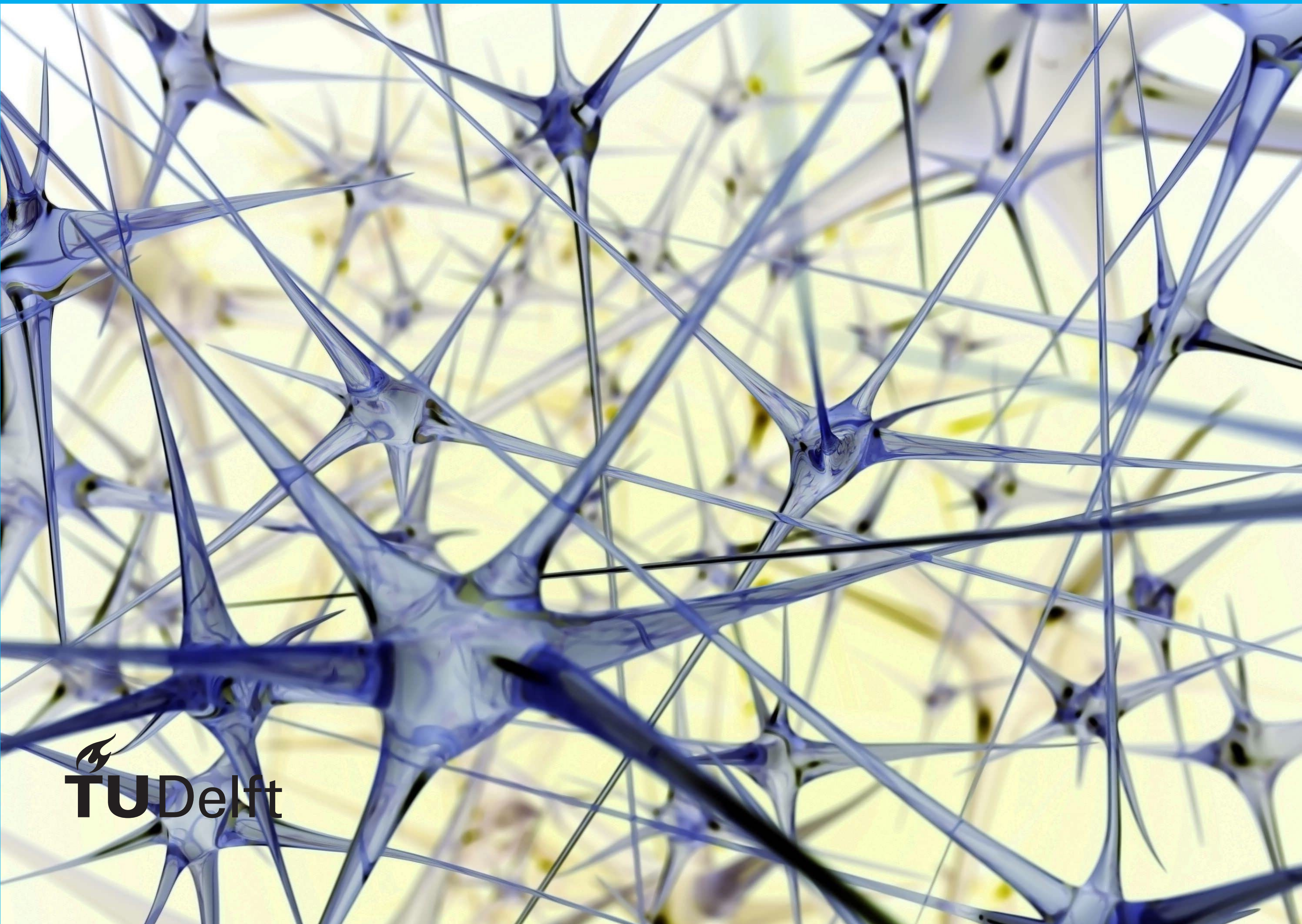
Reinforcement Learning of Visual Features

W. B. Blom

Department : Biomechanical Engineering
Study : Mechanical Engineering
Track : Biomechanical Design
Specialisation : Biorobotics
Coaches : ir. F. Gaisser and ir. A. A. Balasubramanian
Professor : Prof. dr. ir. P. P. Jonker
Type of report : Master Thesis
Date : 27 May 2016



Reinforcement Learning of Visual Features

by

W. B. Blom

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday May 27, 2016 at 1:30 PM.

Student number:	4003373	
Project duration:	September 1, 2014 – May 27, 2016	
Thesis committee:	Prof. dr. ir. P. P. Jonker,	TU Delft, supervisor
	ir. F. Gaisser,	TU Delft
	ir. A. A. Balasubramanian,	TU Delft
	dr. ir. J. Kober,	TU Delft

This thesis is confidential and cannot be made public until the day before the defense.

TU Delft

Abstract

The digital environment has an ever increasing amount smart programs. Programs that also get smarter every day. They help us filtering spam e-mail and they adjust to show us personalized advertisements. These smart programs observe people and serve (other) people. A robot can be seen as a program with a body. Make the program smart enough and it can help us in the real world too. The smartest programs learn from observations to become better at what they do.

Reinforcement Learning (RL) is a type of learning that has been successfully applied to solve a variety of learning tasks. RL is learning from experience in sensory changes and rewards. The robot that uses RL tries to optimize the actions it takes to achieve the maximum reward.

Most RL algorithms do not scale well to large sensory inputs. Images are very large inputs because each pixel is an input. Therefore algorithms have been created to compress the visual information to abstract representations (Visual Features). Neural Q-learning [12] is such a method. It combines the RL algorithm of Q-learning with Artificial Neural Networks (ANNs). ANNs are networks of neurons that each do a small adjustable calculation. The network can transform the input to more abstract or useful information. The ANN can learn by adjusting and optimizing the calculations until the network creates the desired transformation. Using an ANN is a good method to find complex ways to make visual data more abstract and more compressed.

In this theses, Deep Q-learning is tested with a more difficult task combined with a higher world complexity. In the original paper it was tested on ATARI 2600 games and achieved good results. In this thesis Deep Q-learning is tested in a transportation task in a 3D simulation where the learner only has a relatively large first-person perspective image of the robot it controls.

The results show that the complexity of the visual information and the relatively long-delayed reinforcements cause an initialization-noise to reinforcement-signal ratio such that the learner was unable to converge the neural network to describe beneficial behavior. What was learned was forgotten faster than the learner could replay the useful experiences.

It can be concluded that only scaling the environment complexity with the Neural Q-learning algorithm is not possible. The learning algorithm needs an extension that makes it better able to handle long-delayed rewards with large visual inputs.

Preface & acknowledgments

This project started with the search for a thesis subject in the categories of robots and learning. I always wanted to learn about these subjects. Studying mechanical engineering and helping in the Delft BioRobotics Laboratory helped to gain knowledge about robots. Knowledge about robot learning was harder to get by.

Dr. Ir. Wouter Caarls was the person to get to with question about reinforcement learning and I thank him to have the subject of this thesis ready to go. He coached me during the literature survey.

Secondly, I would like to express my deepest gratitude to my supervisors Floris Gaisser and Aswin Balasubramanian who took over from Wouter Caarls when he left to another university. We had weekly discussions for over a year about the research and the progress.

Third, I would like to thank Prof. Pieter Jonker for being the supervising professor and for guiding the final iterations of this report.

Furthermore, I want to thank Assistant Prof. Jens Kober from DCSC for completing the thesis committee.

Last but not least, I would like to thank my father for lending me his new (and fast) laptop to do parallel tests and thanks to my friends for support and distractions.

W. B. Blom
Delft, 14 May 2016

Abbreviations

ANN	Artificial Neural Network
MDP	Markov Decision Process
POMDP	Partially Observable Markov Decision Process
RL	Reinforcement Learning

Nomenclature

a	Action
b	Neural activation bias
r	Reward signal
s	State
t	Time
w_i	Synaptic weight corresponding to x_i
x_i	Neural input i
y	Output signal
$E(x, w, d)$	Error function of an ANN
$O(x, w)$	Neural output signal of an output neuron
$Q(s)$	Action-value function
$Q^*(s, a)$	Optimal action-value function
$R(s, a)$	Reward function
$V(s)$	Value function
$V^*(s)$	Optimal value function
α	Constant step-size parameter
γ	Discount factor
η	Step-size parameter in ANNs
π	Policy
π^*	Optimal policy
φ	Neural activation function

Contents

List of Figures

1

Introduction

Robots are being developed to help us with many different tasks. These tasks are mainly dull, dirty, dangerous and/or impossible for humans. Highly repetitive tasks, like welding a specific car part numerous times, are dull. Cleaning sewers can be less dull, but it is dirty. Checking the inside of a nuclear reactor is less dirty, but it is lethal. Most people would prefer to let someone or something else do such tasks, while other tasks are simply impossible to do. It is impossible to go through a small hole or pipe, but robots can be shaped to do so.

Robots sense, think and act

Robots can be classified according to autonomy in performing the task and how multipurpose they are. Robot autonomy is defined by how the robot adapts to new situations. An industrial welding robot is not autonomous, it always performs the same actions in the same order. This robot is also very dedicated to one task. On the other side of spectrum there is for example a Mars-rover. It is multi-purpose to be able to perform many different tests and observations for the researchers and it has to be autonomous due to communication limits back to earth. Based on a given objective, the robot has to think for itself and determine which of its many actions it should do, e.g. moving around a crater.

For a robot to be autonomous it needs a method of control. This is typically done by use of the ’sense-think-act’ paradigm which is a functional approach to making classical robots autonomous: A robot senses the world, extracting information about the situation, then thinks about what it what it should do towards a certain goal and finally acts on the world. This is then repeated.

Sensing is not only receiving the input from sensors such as cameras. It is also the processing of that information to more abstract forms of describing the world around the robot. With image processing features can be recognized. These features can be described with much less data than the raw image data. This processing is needed to make it easier to make a decision based on this information, because abstract descriptions are easier to use in logic.

Acting is the conversion from a decision to a real change in the world. For example, after a robot decides to go full speed, the controllers will translate this to how much power should go to the wheels. Or the robot can decide to go to a location. The pathplanner will then determine when and where the robot has to turn and step to get to that location.

Thinking, or decision making, is the step where the robot translates the situational information to decisions. Tasks, goals and dangers are weighed against each other to determine the best next action. For example the task can be to clean a room. There are many ways to complete this. The goal can be to do this fast and efficiently and on the path to complete the task the robot encounters obstacles such as walls to collide with.

After each action the world changes and the robot needs to sense again. These changes can be deterministic: a given action in a given state of the world always results in the same next state of the world. In the real world the changes in the world are probabilistic.

In modern probabilistic robotics each robot action in the real world does not always have the same result[15]. There is always the probability that a step forward is not straight forward. With repeated sensing of the environment these deviations can be picked up. Therefore, by use of the sense-think-act cycle a robot can correct its plan towards a goal when it deviates too far.

Robot learning

Strictly programming how a robot should sense, think and act can take a lot of time and takes much experience. A robot can also be taught to do any or all of these steps. Learning is the opposite. A learning robot with enough learning capacity only has to be programmed once to be applicable to many different tasks. Today's learning systems still have relatively low learning capacities.

Generally, a learning robot is provided with a function approximator and a learning rule. The approximated function is what translates for example the situational information to decisions. It is adjusted and optimized to perform better with each experience according to the learning rule. The learning rule evaluates the input-output behavior of the function according to for example the robot task, goals and dangers.

Learning how to perform actions can be done to create complex motor controllers. For example the learned function translates the decision, to go to a location, to how much power should go to the robot wheels. The learning rule is then set to evaluate the function such that it minimizes the the distance to the desired location.

