of images which the AE attempts to reconstruct. Then motion can be detected by taking the time derivative of x, $\frac{\partial}{\partial t}x$. The weight term, see eq 6.1, should also include a small constant c, so that the AE attempts to reconstruct the complete image. The reweighted loss function is given by equation 6.2.

$$L(x, r) = (x - r)^2 \tag{2.35 revisited}$$

$$m = \frac{\partial}{\partial t}x + c \tag{6.1}$$

$$L^*(x, r) = m\,(x - r)^2 \tag{6.2}$$

6.3.3 Normalize AE weights

If the observed problems with the AE is a result of the difference in weight magnitude between trained AE weights and random weights, there are two obvious solutions. First, change the learning rate for the DQN algorithm. Second, normalize the AE. Normalization could be done by adding a L1 or L2 norm penalty to the cost function. Early stopping of the AE training could also be used. These regularization techniques are described in Bengio and Courville (2016).

6.3.4 Multi-Task rather than TL

This thesis separates learning into two phases. First, visual pretraining with an AE, then the RL task. Between the two phases, TL is used to transfer the knowledge from one

domain to another. It may be possible to learn both tasks at the same time. This could be done by multi-task learning, see Bengio and Courville (2016). Multi-task learning uses multiple related tasks to learn a shared representation of the input. By using multiple tasks, more constrains are put on the weights and a better representation can be learnt. The main benefit of this approach is that the need for a dataset is removed. Section 5.1.2 points out that the dataset does not cover every area of the state space. This may limit the benefits of TL.

Note that this does not attempt to fix any of the problems observed with AEs, e.g. ignoring details and magnitude difference. So to get any benefits from this approach, it should probably be combined with one or more of the proposed improvements from section 6.3.1, 6.3.2, or 6.3.3.

6.3.5 Alternative Base tasks

Predict the Next Frame

Oh et al. (2015) trains a CNN to predict the next frame. This may be a better base task than dimensionality reduction. If screenshots are used, the network may learn game dynamics such as the movement and interactions of different objects. It is not possible to use ImageNet images with this approach. However, any video could be used.

The CNN used for this task will be similar to an AE. However, AEs require a low dimensional encoding in the middle of the network. If the encoding is too large, the AE may learn the identity function. If the CNN attempts to predict the next frame, it will not learn the identity function because objects are moving around on the screen. Consequently the ”encoding” can have a higher dimension. This may enable the network to account for more details.

Classification

Classification has previously been shown to generalize well to new tasks, see section 3.10. By training a CNN to perform classification rather than dimensionality reduction, the CNN may focus more on details in the image and thereby giving the DQN algorithm a better starting point.

I see no way of labeling Atari screenshots so that the classification task would make sense. This approach would therefore only work with the ImageNet dataset.

Bibliography

Bellemare, M. G., Naddaf, Y., Veness, J., Bowling, M., 06 2013. The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research 47, 253–279.

Bengio, I. G. Y., Courville, A., 2016. Deep learning, book in preparation for MIT Press.
URL http://www.deeplearningbook.org

Bishop, C. M., 1995. Neural Networks for Pattern Recognition. Oxford University Press, Inc., New York, NY, USA.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., Fei-Fei, L., 2009. Imagenet: A large-scale hierarchical image database. In: Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on. IEEE, pp. 248–255.

Ernst, D., Geurts, P., Wehenkel, L., Littman, L., 2005. Tree-based batch mode reinforcement learning. Journal of Machine Learning Research 6, 503–556.

Graves, A., Mohamed, A., Hinton, G. E., 2013. Speech recognition with deep recurrent neural networks. CoRR abs/1303.5778.
URL http://arxiv.org/abs/1303.5778

Hausknecht, M. J., Stone, P., 2015. Deep recurrent q-learning for partially observable mdps. CoRR abs/1507.06527.
URL http://arxiv.org/abs/1507.06527

Krizhevsky, A., Sutskever, I., Hinton, G. E., 2012. Imagenet classification with deep convolutional neural networks. In: Pereira, F., Burges, C., Bottou, L., Weinberger, K. (Eds.), Advances in Neural Information Processing Systems 25. Curran Associates, Inc., pp. 1097–1105.
URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep
pdf