The same can be done to learn the appropriate processing of sensor information. The learned function is what compresses for example the images of the camera to a more compact, useful and abstract representation to describe the visual information. The generic term for the processed visual information is called visual features. The learning rule for this image processing is less straight forward, because it only affects robot behavior indirectly. If not enough visual features can be learned, the 'think' part has not enough information to make the best decisions. But, if the computer tries to put all visual information into visual features, there will be too many features of which some can also be useless for decision making.

There are many ways available to implement the learned approximation function. An Artificial Neural Network (ANN) is such a function approximator. It can be shaped in size for any of the sense, think and act steps. The function consists of layered groups of neurons. An artificial neuron processes information by weighing its inputs and then combining them in a certain way to one output, also called an activation. By weighing its inputs the neuron can find structures like lines in an image. If the line is found, the output is set to 'true'. The weights for each input is what is learned to be able to recognize different structures.

A group of parallel neurons is called a layer. Each neuron can then recognize different structures, combining the image into a more abstract form, visual features. A common ANN has three layers: First an input layer which only represents all network inputs. Second, a hidden layer which applies the first abstraction, creating features. Third, the output layer which produces the network output, also called the classification step. For example each output neuron can represent a possible decision. Figure 1.1 shows an example schematic representation of neurons in layers in an ANN.

In some cases one hidden layer is not enough. Deep ANNs have multiple hidden layers to

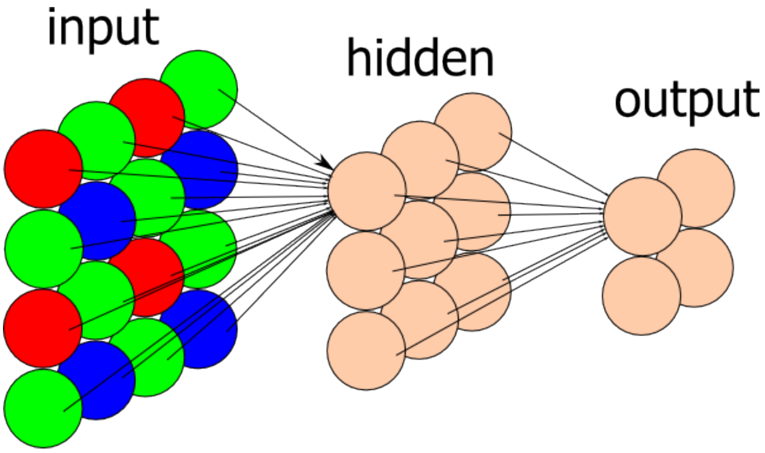


Figure 1.1: Graphical representation of a common neural network: input layer (e.g. RGB pixel values), hidden layer and output layer. Fully connected neurons (only shown for 2 neurons) and no feedback (loop) connections.

create higher order abstractions. For example to recognize more complex objects from the simple lines recognized in the first hidden layer.

The learning rule for improving the weights of the ANN depends on the type of learning. Learning is categorized based on how much supervision is needed. The category of Supervised Learning systems need immediate feedback for each output. What is learned is therefore determined outside the learning system.

The opposite, Unsupervised Learning systems, needs no feedback because they have the learning rule within the system. These learning systems try to learn an abstraction that holds as much information as possible given the desired output size. This can be useful if noise needs to be removed from the input.

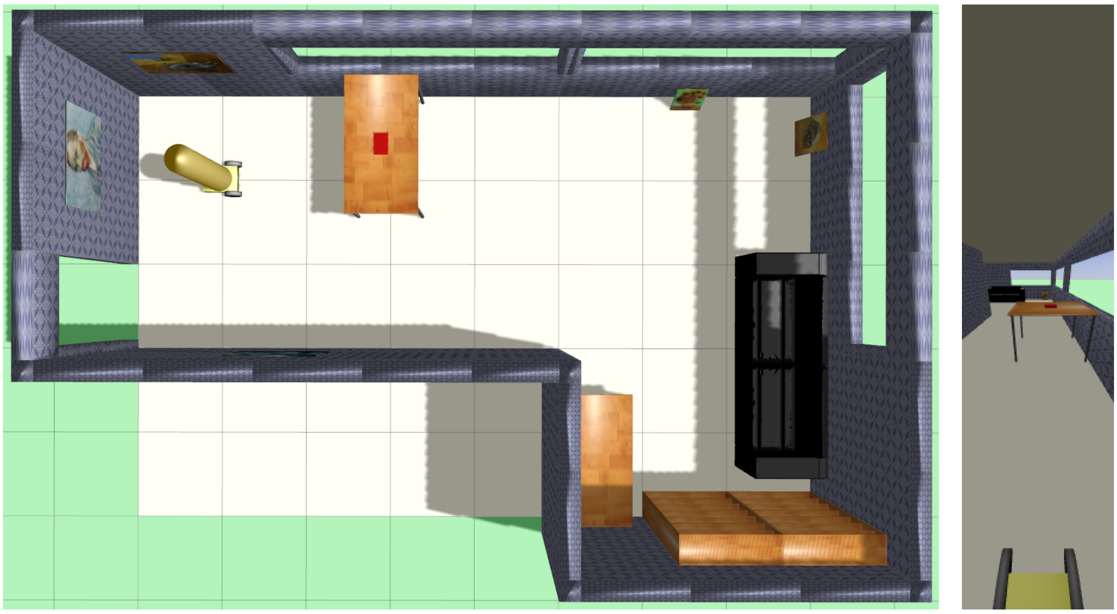
Reinforcement Learning (RL) lies in between supervised and unsupervised learning. It is partially supervised through delayed rewards. RL is well suited for robots, because through rewards the robot task can be determined from the outside while no feedback is needed on how the robot gets to the reward. Internally, the RL system learns which actions are more suitable to get the maximum out of the external rewards.

The dimensionality problem

Most RL algorithms have limits. A major problem is that they are not able to handle large inputs like visual input streams. These algorithms are designed to handle only a handful of inputs, such as only the location of a robot. The main problem is in gathering experience. The algorithm needs to know what to do for every possible location. With a single location this is only a two or three dimensional problem. A look-up table can be learned to store how profitable each move is in every location. The look-up table is another example of the approximation function in learning.

If the input is changed to a camera image, every color value (i.e. RGB) of every pixel is a separate input to the learning algorithm. To store the experience of the RL algorithm, a high dimensional table would be needed. For example, 300 pixels × 400 pixels × 3 color values = 360,000 input dimensions compared to the 2 or 3 dimensions of the location problem. Gathering enough experience in the real world to fill in most of this table takes significantly more time. This is called the dimensionality problem. Images of this size are needed in real-world applications like room cleaning, because also a human is unable to distinguish similar objects when there are only few pixels.

Using an ANN in RL, instead of a look-up table, counters the dimensionality problem partially.



(a) Top view of the simulated room with the ceiling removed.

(b) Visual input of robot

Figure 1.2: Simulated environment

An ANN can do the same with less variables and less variables require less experiences to learn from. This is possible because an ANN calculates the output with a learned function instead of a learned value for every different situation.

Therefore sensing, thinking and acting can be combined into one learning system by using ANNs in RL. For this a deep ANN might be useful. The first hidden layers can extract visual features, such as complex objects and their locations. The higher hidden layers can use these features to make decisions. Applying RL on both sensing and thinking with the same ANN also solves the problem of figuring out what visual features need to be learned, because these are now learned depending on what is useful for the best network output.

Deep Q-learning is a method that combines deep ANNs with RL. In ‘Playing Atari with Deep Reinforcement Learning’ [12] Mnih et al. produced such an algorithm to let a computer play Atari games. For example Space Invaders. The visual input of 84 × 84 pixels is a large input, that is used directly for an ANN that also does the ’think’ part with RL.

Thesis experiment

In this theses, Deep Q-learning is tested with a more difficult task combined with a higher world complexity. The world is a single simplified ’living room’ wherein a robot gets visual data and is able to move around, pick-up objects and put-down objects. The robot’s task is to pick-up a book and put it down on a bookshelf.

World complexity is the complexity of what is detectable in the environment. This includes first the size and dimensions (2D or 3D) of the environment. Second, the complexity and number of different objects, such as tables, sofas, paintings, plants, etc. Third, the way the learning system can see the environment: Resolution of the image data, the amount of noise and the size of the colorspace. More of each property increases the world complexity. Experiments in this thesis are done with a 3D simulation (Figure 1.2a) and a larger image size than in the Atari experiment (Figure 1.2b), but with a limited amount of objects.

The task complexity is defined as how much effort (or calculations) the learner has to put into completing the task successfully. This is influenced by the input size, the amount of actions possible and the amount of actions needed for success. A larger input space means there is more to be filtered and more to be compressed into something useful. A larger action space means more options to explore. The main increase in task complexity is an increase in the amount of actions needed for success. The simulation in this thesis teaches and tests a virtual robot with a 'clean-up' task. Its goal is to find and pick-up a book and then put that book on a bookshelf.

Deep Q-learning can have problems with an increased input size and complexity. More inputs means more information to consider which requires more neurons. More complex input means more levels of abstraction are required, therefore even more layers of neurons are needed. This strengthens the dimensionality problem, because there are more variables to be learned. In chapter 3 one possible improvement is presented for when 100% RL is not enough.

Overview

This report is structured as follows:

Chapter 2: A more detailed background to the topics of visual features, robotic learning and their implementations.

Chapter 3: The experiments are described in detail. Details about the sizes of the ANN, the mechanisms of the simulation and the definition of the feedback to the learner.

Chapter 4: Presentation of the results and findings including follow-up experiments.

Chapter 5: A discussion of the mechanisms behind the results.

Chapter 6: A summary and the conclusion of this report.

2

Literature

This chapter is an introduction to the background literature for this thesis. The three sections are about what visual features are (section 2.1), the different learning types (section 2.2) and how this can be implemented (section 2.3). All three sections are limited to what is relevant to understand this report. Section 2.3 also describes two methods from papers. These methods form the basis for this research.

2.1. Visual Features

Visual features are derived from a visual input. They are a more compact and abstract way to selectively represent visual information. A feature could represent an edge or corner point in an image. That is a low-level feature. Low level features can be calculated with relatively small calculations.

In many applications higher abstractions are needed, selectively compressing the visual information further. Combining the edges to shapes and combining these shapes to recognize objects create high-level features. High-level features are more useful and usable for robots. If a robot needs to know if there is a wall less than 20cm in front of its wheels its controller only has to check the bit that represents that visual feature.

Visual features for robots can be static or dynamic. Static features are designed by hand. For example, when the robot needs a feature to recognize if there is red book in the picture. It can be calculated with an RGB to HSV colorspace transformation and then a check for pixels within the range of red for hue. Dynamic features are, for example, learned features. Dynamic features are useful when it is initially not known what features are needed.

Many different methods have been developed for creating visual features, some more simple than others. In this thesis the convolution is the most used feature calculation method which is described in subsection 2.1.1.

2.1.1. Convolution on images

Convolution on images, also called image filtering, is a useful calculation in the field of processing digital images[10]. The filters are smaller than the image and test for a pattern on every possible location. The advantage here is that a small filter contains few variables which are applied many times in the image. Second, the output contains location information: one number for every location for every filter that describes if the pattern is recognized there, or not. The equation is

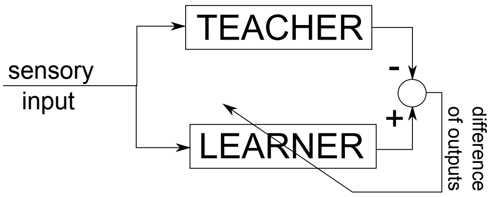


Figure 2.1: Diagram of supervised learning

shown below:

$$O = I * f \tag{2.1}$$

$$O(r,c) = \sum_{i=0}^{n} \left(\sum_{j=0}^{m} \big(I(r+i, c+j) * f(i,j) \big) \right) \tag{2.2}$$

$$O(r,c) = \sum_{i=0}^{n} \left(\sum_{j=0}^{m} \left(\sum_{k=0}^{p} \big(I(r+i, c+j, k) * f(i,j,k) \big) \right) \right) \tag{2.3}$$

In equation 2.1 to 2.3 the input image is represented by matrix I, the filter by matrix f of size $n \times m$ and the convolution output by matrix O. Equation 2.2 shows how to calculate the output value at row r and column c. If the image has multiple channels for color, then equation 2.3 shows the output value where k indicates a color channel (like red in RGB channels).