Lange, S., Riedmiller, M., July 2010. Deep auto-encoder neural networks in reinforcement learning. In: Neural Networks (IJCNN), The 2010 International Joint Conference on. pp. 1–8.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., Kavukcuoglu, K., 2016. Asynchronous methods for deep reinforcement learning. CoRR abs/1602.01783.
URL http://arxiv.org/abs/1602.01783

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., Hassabis, D., Feb. 2015. Human-level control through deep reinforcement learning. Nature 518 (7540), 529–533.
URL http://dx.doi.org/10.1038/nature14236

Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., Maria, A. D., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., Legg, S., Mnih, V., Kavukcuoglu, K., Silver, D., 2015. Massively parallel methods for deep reinforcement learning. CoRR abs/1507.04296.
URL http://arxiv.org/abs/1507.04296

Oh, J., Guo, X., Lee, H., Lewis, R. L., Singh, S. P., 2015. Action-conditional video prediction using deep networks in atari games. CoRR abs/1507.08750.
URL http://arxiv.org/abs/1507.08750

Razavian, A. S., Azizpour, H., Sullivan, J., Carlsson, S., 2014. Cnn features off-the-shelf: an astounding baseline for recognition. In: Computer Vision and Pattern Recognition Workshops (CVPRW), 2014 IEEE Conference on. IEEE, pp. 512–519.

Riedmiller, M., October 2005. Neural fitted q iteration - first experiences with a data efficient neural reinforcement learning method. In: Lecture Notes in Computer Science: Proc. of the European Conference on Machine Learning, ECML 2005. Porto, Portugal, pp. 317–328.

Schaul, T., Quan, J., Antonoglou, I., Silver, D., 2015. Prioritized experience replay. CoRR abs/1511.05952.
URL http://arxiv.org/abs/1511.05952

Sutton, R. S., Barto, A. G., 1998. Introduction to Reinforcement Learning, 1st Edition. MIT Press, Cambridge, MA, USA.

Sutton, R. S., Barto, A. G., 2015. Introduction to reinforcement learning, in progress.
URL https://webdocs.cs.ualberta.ca/~sutton/book/the-book.html

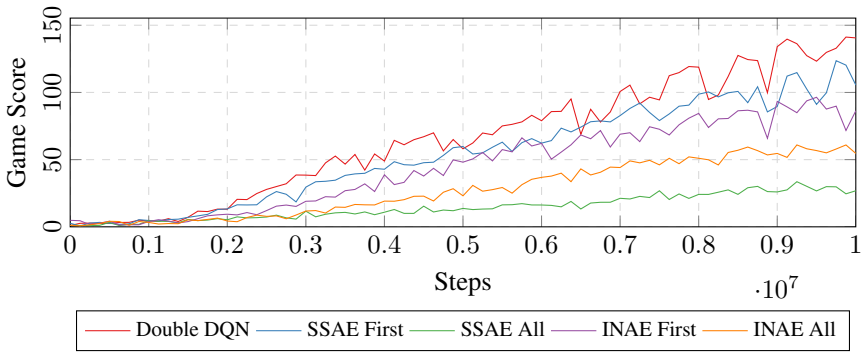
Tesauro, G., Mar. 1995. Temporal difference learning and td-gammon. Commun. ACM 38 (3), 58–68.
URL http://doi.acm.org/10.1145/203330.203343

van Hasselt, H., Guez, A., Silver, D., 2015. Deep reinforcement learning with double q-learning. CoRR abs/1509.06461.
URL http://arxiv.org/abs/1509.06461