Using one filter is not enough in most applications. One filter will for example only recognize a vertical line, but not a horizontal line. One can use as many filters as desired, also called a filter bank. Each output, a filter applied at a specific location, is an example of a visual feature.

2.2. Robotic Learning

Robot learning can be categorized into three types[5]: supervised learning, unsupervised learning and reinforcement learning. The first one is learning with a frame of reference and the latter two are learning with internal feedback. The three types of learning are described in more detail in the following subsections.

2.2.1. Supervised learning

Supervised learning is learning with a teacher (Figure 2.1). This teacher constantly gives the desired output of the learning system. Training is done by a large or complete set of training inputs. Then the trainer and learner give an output. The learner is updated to minimize the difference between the outputs. This can be done with a batch of experience with a single learning step or iteratively. when the difference between outputs has come to a minimum, the learning process is done and the system should be able to do the task on its own.

A common experiment set-up is to create large set different input-output combinations. A large part is used as the training set, e.g. 75%. The rest is for testing. Multiple sessions of training and testing can be done by re-assigning input-output combinations randomly to either the training set or the test set between sessions.

Supervised learning has the following components in the case of learning visual features from images:

- A large set of images that covers most of the possible visual inputs.
- A set of desired visual features. For example the position, size and color of the desired object from the environment.

Figure 2.2: Diagram of unsupervised learning

- And for every training image the desired learning system output in the form of the values of this set of features.

The visual features created here are useful, but they are predefined by the programmer. Therefore the set of visual features could have a feature too much or a feature missing for the task at hand.

2.2.2. Unsupervised learning

Unsupervised learning has no external system like a teacher (Figure 2.2). This type of learning is also called self-organized learning [1]. The learning system tries to create an output set of uncorrelated features that together describe the set of training inputs as much as possible. The feedback to the system is internal feedback according to a learning rule. This learning rule is task-independent. The output of the learning system can be the same as with supervised learning, but now the output labels have to be determined instead of them being provided by a teacher.

In the case of learning visual features, unsupervised learning only needs a large set of training images that covers most of the possible visual inputs. The learning algorithm will then try to categorize the training images.

The set of visual features created here should have all the information that can be extracted from the input, but that doesn't mean that all features are useful for the task at hand. Some features are not useful because there is no feedback on usefulness during the creation of these features.

After learning, the categorization method can be used as an inflexible method to reduce the amount of information passed to the next learning step (i.e. control of actions).

Auto-Encoders

One way to do unsupervised learning is with an Auto-Encoder. An encoder is a program that has a database with structures it can recognize [6][16][9]. A structure can for example be a group of pixels that show a vertical line or some other line/shape at some angle. The combination of pixel intensities has a code. The encoder then creates a small binary image that shows where that specific structure exists within the image. Each structure gets its own binary image. If the structures are of a 3×3 gray pixel form, then there are $3 \times 3 \times 256 = 2304$ possible structures. It is possible to encode with less than that. A large part of the possibilities resemble noise or are almost equal, therefore it is a good method for information and noise reduction. Figure 2.3 shows some 10×10 structures of a trained auto-encoder. Most of these structures encode diagonal lines, but some of them (also) show ends, corners or round edges.

Auto-encoders have an adaptive database of structures to test for. That means that the information in the database is not predefined, but shaped with every new experience. Unsupervised learning with an auto-encoder is done by decoding the output back to a reconstruction of the original input. The difference between the input and the reconstruction is then used as the error that needs to be minimized.

Encoding can be done in series for further abstractions. This way combinations of lines can be recognized to detect objects[16].

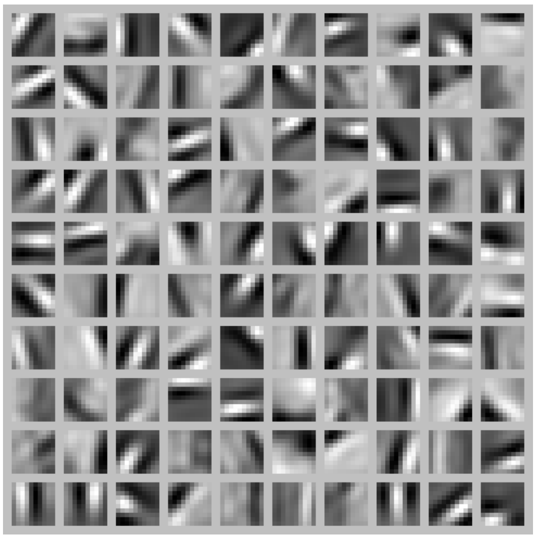


Figure 2.3: Visualization of the structures of a trained auto-encoder.

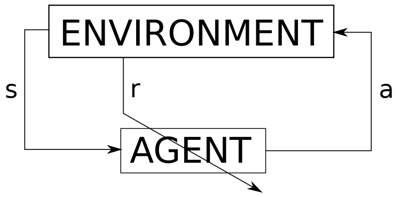


Figure 2.4: Diagram of Reinforcement learning with signal symbols

2.2.3. Reinforcement learning

Reinforcement learning (RL) is learning through interaction. RL lies between supervised and unsupervised learning in terms of feedback. The feedback from the environment comes in the form of reinforcement. A reinforcement can be reward or punishment after finishing a task. Feedback during the task can exist for every step, for some situations or only on the end of the task. Because of this, experience from sensory input and reinforcement can be stored to be able to decide later on what actions are better than others.

To learn visual features with RL, the sense and think parts of the 'sense-think-act' cycle (chapter 1) need to be combined. Only then the feedback from reinforcements can affect the creation of visual features. An advantage of combining sensing and thinking, is that the created visual features are all useful for the task at hand.

Because the focus of this survey is on using reinforcement learning, more detail will be given on the elements, signals and problem types that exist within RL. This is only a short introduction, a more complete description can be found in [14].

The elements and signals of RL

A reinforcement learning problem consists of two systems (Figure 2.4) called the agent and the environment. The agent can best be described as the brain while the environment can be described as the rest of the agent's body and what the agent is interacting with. The agent receives the state and reward through it's sensors and acts upon that. That action can change the state and reward it gets the next moment.

The signal of the state (s) is a list of available information about the agent and the environment that should contain how the world is now and how it got in this state. When there is only a limited amount information available it is possible that the agent cannot fulfil it's task. Therefore a good system design is key. The action signal (a) is very similar to the state signal, but it tells the environment what the agent is doing to it. For example it can be movements of joints of a robot, sentences from a chatbot or the next action in a pc game. The reward signal (r) is just one value, because it tells how good the agent is doing at achieving the goal. The main focus of the agent is to maximize the cumulative reward. An experience is a combination of a perceived state, a taken action and a received reward.

The agent and environment have four important subelements: The reward function, the value function, the policy and a model.

- The reward function $(R(s,a))$ translates the state of the world and actions of the agent into the reward signal. Usually, an observer test the world for specific features to determine the reward. For example, the observer tests for the location of the robot when its goal is to go to that location. Negative rewards can be given as punishment. For example, for when the robot drains its batteries too fast.
- The value function $(V(s)$ or $Q(s,a))$ works as a function approximation to be able to optimize expected reward based on past experience. The value function's output is a measure of how much reward can be expected when going through the current state.

  $$V(s) = E\left[\sum_{k=0}^{\infty} r_k|_{s_0=s}\right]$$

  It is constantly updated to improve correctness of the expectations.
- The policy $(\pi(s)$ or $\pi(a|s))$ chooses actions based on the state or output of the value function. In a more abstract way it also chooses between exploration and exploitation. Exploitation would be a using a 'greedy' policy which always takes the most valuable action based on value function. Exploration would be, for example, using a random-action policy which eventually explores all possible experiences.
- A model $(T(s,a))$ is optional. It is a model of how (part of) the world works. A model can be used to shape the value function and improve the action selection. For example, preprocessing of visual information to abstract visual features instead of learning these.

  $$T(s,a) \rightarrow s_{\text{for learning}}$$

The RL problem

An important property of reinforcement learning problems is whether the problem is Markovian. The Markov property is fulfilled if the state contains a summary of all information of current and past sensations that is sufficient for completing the task. If a reinforcement learning problem has this property it is called a Markov Decision Process (MDP).

If the state is missing some parts of the full state, it is called a Partially Observable MDP (POMDP). POMDPs can still be solvable because the missing information can be estimated. The missing information can for example be logically deduced (i.e. only one action can result in the current state) or calculated (i.e. velocity can be estimated from change in value given the time step).

The goal of the reinforcement learning task is finding or approximating the optimal policy (π^*). That is the policy that results in the highest cumulative reward. For this the policy needs the optimal state-value function ($V^*(s)$) or the optimal action-value function ($Q^*(s,a)$). These are defined as in Equation 2.4 and 2.5.

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{2.4}$$

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{2.5}$$

$$\pi = arg \max_{a} Q(s,a) \tag{2.6}$$

V^* can also be generated from Q^* by taking the best actions in every state:

$$V^*(s) = \max_{a} Q^{\pi^*}(s,a) \tag{2.7}$$

Q-learning

Q-learning is one of the methods to update the action-value function. The update rule is described in Equation 2.8 where α is the learning rate and γ is the discount factor. A larger learning rate causes the learner to make larger updates, but it can also cause overshoot and instability when set too large. A larger discount factor causes more states around a reward to be of high value when the reward back-propagates through the experiences. The learner will ignore smaller rewards when the discount factor is too high.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{a} Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2.8}$$

The strengths of Q-learning are that it is:

- without model: No pre-learning knowledge needs to be implemented.
- Incremental: Learning while experiencing is possible.
- off-policy: Optimization of Q is independent of how actions are chosen.

Q-learning also has drawbacks. Because it makes an estimate of an estimate, also known as bootstrapping, a corrupted guess has a long term effect. Also the agent needs to reach the goal multiple times to create an estimate that covers the whole trip to it. These two drawbacks can be made less prominent by introducing eligibility traces.

When using eligibility traces with Q-learning all previous steps from the episode are updated for every time step. Steps that were taken longer ago had less to do with the current (expected) reward and have therefore a smaller update. The parameter λ ($0 < \lambda < 1$) determines how much $Q(s_{t-1}, a_{t-1})$ is less important than $Q(s_t, a_t)$.

With eligibility traces a corrupted action value is not only updated once every visit, but every visit and the rest of the episode. And the action-value function converges much faster because the whole trip to the goal is updated. For example: If the agent has to take 20 steps to the goal, previously it took 20 trials to update all the values for these 20 states at least once in the optimal direction. With eligibility traces all 20 values are updated in the same trial (if λ is not too small).

2.3. Implementations

In this section several methods of implementing a learning algorithm are presented. In subsection 2.3.1 the basic components and the dimensionality problem is presented. One solution is an artificial neural network, which is explained in more detail in subsection 2.3.2. Subsections 2.3.3 and 2.3.4 describe two papers which use a network with visual input.

2.3.1. Basic components and common problems

Making a computer learn is similar to solving an optimization. There is the optimization function that calculates an output for every input. This function has variables which form the optimization space. The variables are updated according to an optimization method. A small example is the situation of a computer learning (Q-learning) to swing a pendulum to the up position:

1. The input/state is two numbers. The angle and the angular velocity of the pendulum.
2. The outputs are several actions which tell how much torque is applied to the pendulum.
3. The optimization function is the Action-Value function. This can be implemented as a 3-dimensional table. By discretizing the inputs one can look-up the action-values.
4. Using the policy an action is chosen (random or by using the action-values).
5. The Q-learning learning rule is applied to calculate a better action-value for the action in the starting state. The learning rule can be seen here as the optimization method.

In the case of the pendulum there was a small optimization space. A look-up table was sufficient here. In the introduction was shown that if the input is scaled up, so does the look-up table in dimensions. It takes too much time to gather enough experiences to fill the whole table when there are a million inputs. One solution to this dimensionality problem is replacing the look-up function with an artificial neural network (ANN). How this ANN works will be explained in the following subsection (2.3.2). Here it is just a non-linear function that has variables for optimization. The inputs can be used as given by the environment. The output is the same: action values.

When using an ANN the learning rule is not enough to update the action-value function. The learning rule only calculates the desired output of the network. The difference between this desired output and the previously calculated output is called the error. The optimization method of the network then calculates the updated variables using this error.

2.3.2. Artificial neural networks

The field of Artificial Neural Networks (ANNs) tries to imitate the biological brain. Some researchers use it to test their theories of how the human brain or part of it works. A lot of research on reinforcement learning and visual feature learning therefore incorporate ANNs to model for example the visual cortex[3][17] or the reinforcement signals from neurotransmitters[13].

The information in this subsection comes from the book “Neural Networks and Learning Machines” by Simon Haykin[5]. In the text below will first address the basic definitions of the artificial neuron and neural network. This is followed by an explanation on how ANNs can be used for reinforcement learning and learning of visual features.

The basic artificial neuron

The human neuron has three important parts: the dendrites, the body and the axon with its synaptic terminals. Multiple input signals come in through the dendrites and after processing one output goes out through the axon. The output signal is then input to dendrites of other neurons. These connections between neurons have a strength. If the connection is not used enough the connection becomes weak and finally breaks. An example of weak or broken connections is when a person does not rehearse a task and then the person forgets how to do it.

The basic artificial neuron is not very different. There is a set of input signals x_i, which are strengthened or weakened by multiplication with their synaptic weights w_i. The weighted signals are then summed up in a linear combiner together with a bias b. That will become the input to the activation function $\varphi()$. The activation function determines how strong the output or axon will

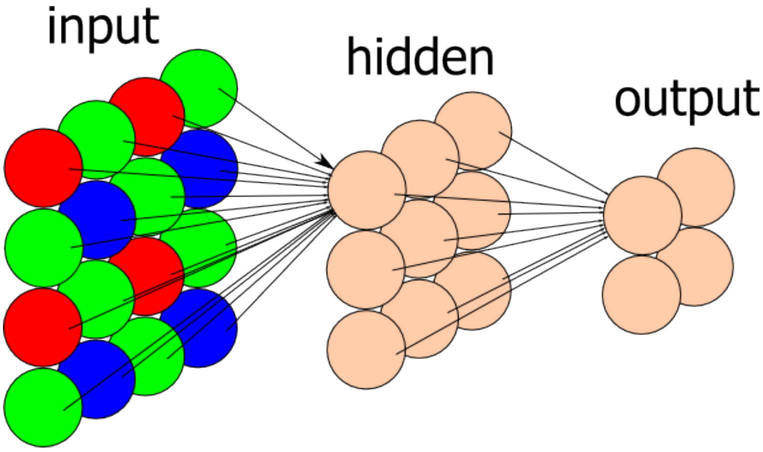


Figure 2.5: Graphical representation of a common neural network: input layer (e.g. RGB pixel values), hidden layer and output layer. Fully connected neurons (only shown for 2 neurons) and no feedback (loop) connections.

fire. Typically the output y is somewhere between 0 and 1. A threshold function will do the job, but that is hard to differentiate which is needed for updating the synaptic weights. So a function that approximates the threshold function in a continuous way like the sigmoid function is better. The workings of this type of artificial neuron can be described with the following function:

$$y = \varphi\left(b + \sum_{i=1}^{m} w_i x_i\right) \tag{2.9}$$

Where m is the number of input signals to this neuron.

The artificial neural network

A network of artificial neurons can take many forms, because there are many different applications with their own demands on the network. A common form of neural network is one with three layers (Figure 2.5): an input layer, a hidden layer and an output layer. The input layer can be seen as simple sensor cells that only have a neural output. The hidden layer is the first processing layer that uses all signals from the input layer. This layer has an arbitrary amount of neurons. The output layer is the second processing layer and has as many neurons as outputs. To read out the network, the signals are calculated in a feed-forward manner.

Larger networks contain more neurons per layer and more layers. When there is more than one hidden layer it is called a deep neural network. The name comes from the field of deep learning. An important subject of deep learning is creating high-level abstractions.

If the network allows for feedback connections the program gets the possibility to have a short term memory, to differentiate and to integrate. These are handy properties when analysing changes between instances in an input stream. These networks are called recurrent networks. The feedback connections are connections where the signal has the activation of the previous cycle.

The network learns by adjusting its weights every cycle after receiving feedback or reward. There are different methods to do this depending on the type of learning. If a desired output is known for a set of inputs (see subsection 2.2.1), the backpropagation algorithm can be applied for supervised learning of a layered feed-forward network. The weights are updated according to the error of the output using the gradient descent method:

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} \tag{2.10}$$

$$E(\mathbf{x}, \mathbf{w}, \mathbf{d}) = \sum_j \left(O_j(\mathbf{x}, \mathbf{w}) - d_j\right)^2 \tag{2.11}$$

Here j is the identifier of the neuron and O_j is the output of an output neuron with d_j as its desired output. η is the step-size parameter that determines how much is learned from an iteration.

The efficiency and capabilities of a network is also largely determined by how layers are connected to each other. The network in Figure 2.5 shows fully-connected layers. These layers have the maximum amount of connections to their previous layer: each neuron receives and weighs all the signals from the previous layer. It is also possible to have much less connections and have neural weights shared between neurons. One can construct the connection such that it calculates a special function.

Convolution (see subsection 2.1.1 is such a special function. These convolutional layers calculate the convolution of the input with filters made of neural weights. Each neuron represents a feature that used a specific filter on a specific group of inputs. One can also use local averaging, maximizing, minimizing in the same manner. These layers are called pooling layers. These layer types retain the location of the visual feature. Combined they make a strong tool for recognizing structures in images where the feature can have a little location randomness (e.g. with handwriting).

2.3.3. Deep Q-learning Networks

In the paper called 'Playing Atari with Deep Reinforcement Learning' V. Mnih et al. describe a learning method they call Deep Q-learning[12]. They combined Q-learning with deep neural networks by using the neural network for function approximation of the action-value function, which is updated using experience replay[10]. The algorithm and its strengths are described below to get a basic insight on how this method is used in the current experiment and why several design choices were made. The algorithm is repeated in algorithm 1. A detailed explanation on how the method works can be found in the paper ([12]).

In the paper Deep Q-learning is applied to several Atari computer games. The game provides raw image data which is cropped and color converted before used as state for the action-value function. The neural network consisted of two convolutional layers followed by two fully connected layers. The outputs were the action-values that represented how valuable each game action is.

Mnih et al. state the following advantages of using deep Q-learning:

- Better data efficiency, due to the potential many uses of each experience. This is a result of experience replay.
- No correlation between update samples, due to random batches for experience replay. Randomness breaks correlations and lowers variance compared to using only the latest experiences.
- Off-policy learning. Optimization of the action-value function is independent of how actions are chosen.

Deep Q-learning gets its strength from the experience replay. Random batch updates result in potential many uses of each experience (data efficiency), no correlation between batches and

```
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode = 1,M do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for t = 1,T do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a'; \theta))^2$ according to Equation 2.8
    end
end
```

Algorithm 1: Deep Q-learning with Experience Replay

off-policy learning. Therefore changing from random batches to batches of only the latest experiences would make no sense in the current experiment. That leaves the variables batch size and replay memory size to change what is learned from. If significant experiences have a low chance of occurring then the replay memory size must be large enough to have enough time to learn from it. But a larger replay memory lowers the chance of any experience to be chosen for the batch update. Increasing the batch size counters this effect through higher data efficiency.

2.3.4. Stacked Convolutional Auto-Encoders

In the paper called 'Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction' J. Masci et al. present a convolutional auto-encoder[11]. It is a convolutional layer as described in subsection 2.3.2 using the unsupervised update method for auto-encoders as described in Figure 2.2.2. The network is tested on written number detection and object detection. The equations from the paper are repeated below:

$$h_k = \varphi(x * W_k + b_k) \tag{2.12}$$

$$y = \varphi\left(\sum_{k \in H} h_k * \tilde{W}_k + c\right) \tag{2.13}$$

$$E(\theta) = \frac{1}{2n} \sum_{i=1}^{n} (x_i - y_i)^2 \tag{2.14}$$

$$\frac{\partial E(\theta)}{\partial W_k} = x * \delta h_k + h_k * \delta y \tag{2.15}$$

here x is the input image of which y is a reconstruction. h is the set of visual features where h_k is the set of visual features that was created with convolution filter W_k and corresponding bias b_k. The convolution in Equation 2.12 is a 'valid' convolution. The calculations are only done for vertical and horizontal positions of the filter where the filter fully covers a part of the image. This results in a smaller width and height of the ouput.

The decoding needs the opposite of a 'valid' convolution to have a reconstruction that has the same size as x. 'Full' convolution also does the calculation outside the border of the input image / feature map. As long as at least one pixel / value / weight of the filter overlaps with one pixel / feature from the input. The filter also needs to be rearranged such that it does the opposite calculation of the encoding. $\tilde{W}_k$ is W_k when rotated by 180 degrees.

The activation function $\varphi()$ and biasses b_k and c correspond to Equation 2.9 as the auto-encoder is part of a neural network. In a large network one can stack these convolutional auto-encoders to create higher-order abstractions.

In the paper was tested what the effect is of alternating the auto-encoder layers with max-pooling layers or only stacking the auto-encoders. The networks with max-pooling layers have the following advantages:

- Translation-invariant features, because only one of the features in the input pool of the layer needs to be activated to have an output. Higher layers have to check only one feature instead of the whole pool to check if something is recognized.
- Down-sampling the number of features. Less features to deal with in later calculations and improved filter selectivity.

The result was convolutional filters and resulting features that were more generally applicable.

3

Research

In this chapter the research set-up is presented, starting with the research goals in section 3.1, followed by descriptions of the reference research (section 3.2) and the set-up for this research (section 3.3), which are then compared in section 3.4, and the testing methodology is presented in section 3.5.

3.1. Goals

The goal of this research is to test the Deep Q-learning algorithm (algorithm 1 on page 16) with a more difficult task combined with a higher world complexity. That is, to try to make a system that can handle higher dimensional visual inputs and more complex tasks, while maintaining a high level of adaptability. Such a learning system can be used in many robot applications. For example to learn real-life visual tasks without the need of an expert for constant feedback. If this ultimate goal is unobtainable, the goal of this research is to find what limitations the algorithm holds.

3.2. Reference

Starting point is the paper by Mnih et al. [12]. Their experiment showed a fully adaptive algorithm (subsection 2.3.3) using a high-dimensional input for the complex task of playing a computer game.