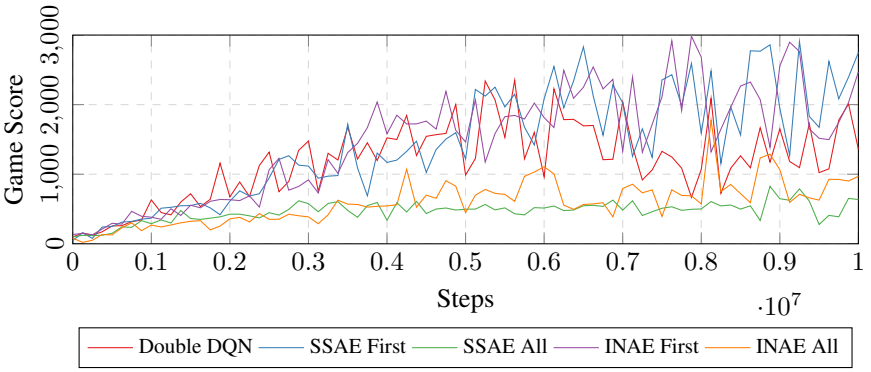
Wang, Z., de Freitas, N., Lanctot, M., 2015. Dueling network architectures for deep reinforcement learning. CoRR abs/1511.06581.
URL http://arxiv.org/abs/1511.06581

Yosinski, J., Clune, J., Bengio, Y., Lipson, H., 2014. How transferable are features in deep neural networks? In: Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., Weinberger, K. Q. (Eds.), Advances in Neural Information Processing Systems 27. Curran Associates, Inc., pp. 3320–3328.
URL http://papers.nips.cc/paper/5347-how-transferable-are-features-in-pdf