World

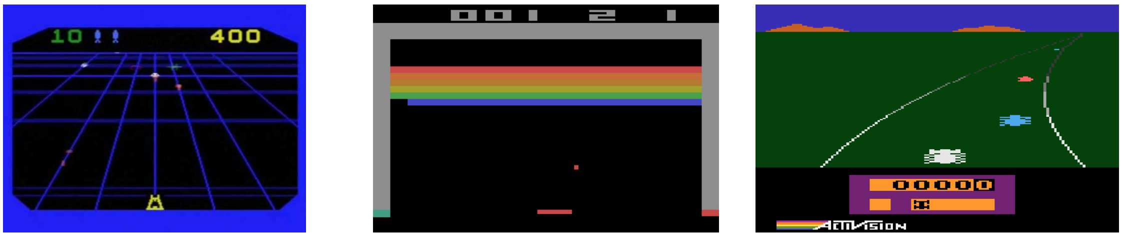
The learner's worlds according to the paper are the seven ATARI 2600 games named Beam Rider, Breakout, Enduro, Pong, Q*bert, Seaquest and Space Invaders (Figure 3.1. These games are emulated with the Arcade Learning Environment [2] such that these games can be played by a computer program.

Task

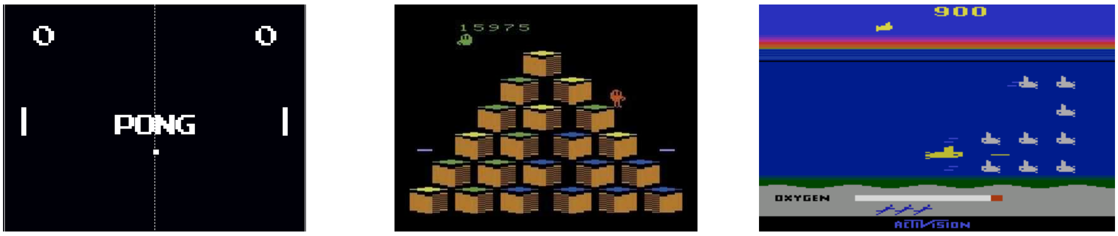
The goal of the learner is to play each ATARI game and achieve the highest score possible. Rewards are given for changes in game score: 1 point for any increase and -1 for any decrease.

Sense

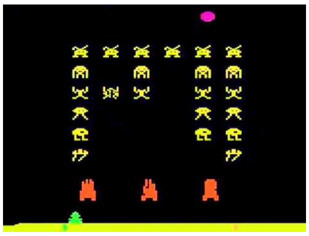
The images shown in Figure 3.1 is not what is used as input for Deep Q-learning. The images, which are originally 210×160 pixels with 128 colors, are sub-sampled such that they only contain 84×84 pixels with four colors. In groups of four image frames, these images are processed and stacked to form one $84 \times 84 \times 4$ input for the learning algorithm.



(a) Beam Rider (b) Breakout (c) Enduro

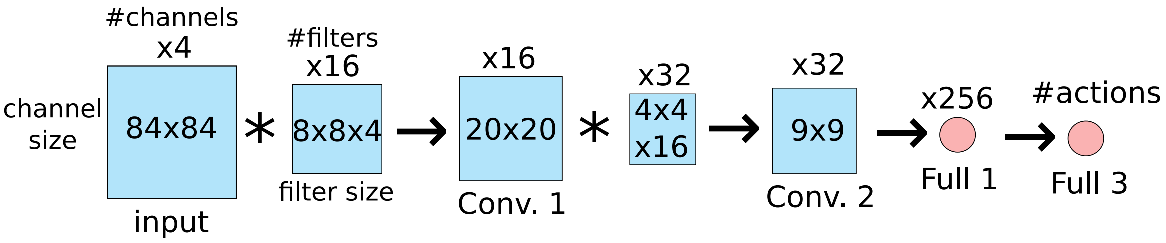


(d) Pong (e) Q*bert (f) Seaquest

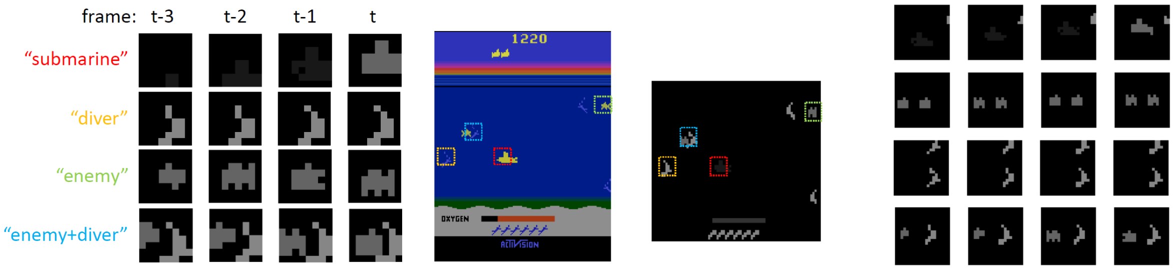


(g) Space Invaders

Figure 3.1: Screen captures of seven ATARI 2600 games



(a) Diagram of the layers in the action-value function. Conv. means Convolutional layer. Full means Fully connected layer



(b) Example of filters and neural activation of the first convolutional layer

(c) Example of filters of the second convolutional layer

Figure 3.2: Visualization of the action-value function. The example is from a different paper [4] that also learns with convolutional layers in the "Seaquest" game.

Think

The Deep Q-learning algorithm is described in detail in the paper [12] and is summarized in subsection 2.3.3. A schematic description of the neural network architecture is shown in Figure 3.2a. The network has two convolutional layers and two fully connected layers. Each output neuron represents the action-value of one of the possible actions in a game. With both convolutions in Figure 3.2a the convolution output is sub-sampled (using stride) to reduce the amount of features and therefore also reduce the amount of calculations. The first convolution has a stride of 4 in both directions, which means 3 in 4 locations on any axis to apply a filter are skipped. The second layer uses a stride of 2.

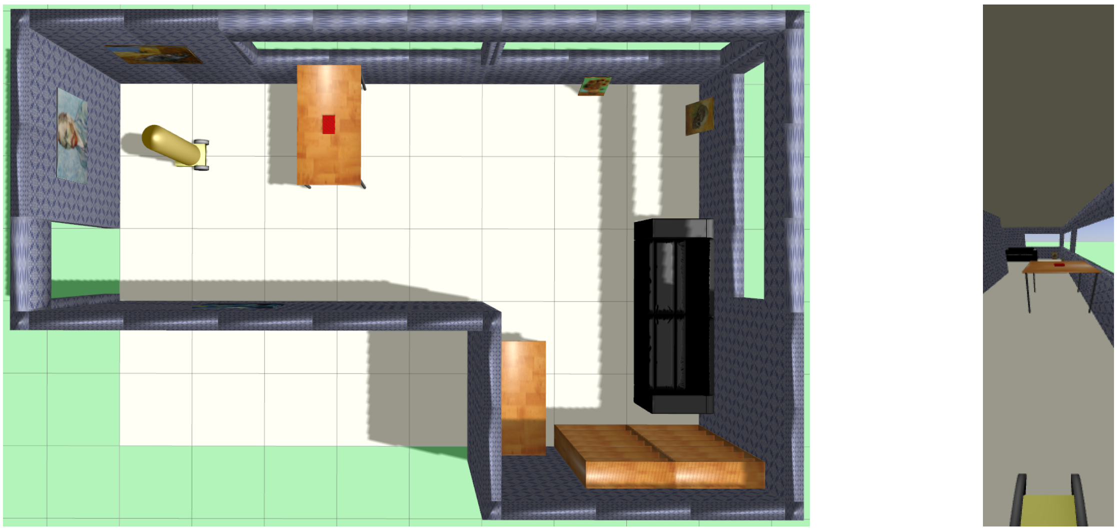
Figure 3.2b and 3.2c show examples of visual features and neural activations from a different research (Guo, X. et al. [4]) where the a similar network was used on "Seaquest". The left part of Figure 3.2b shows 4 features. Which show from left to right a feature label and then the 4 channels. The middle part shows where the features are recognized in a game image. The right part shows where the features are recognized in the input image. In Figure 3.2c a similar feature visualization corresponding to the second convolution is shown.

Act

The action-space differs between the games. Movement actions are enough to play, for example, Breakout (Left and Right) and Pong (Up and Down). In other games more actions are available, like launching torpedoes in Seaquest, next to the 8 movement directions (up, down, left, right, up-left, up-right, down-left, down-right).

3.3. Set-up

In this section the research set-up for this thesis is presented in the same order as the reference research was described in previous section.



(a) Top view of the simulated room with the ceiling removed.

(b) Visual input of robot

Figure 3.3: Simulated environment

World

The world of the learner is a 3D simulation made in Gazebo [8]. The simulated environment resembles a simplified living room which is shown in Figure 3.3. It looks similar to a real-world test environment one can find at universities. In Figure 3.3a the top view is presented: showing the robot in yellow in the top-left, the object to pick up is a red book on the table, the drop location is the bookshelf in the bottom-right.

In Figure 3.3b the room is seen from the robot eye. The room has an L-shape to have parts of the room out of view. All objects have different colors to better discriminate between them: books are red, passable ground is white, non-passable ground is green, walls have a blue pattern, solid objects have a wood pattern or a black surface. Paintings are placed in relatively empty locations for better navigation: every location must generate a different visual input for the robot.

The robot is able to sense the room only through its camera and is able to act on that by moving through the room. More detailed descriptions are given after the task description.

Task

The goal (the state where the episode ends in success) is upgraded in steps. First the goal is to find and pick-up a book. If that is a success then the goal is upgraded to pick-up the book and then bring it to the bookshelf. And finally the goal can become to find and bring all the books to the bookshelf.

For reaching the goal, the learner gets the highest reward ($r_{fin} = 1000$). For every collision it receives negative reward ($r = -10$). The same punishment is given for when the robot goes through the wall. To enforce efficiency, a negative reward of $r = -1$ is given with each action. Then to lead the learner in the right direction:

- When the object can be seen but is out of grasping range: a small reward is given for moving closer ($r_{closer} = 5$) and a smaller negative reward is given for moving further ($r_{further} = -1$). If the robot is not facing straight at the object, the reward will be closer to zero.

- When the object can be seen and is inside grasping range: The same positive reward is given for turning towards the object and the same negative reward is given for turning away from the object.

The smaller rewards also serve to teach the robot to explore until it sees the pick-up object. These "guiding" rewards can be switched off when the action-value function is sufficiently explored.

Sense

The visual input is set to several times larger than mentioned in the previous section: 512 by 128 pixels with RGB colors. This is the easiest way to increase the task difficulty, because more pixels means more information to process. Details will also be more visible with higher resolutions which allows for objects to be better observable at larger distances. There is also no sensor noise, which could have been another added difficulty.

The long side of the image is the vertical dimension and there is a wider camera angle to keep the action space small. In this case the robot can see its base/wheels but also the highest bookshelf when standing close to it. The robot does not have to tilt its head which reduces the amount of actions needed.

The use of colors is chosen to make it easier to recognize objects. Using the RGB colors triples the input size. The RGB colorspace was used because the Gazebo simulator delivered the the images in that way. In the simulation, objects and surfaces of relevance can be given different color groups to, for example, make it easier distinguish between floor and wall.

Think: architecture

The baseline architecture for the learner is one similar to the architecture presented by Mnih et al. [12]. Q-learning is applied using their Deep Q-learning algorithm (algorithm 1). Their neural network is adjusted to handle the new input and output as described in this section:

1. Input layer of $128 \times 512 \times 3$.
2. Convolutional layer with 32 filters of size $8 \times 8 \times 3$ and using a stride of 4, resulting in an output of size $31 \times 127 \times 32$.
3. Convolutional layer with 64 filters of size $4 \times 4 \times 32$ and using a stride of 2, resulting in an output of size $14 \times 62 \times 64$.
4. Convolutional layer with 128 filters of size $4 \times 4 \times 64$ and using a strid of 2, resulting in an output of size $6 \times 30 \times 128$.
5. Convolutional layer with 256 filters of size $6 \times 30 \times 128$, resulting in an output of 256 visual features.
6. Fully connected layer of 512 neurons.
7. Output layer of as many fully connected neurons as there are actions. These output neurons have a linear activation functions because action-values can be unpredictable and higher than the maximum output of a sigmoid-function (1.0).