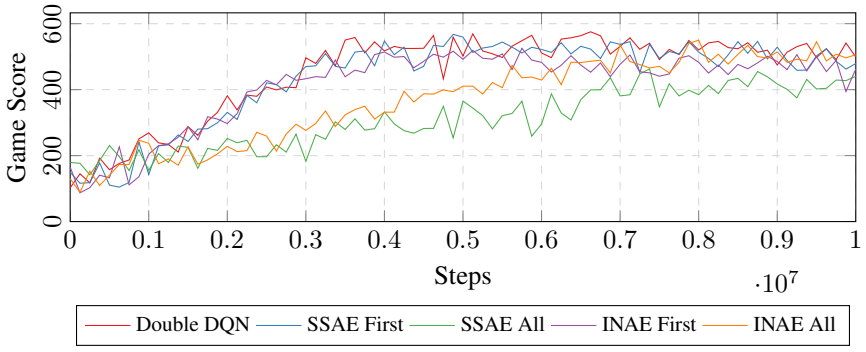
Appendix



(a) Breakout

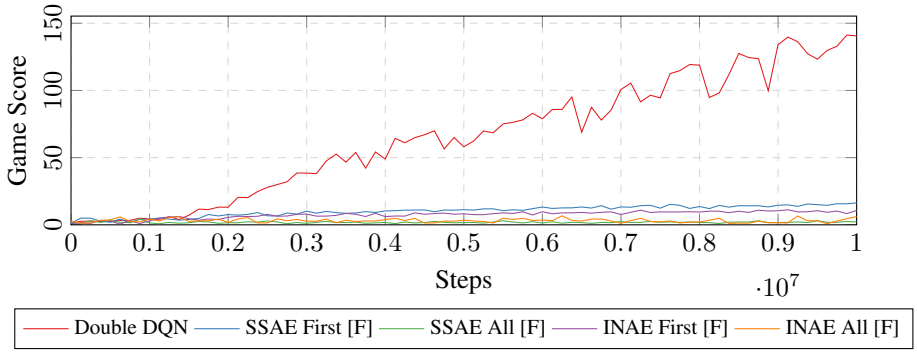


(b) Q*Bert

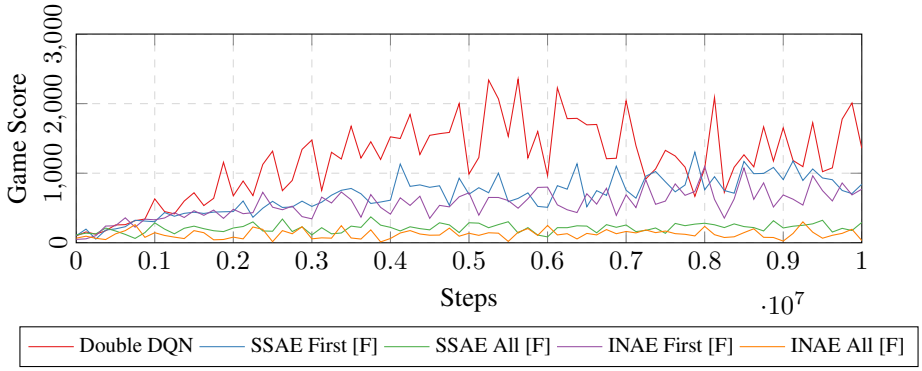


(c) Space Invaders

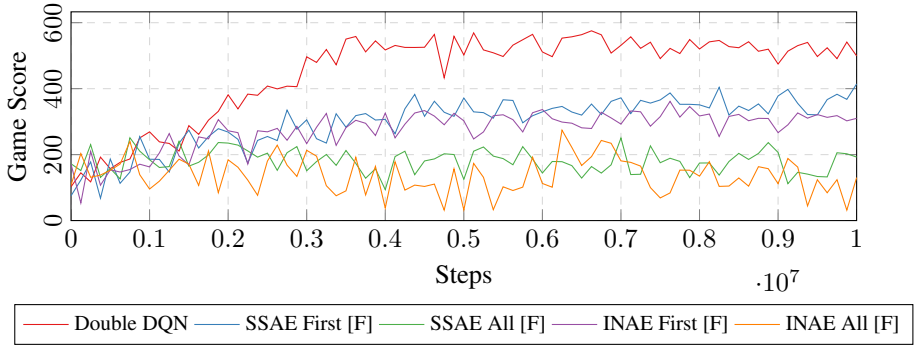
Figure 6.1: Repeats figure 5.1 without smoothing.