These numbers can also be seen in the diagram of Figure 3.4. The network, when properly learned, should show features as were shown before in Figure 3.2 and Figure 2.3. The features corresponding to this network will be shown in the next chapter.

The convolution filter sizes and the stride are the same as in [12]. The amount of filters per layer used are doubled. The amount used for the Atari 2600 games was not sufficient in this experiment and therefore increased until learning was possible. The relative number of filters

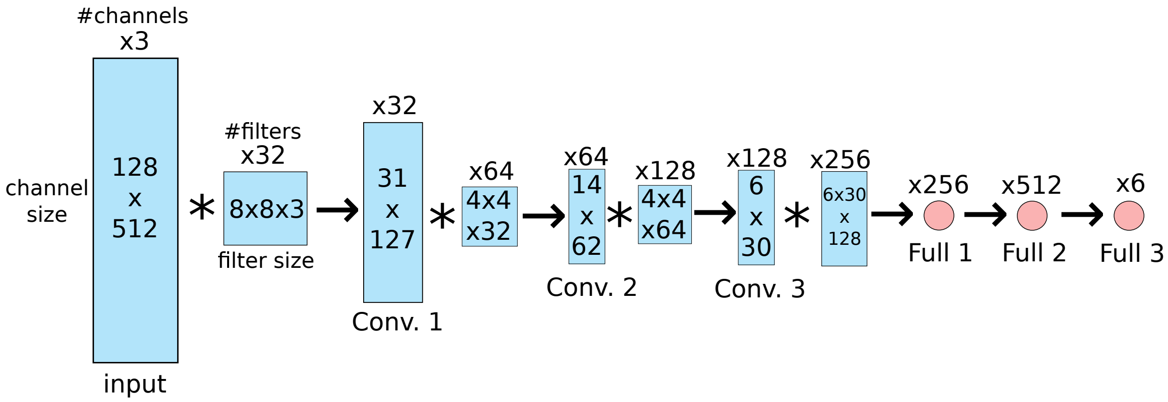


Figure 3.4: Diagram of the layers in the action-value function. Conv. means Convolutional layer. Full means Fully connected layer

between layers is again the same as in [12], multiplying by 2 for every next layer. These numbers will be tuned in the first experiments as described in section 3.5.

The fourth convolutional layer is actually the same as a fully connected layer. Here each filter is only used in one neuron. This means there are three fully connected layers, which is one more than in [12]. These fully connected layers have features that represent information that is contained in the full input image, compared to the convolutional layers which have features that only represent a part of the image. The fully connected layers are expected to form the logic operations, to make decisions based on the abstract features from the highest convolutional layer. In this context, an extra layer should allow for more complex logic and therefore allow for more complex tasks.

Think: learning

Learning is done in four phases, each running multiple episodes. One episode has a maximum of 500 actions. The episode ends then or when the goal is reached. With every new episode the robot and books are reset to random locations in the room.

1. Pre-learning phase: 10 episodes with only random actions are stored into the replay memory. Then the auto-encoders converge with batches of 128 random experiences. First the lowest Convolutional Auto-Encoder (CAE) layer converges. Then the next CAE layers are added with the batch updates in between each addition. After this phase the CAE layers are converted to reinforcement learning convolutional layers. This is the same as in the research of [11].

2. Exploration phase: In about 20 episodes a large part of the state-space is explored due to random starting locations and random action afterwards. This is where the action-value function should converge.

3. Mixed phase: The randomness of actions decreases from 100% to 0% over 20 episodes. A non-random action is the best-valued action for the current state. This is also known as an epsilon-greedy policy.

4. Exploitation phase: Another 20 episodes of learning while always taking the best-valued action. This phase is only about fine-tuning. The robot should show behavior that is close to the goal behavior.

After learning the value function the system is evaluated by running a minimum of 20 episodes without learning.

Act

The action space is kept as small as possible. These actions are necessary for this environment:

- Robot motion actions, to get to objects and to bring those objects to the right place.
    - Forward
    - Backward
    - Rotate left
    - Rotate right
- Robot-object interaction actions:
    - Pick up
    - Put down

All these actions have an instantaneous effect on the environment, therefore the robot knows no speed. In case of motion the robot is teleported to the new location and in case of robot-object interaction the object is teleported into the hand of the robot or back on the furniture. The motion in between is not important to the visual inputs of the higher order decision making in the AI, because nothing in the environment has a velocity. There is also no probability distribution of where the robot ends up.

The implementation of each action is straight forward: First, Steps have a size of 10cm in the direction of the robots orientation (or the opposite for the backward action). Then rotations are about 11 degrees about the Z-axis (the calculations need this angle to be defined as halfturn, which is 0.1 for robot rotations). Third, the pick-up and put-down actions do not move the robot. These actions teleport the object from or to the relative holding position which is in view of the robot's camera. With pick-up the action chooses the most visually central object when there is more than one object in view and range. With put-down the action generates a location for the object. For example, a book will be put in the first available slot on the bookshelf.

There are secondary actions which activate in special cases. When the robot collides with anything and/or leaves the room the state of the simulation is reverted to undo the last two actions. Reverting by one instance was not enough, because the collision signal has a delay. Reinforcement learning should be able to deal with delayed punishment and reverting back by two instances is sufficient.

The other secondary action is the simulation reset which activates when the episode reaches the maximum amount of actions, when the goal is reached or when a critical failure is made by the robot. A critical failure would occur when the robot drops an object before it reaches the object's destination. This prevents the possibility of the robot learning to repeatedly pick-up and put-down for short-term rewards. A reset places the robot in a random position and orientation on the available ground. Also the holdable objects are placed in random positions to avoid the robot learning to go to features in the surroundings.

Let's first take the primary task, to get the book, to check if the problem is still Markov. Does an input image have enough information to measure which task is most valuable? Figure 3.3b shows the ground in the lower half with the robot wheels in the lowest part. If any feature, but for the ground or a book, shows right in front of the wheels, the value of the 'forward' action should be low and vice versa. If one of the red books is slightly more to the side (left or right) the value of the corresponding rotation should be high. If the book is low in the image and straight ahead, the pick-up action is probably best.

Therefore with enough visual features that represent the relative pose of books and obstacles, each best action can be deduced with simple logic. Even when the robot is holding a book (always showing in the same location in the view, relatively close to the camera) one visual feature can be enough and can be used to choose to go to the drop location, instead of another book.

3.4. Differences and Consequences

This section describes the differences between the reference research set-up and the research set-up of this thesis. Each difference has consequences for the learning algorithm which are also described with each part.

World complexity

World complexity is the complexity of what is detectable in the environment. This includes the size and dimensions (2D or 3D) of the environment and, second, the complexity and number of different objects, such as tables, sofas, paintings, plants, etc.

The size of the environment does differ. But is not easily quantifiable, because the ATARI game worlds and the Gazebo simulation are all different. The ATARI games Beam Rider and Enduro have relatively large and long-stretched worlds compared to the compact worlds of the other five ATARI games used in for Deep Q-learning and the Gazebo simulation. What can be said, is that the Gazebo simulation has more small details than the ATARI games which makes the size of the world information larger. The amount of details is also a measure for the complexity of objects.

The consequence of more details in the world is that more visual features are needed to describe them all. This means that more neurons are needed in the image processing layers (the convolutional layers) of the neural network which is accounted for by doubling the number of neurons per layer in the network architecture.

The dimensions of the environment are 2D for (most of) the ATARI games and 3D for the Gazebo simulation. This greatly influences the complexity of the world, because in the ATARI games one or more world axes of the world frame overlap with or are on the same plane as the axes of the camera/image frame. In other words, the action "right" displaces the controlled visual item in the same direction. In the 3D Gazebo simulation this is different. The action "right" results in all visual features displace to the left and the action "forward" results in all visual features displacing to the edges of the image.

Another difference between 2D and 3D worlds is that 3D objects have a relative pose that can be described with seven variables (a 3D location and a quaternion for 3D orientation), while 2D objects have a relative pose that can be described with only three variables (a 2D location and one angle).

The consequence of this misalignment of world and image axes and pose descriptions is that more visual features are needed to describe the same object. More features are needed, because objects have a different pattern on the image for different distances and orientations. For example, in Space Invaders, each object has only one orientation which requires only one filter to be learned in a convolutional layer of the network. In the Gazebo simulation one object could need ten filters to be properly recognized. Therefore also for this change more neurons are needed which is also accounted for by doubling the number of neurons per layer in the network architecture. The amount of filters needed will be determined in the experiments.

The number of different objects of relevance is low in both the reference research and this thesis. There are about six important objects in the Gazebo simulation and the ATARI games vary between two, for Pong (Figure 3.1d), and nine important objects, for Space Invaders (Figure 3.1g).

Task complexity

The task complexity is defined as how much effort (or calculations) the learner has to put into completing the task successfully. This is influenced by the input size, the amount of actions possible and the amount of actions needed for success. The next three parts of this section compares task complexity in the context of the reward function, in the context of the input size and in the context of the action space.

Reward function

The main increase in task complexity is an increase in the amount of actions needed for success. In the reference research the game score is increased regularly which is shorly-delayed feedback. For example, in Seaquest (Figure 3.1f) a torpedo hits a boat within several actions. Let's assume this is within three seconds which is 45 actions. The smallest amount of actions needed in the Gazebo simulation is a little less or about the same as that number. To cross the room the virtual robot needs about 80 steps of $10cm$ in a straight line. The closest goal object is always closer than that.

Increasing sparsity of rewards means there are more iterations needed to trace back through all the experiences. It also results in greater effects of "hills" in the action-value function which have to be overcome. An example "hill" is zero or negative rewards on the road to get to the goal. Also zero reward, because the learner initially does not know how to get through the space of no feedback. With enough exploration through random actions this can be overcome.

The discount factor (γ from Equation 2.8) is set to 0.9 for the set-up in this thesis. Which results in the last 50 steps towards the goal to have a higher value than the guiding rewards due to the final reward ($r_{fin} \cdot \gamma^{steps} = r_{closer}$). In 50 steps the robot can move $5m$ (steps of $10cm$). That is more than half of the length of the room and should be enough to get to the closest object to pick up, which should result in a robot taking the fastest route to the goal.

Input image resolution and colorspace

The input size is increased. In the ATARI game experiments the learner received $84 \times 84 \times 4 = 28224$ inputs, while with the new experiments the learner receives $512 \times 128 \times 3 = 196608$ inputs. That is an increase of a factor of about 7.

A larger input space means there is more to be filtered and more to be compressed into relevant abstractions. Therefore also seven times more neurons are required in the network. This is done by adjusting the dimensions of the convolutional layers proportionally to the change in input size.

The third dimension of the input is also different. In the ATARI game experiments the third input dimension represents time. Each of the four positions is a different frame. This allows for detection of movements between frames. In the Gazebo simulation, objects do not move without being caused by actions of the simulated robot. Therefore it doesn't need the different frames in the same input. Also, as explained the last part of section 3.3, each camera image should have enough information to make an action choice. Therefore, there is also no need to have previous frames simultaneously in one input.

The third dimension in the new experiment's learner inputs, is the colorspace. The colors in the ATARI game images were converted to four values of gray. This was not done with the images from the Gazebo simulation. The decrease in the number of positions in the third input dimension is already accounted for with the change in dimensions of the convolutional layers, as mentioned above.

Action space
A larger action space means more options to explore which require more calculations and more experiences to learn from. The action space is kept relatively small. In the Gazebo simulation there are six possible actions and according to Mnih et al. the ATARI games have between four and 18 possible actions.

As a result the new experiments should take less time and need less classification neurons (fully connected neurons, higher layers) in the context of action space, compared to, for example, Seaquest. Seaquest has 18 possible actions: 8 directions or no movement (= 9) in combination with pressing a button (×2).

Neural network
In the action-value function, next to the increase in neurons, also the number of layers is changed to handle the larger images.

A larger input needs more steps of compression to be of small enough size for any fully connected layer. This is accounted for with one extra convolutional layer. Also, higher task complexity needs higher level abstractions. This accounted for by one extra fully connected layer.

The increase in the number of layers, however, also increase the difficulty of the control problem. More layers means more steps, of transforming the information, to be learned. Above certain levels this can become so complex that it cannot be learned.

Learning types
Also the layer types and learning types may need change. Changing the task has influence on how the action-value function updates and converges. Further delayed rewards result in convolutional layers that take longer to find a consistent set of useful features. Therefore more learning time is needed while time is something to be minimized due to hardware. The project has to run on a high-end student laptop (CPU: Intel i7-4700MQ@2.40GHz×8, RAM: 16GB DDR3).

Deep Q-learning can have problems with an increased input size and complexity. More inputs means more information to consider which requires more neurons. More complex input means more levels of abstraction are required, therefore even more layers of neurons are needed. This strengthens the dimensionality problem, because there are more variables to be learned.

In this experiment the first hidden layers will be replaced by unsupervised learning layers to reduce learning time and to reduce the effect of the dimensionality problem. Because, unsupervised learning layers converge on their own and possibly faster than the reinforcement learning layers. When they converge faster these layers also give an early and more stable input for the reinforcement learning layers that are stacked on top, which in turn should result in faster convergence of the higher layers. Unsupervised learning helps a little with the dimensionality problem, because it takes part of the optimization space of the reinforcement learning algorithm.

Although unsupervised learning is a different learning method, it can work with the same configurations of neurons. Only the way the error of the neural activations is calculated is different. It is not expected to result in much different visual features, because in both learning types the lower layers need to extract edges and/or colors before higher order abstractions can be constructed.

An extra measure to reduce learning time is to switch off learning in the unsupervised learning layers as soon as they are converged. This also gives a more stable input for the reinforcement learning layers. To determine when to stop, there will be first some experiments without a switch. From these preliminary results can be determined when the maximum or average weight update is low enough to switch off learning.

The choice of layer type for the unsupervised learning layers is limited by the layer type stacked on top of the last unsupervised learning layer. In this case that is a convolutional layer which requires an input where each input has a relative distance to other inputs like pixels in an image: an image of activation signals. Therefore the unsupervised learning layers need to be of types like convolutional layers or pooling layers.

The Convolutional Auto-Encoder layers, described in [11] and summarized in subsection 2.3.4, make a convenient choice for layer type and learning type, because the same layer parameters (number of neurons and size of filters) can be used to test both a full-RL action-value function and a partial-RL action value function. This also results in a more valid comparison of the two implementations.

3.5. Experiments

Different network configurations have to be evaluated first to obtain most optimal results: How many features should there be in each layer? The search for these numbers should be done with the tests that give the most results. That means including unsupervised learning in the pre-RL phase, continue learning these features with RL, learning with a more helping reward function (including the mini-rewards) and to more features than needed in higher layers to start with.

First, to find the correct amount of features per layer, two variables are experimented with: the number of first-layer features and the increment rule for higher layers. With features are meant: filters for convolution layers and neurons for fully connected layers. An increment rule can for example be as used in the Atari experiments [12] a power of 2 rule: multiplying the amount of features in the previous layer with 2 to get the amount for the current layer as can be seen in Figure 3.2a and 3.4. This rule can for example also be linear or constant, but the power of 2 rule worked in the Atari experiment and gives most likely more than enough features in higher layers when the number of first-layer features is found.

Finding the number of first-layer features is done by retesting and increasing the number until a successful result is found. Starting with 8 this number will be multiplied by 2 with every retest. Increasing the amount of features with each retest also tests for the peaking phenomenon [7], which is the relation between the amount of features, the number of variables in the approximation function and the classification error. This effect can be seen when visual features in, for example, the unsupervised layers are in greater numbers and are being overtrained.

4

Results

The sections in this chapter describe the execution and results of each of the tasks mentioned in section 3.5.

Figures like Figure 4.1 show the convergence of the different layers of the action-value function. The calculation of the graphical data is a little more complex than can be explained in one sentence. Therefore it is described by equations 4.1 - 4.3. These equations are applied separately for every layer. W_t is the sum of all weight changes after cycle t (single minibatch). W is the vector of all W_t, of which C is the maximum. Finally in Equation 4.3 the elements of vector W are scaled to a fraction of C, after which the new vector is locally averaged to create the graph data D. Local averaging is done according to a smoothing operation which is explained in equations 4.4 to 4.8. The second argument of `smooth()` shows 101, which means the local average is taken from the range $[t-50, t+50]$. There are a few consequences of these data transformations: First, the layers can not be compared with each other due to the scaling with their own maximum. Second, the graph data is skewed for the 50 first and last moments due to the smoothing operation, because it has less data to average with. It causes spikes towards the unaveraged values. One can still talk about convergence within the layer: when the layer was changing most, when the layer started and finished converging.

$$W_t = \sum \Delta w \tag{4.1}$$

$$C = \max(W) \tag{4.2}$$

$$D = \texttt{smooth}(W./C, 101) \tag{4.3}$$

$$
\begin{aligned}
D(1) &= \frac{W(1)}{C} && (4.4)\\
D(2) &= \sum_{i=1}^{3} \frac{W(i)}{3 \cdot C} && (4.5)\\
D(3) &= \sum_{i=1}^{5} \frac{W(i)}{5 \cdot C} && (4.6)\\
&\vdots && (4.7)\\
D(t) &= \sum_{i=t-50}^{t+50} \frac{W(i)}{101 \cdot C} && (4.8)\\
& && (4.9)
\end{aligned}
$$

4.1. Full-RL action-value function without pre-learning with CAE

These results show what the program learns when the action-value function has no pre-learning with convolutional auto-encoders. Figure 4.1 shows the convergence of different neural layers. The output layer (layer6) is being updated continuously. None of the convolutional layers update with large steps before the simulation reaches action 15k. During the mixed phase and the exploitation phase the lower layers change and the output layer can finally converge a little. Figure 4.2 shows that in the end the program only learns to do the pick-up action which is also the only action that consistently results in no punishment, but unfortunately also in no reward when not in front of a book.

4.2. Getting objects

4.2.1. Getting objects without learning

The results shown in Figure 4.3 provide a baseline to compare with. In this test the action-value function is never updated. Of the 81 episodes 12 were a success through random actions in the exploration- and mixed phase (black spikes of 100 reward in Figure 4.3a and Figure 4.3b). With more exploitation of the action-value function the step-forward action is chosen continuously as happened to be the result of random initialization (Figure 4.3d). That is also the cause of the increasing occurrence of negative rewards when action randomness changes to zero (Figure 4.3b).

4.2.2. Getting objects with bad CAEs

The result shown in Figure 4.4 is an example of what happens when updating of the action-value function is turned on, but the convolutional auto-encoders diverge. The exploration phase is similar to Figure 4.3a and is therefore not shown. The difference is in the mixed- and exploitation phase where can be seen what is learned. The results show it successfully avoids collision (no negative rewards in the exploitation phase, Figure 4.4b) by continuously choosing the pick-up action (Figure 4.4c), which does nothing when there is no object to grasp.

The learner however does not take action to go to one of the books and get a larger reward. This is because the robot can't 'see'. The convolutional autoencoders diverge due to the use of the wrong loss function and therefore the input does not get to the fully connected layers. With use of the biases the network is able to find the best action on average which is then chosen 100% of the time.

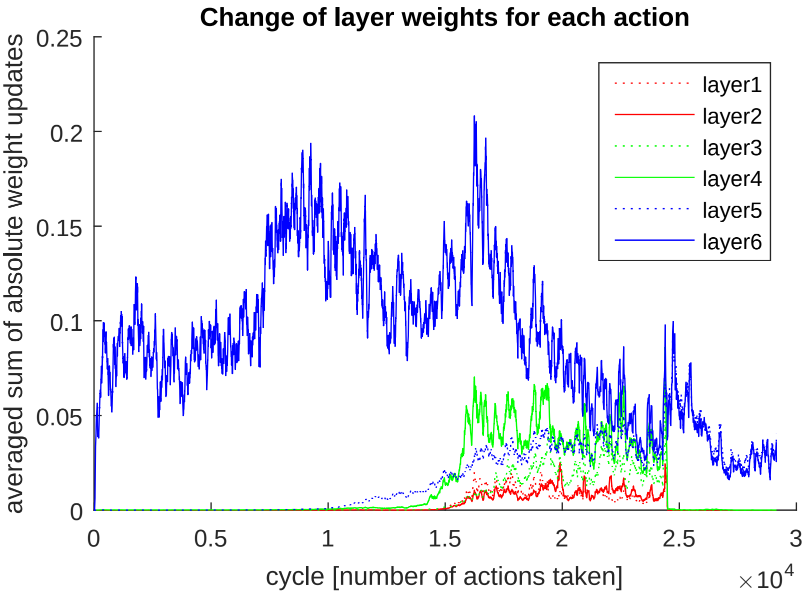


Figure 4.1: The locally averaged (100 cycles) ratio between the sum of absolute neural weight updates and the maximum update of that layer over the whole test. The first 10k cycles are the exploration phase. The second 10k cycles are the mixed phase. The rest is the exploitation phase.

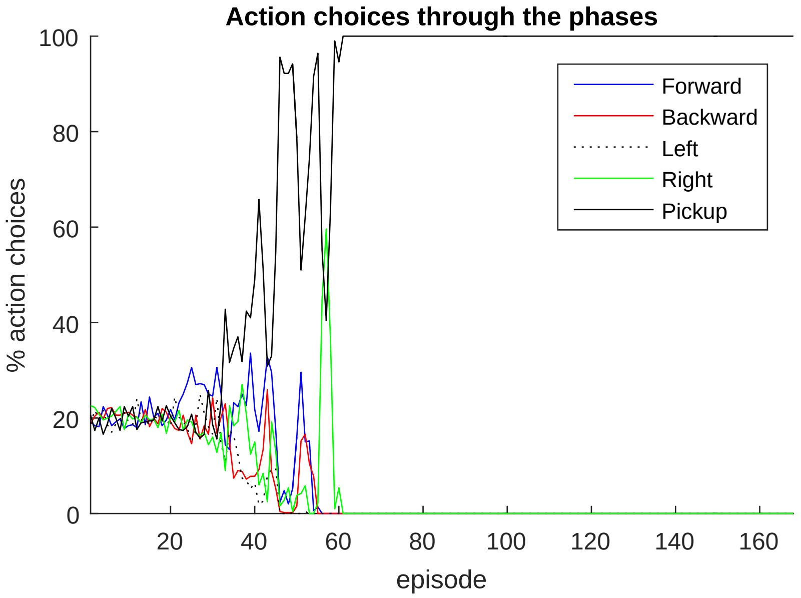
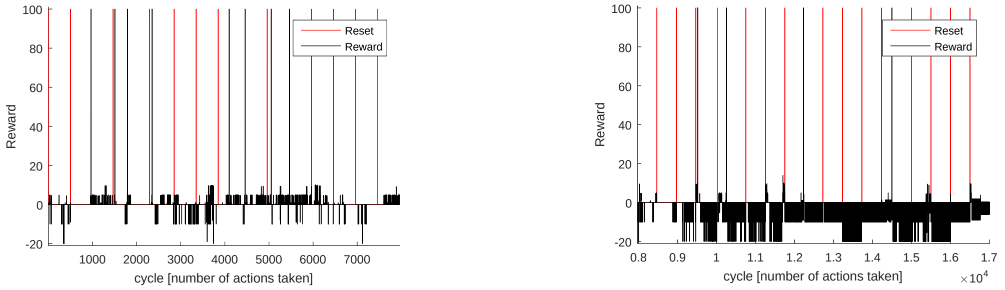
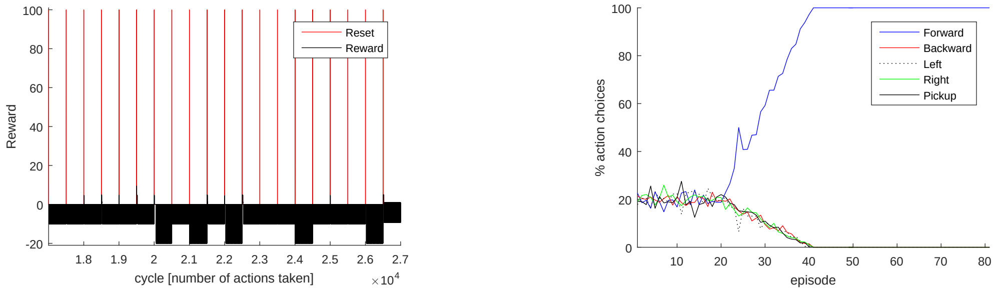


Figure 4.2: The percentage of action choices for each action per episode. The first 20 episodes are the exploration, the second 20 episodes are the mixed phase, the third 20 episodes are the exploitation phase and the rest is the evaluation phase.