(a) Breakout

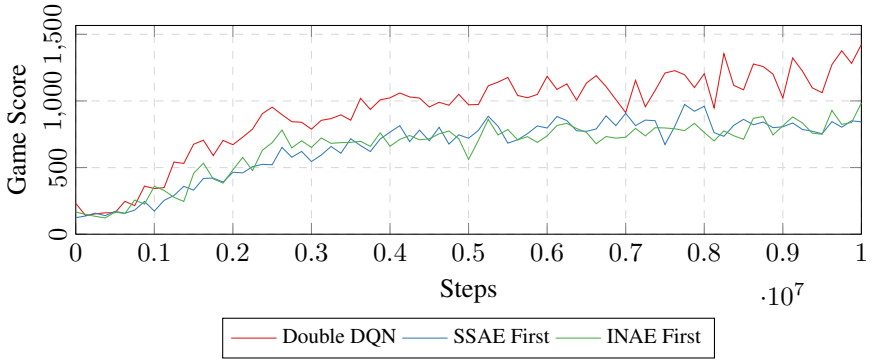


(b) Q*bert

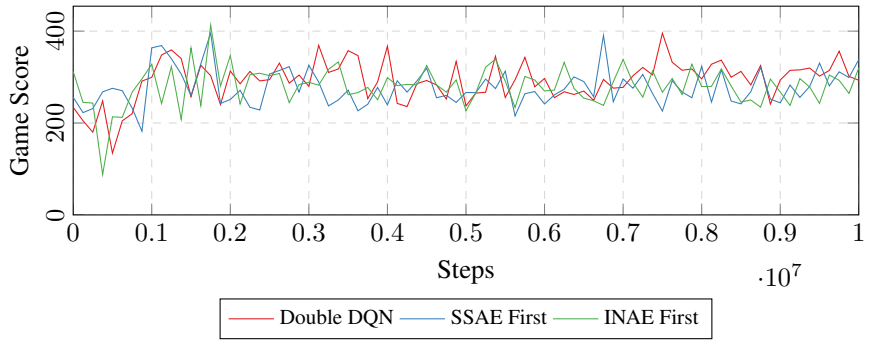


(c) Space Invaders

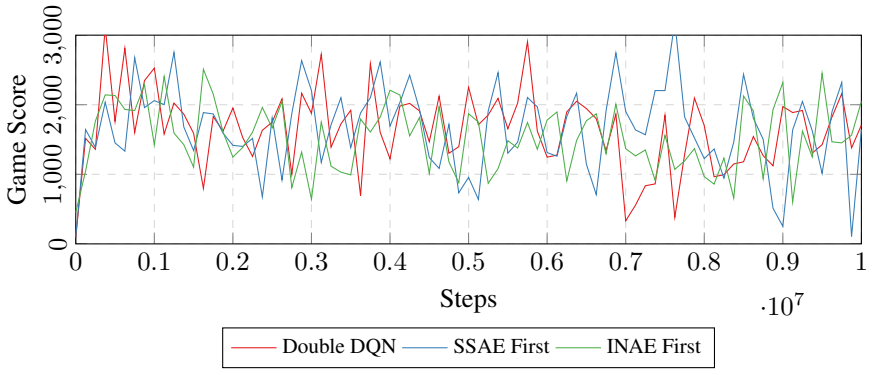
Figure 6.2: Repeats figure 5.4 without smoothing.



(a) Alien

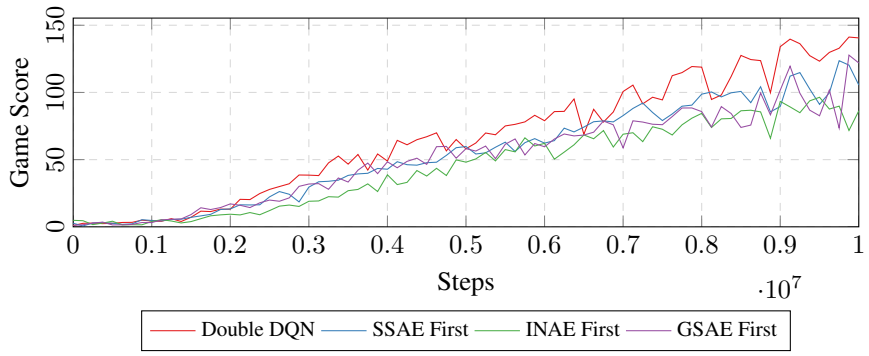


(b) Berzerk

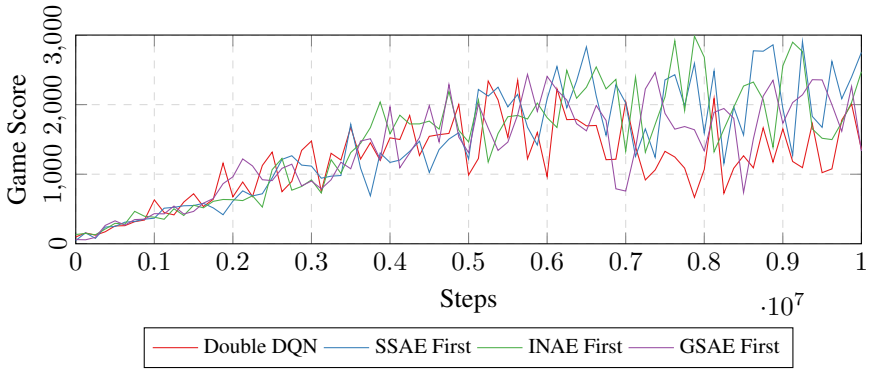


(c) Krull

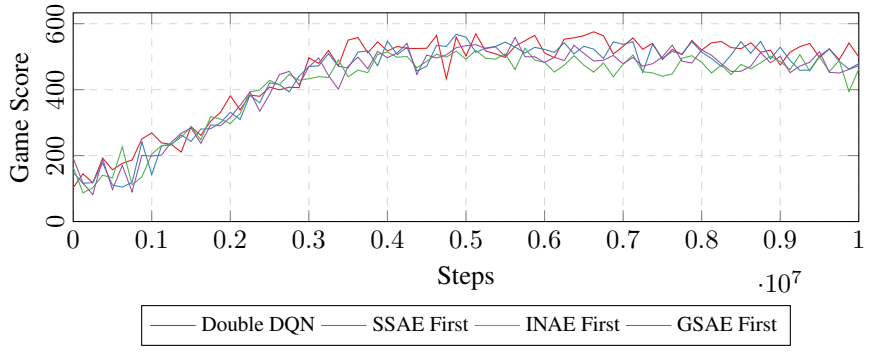
Figure 6.3: Repeats figure 5.6 without smoothing.



(a) Breakout



(b) Q*bert



(c) Space Invaders

Figure 6.4: Repeats figure 5.7 without smoothing.

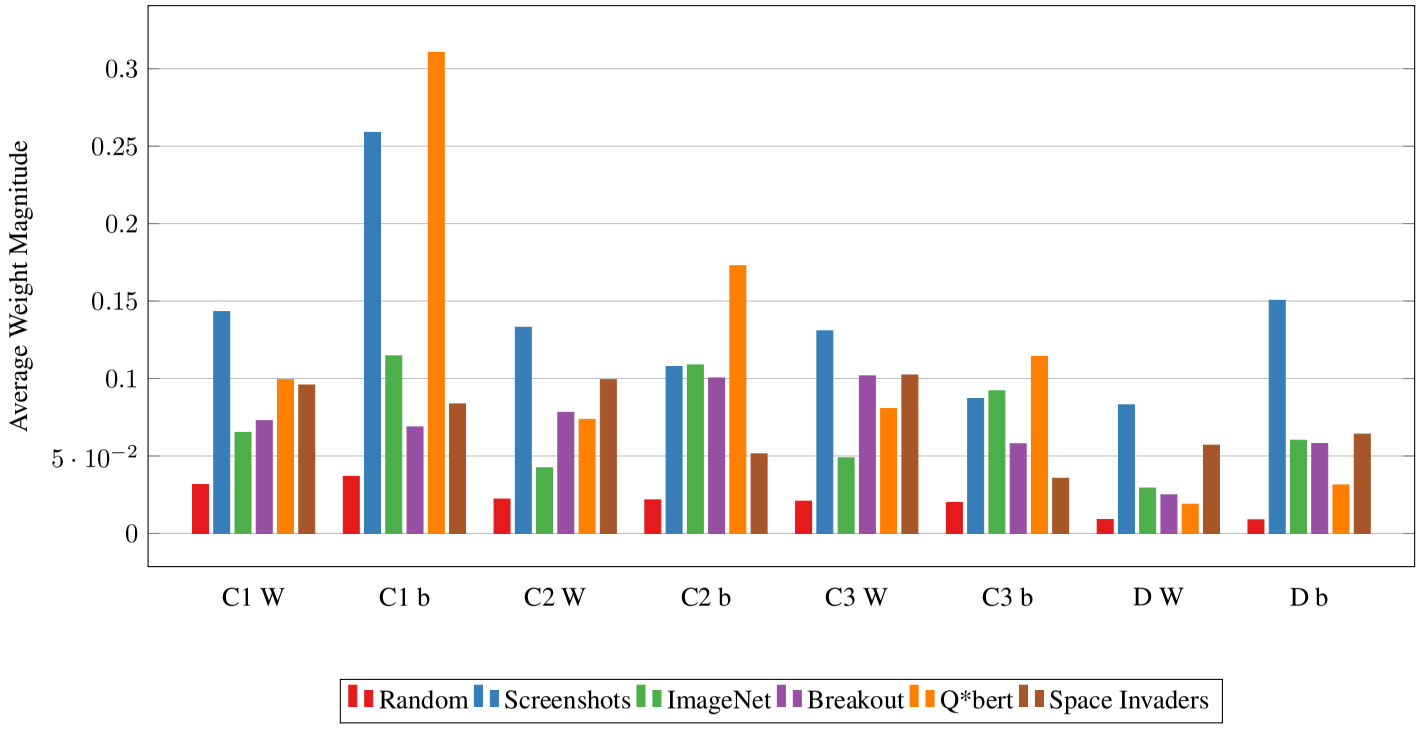


Figure 6.5: Shows the average weight magnitude for different staring points of the DQN algorithm. Random is the standard weight initialization method. Screenshots and ImageNet is the trained AE weights which are copied to the DQN network. 'C1 W' plots the bar chart for filters in the first convolutional layer, 'C1 b' is the biases for the first layer.