(a) Reward during exploration phase

(b) Reward during mixed phase



(c) Reward during exploitation phase

(d) Action percentages per episode

Figure 4.3: Example results of the learner when the action-value function never updates. Subfigures a, b and c show the reward for each phase where the episode ends with every success (black spike) or a reset after 500 actions (red spike). Subfigure d shows the percentage of action choices for each action per episode. The first 20 episodes are the exploration, the second 20 episodes are the mixed phase, the third 20 episodes are the exploitation phase and the rest is the evaluation phase.

The reward function in this test does not have the small negative rewards for going further away from the goal (described in section 3.3). The next test will include this.

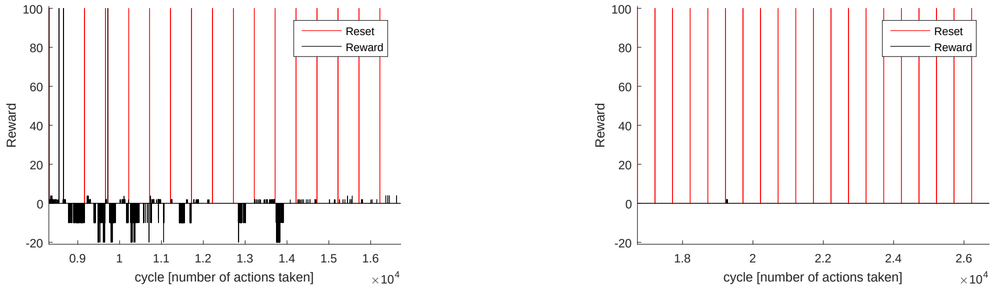
The autoencoder layers are continuously updated through the test. In this test is found that these converge in 3 cycles and that can therefore not be seen in Figure 4.5 (layer 1, 2 and 3). When comparing Figure 4.1 with Figure 4.5, the higher layers are now also converging properly before the exploration phase is finished.

4.2.3. Getting objects with good CAEs

In this test the internal loss function of the convolutional autoencoders is as in Equation 2.14. The input image now reaches the higher layers for better decision making. The network has 4 times less features as described in section 3.3 to speed up the tests and the pre-learning phase is activated.

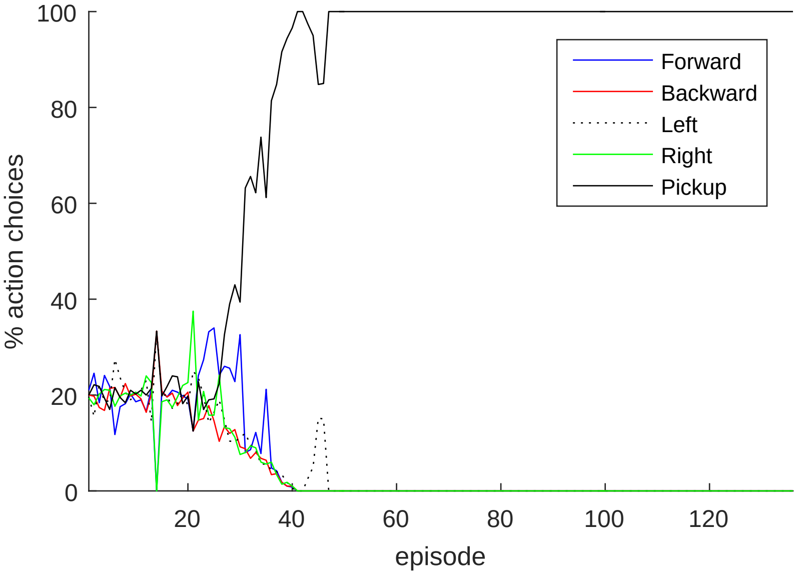
Figure 4.6 shows the action percentages for each episode. Episode 1-5 are pre-learning, episode 6-20 is exploration, episode 21-40 are of the mixed phase, episode 41-60 are the exploitation phase and from episode 61 and on is the evaluation. The figure shows that during fine-tuning (exploitation phase) the forward and rotation actions are mostly used. Which are the 3 actions that are most rewarding when not going for the books.

Figure 4.7 shows the movements of the simulated robot in the room. The red dots are the books. Yellow boundaries are the furniture (tables, sofas, bookshelfs). The robot movements are shown in blue for which the orientation is a green line for each instance. The episode depicted here is one of the final exploitation episodes: Forward is used to go around the room and the



(a) Reward during mixed phase

(b) Reward during exploitation phase



(c) Action percentages per episode

Figure 4.4: Example results of the learner when the action-value function does updates. Subfigures a and b show the reward for two phases where the episode ends with every success (black spike) or a reset after 500 actions (red spike). Subfigure c shows the percentage of action choices for each action per episode. The first 20 episodes are the exploration, the second 20 episodes are the mixed phase, the third 20 episodes are the exploitation phase and the rest is the evaluation phase.

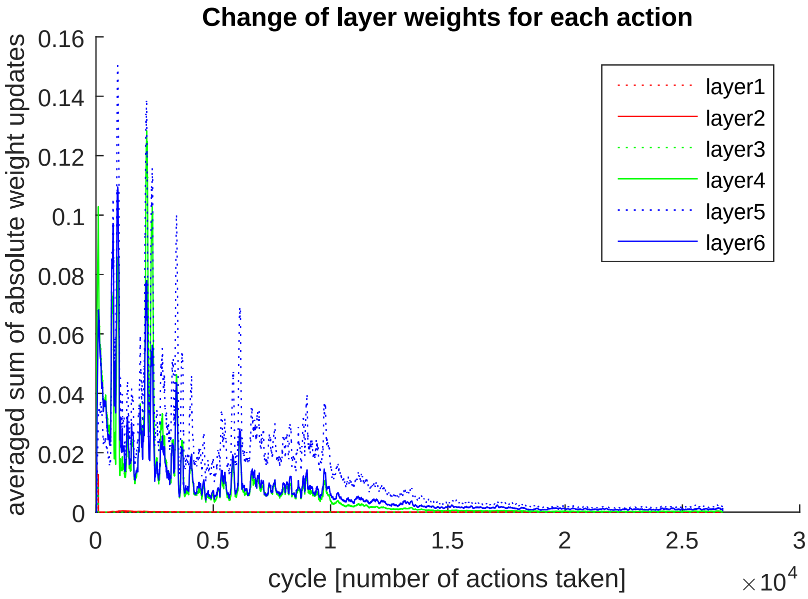


Figure 4.5: The locally averaged (100 cycles) ratio between the sum of absolute neural weight updates and the maximum update of that layer over the whole test. The first 20k cycles are the exploration phase. The second 20k cycles are the mixed phase. The rest is the exploitation phase.

rotations are used to avoid collisions. If there was a collision the orientation line would be black in staid of green.

These results show the robot learns to handle the basic movements, but does still not recognize the books as a target. More features, especially in the convolutional layers, should allow for more information to be passed on to the final decision layers. The next test has double the amount of features.

4.3. The Number of First-Layer Features

Increasing the number of first-layer features (section 3.5) is possible up to 64 with the available time and hardware. 8 features takes less than half a day, 64 features takes a little less than 7 days to complete. The results are shown in Figure 4.8. Episodes 1-10 are the pre-learning phase, then there are 3 episodes in which only auto-encoder iterations are executed, one episode for each auto-encoder and 64 iterations per episode with a batch size of 128 inputs. Episodes 14-33 are the exploration phase with randomness set to 100%. With every action a batch of 32 experiences is taken for learning the action-value function. Episodes 34-53 are the mixed phase in which randomness drops gradually with every action down to 0%. The exploitation phase is episode 54-73 after which learning is switched off and the remaining episodes are the evaluation of the learned behavior.

All of the tests stop having success episodes (ending in picking up a book) as soon as the action randomness goes near 0%. During 100% random actions about 30% of the episodes end in success. When learning is switched off, or earlier in the 16 features case, the behavior only chooses one action everytime.

The convergence images (right column in Figure 4.8 show that the bulk of the weight updates are done in a shorter amount of iterations when there are more features, especially for the four lower layers. The two highest layers need more iterations to converge, but also for these it takes less iterations to do the largest updates when there are more features.

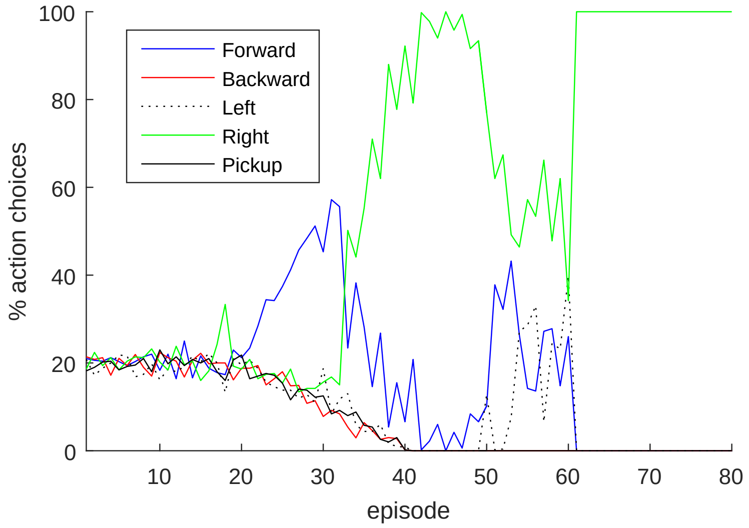


Figure 4.6: Action percentages per episode

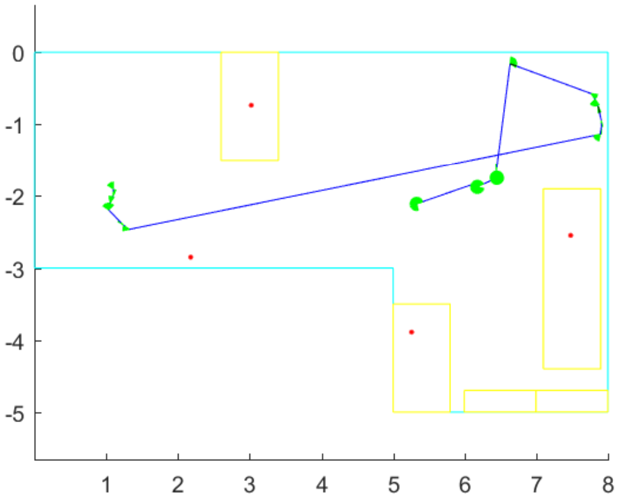


Figure 4.7: Schematic version of Figure 3.3a where the blue lines are steps, green lines show orientations

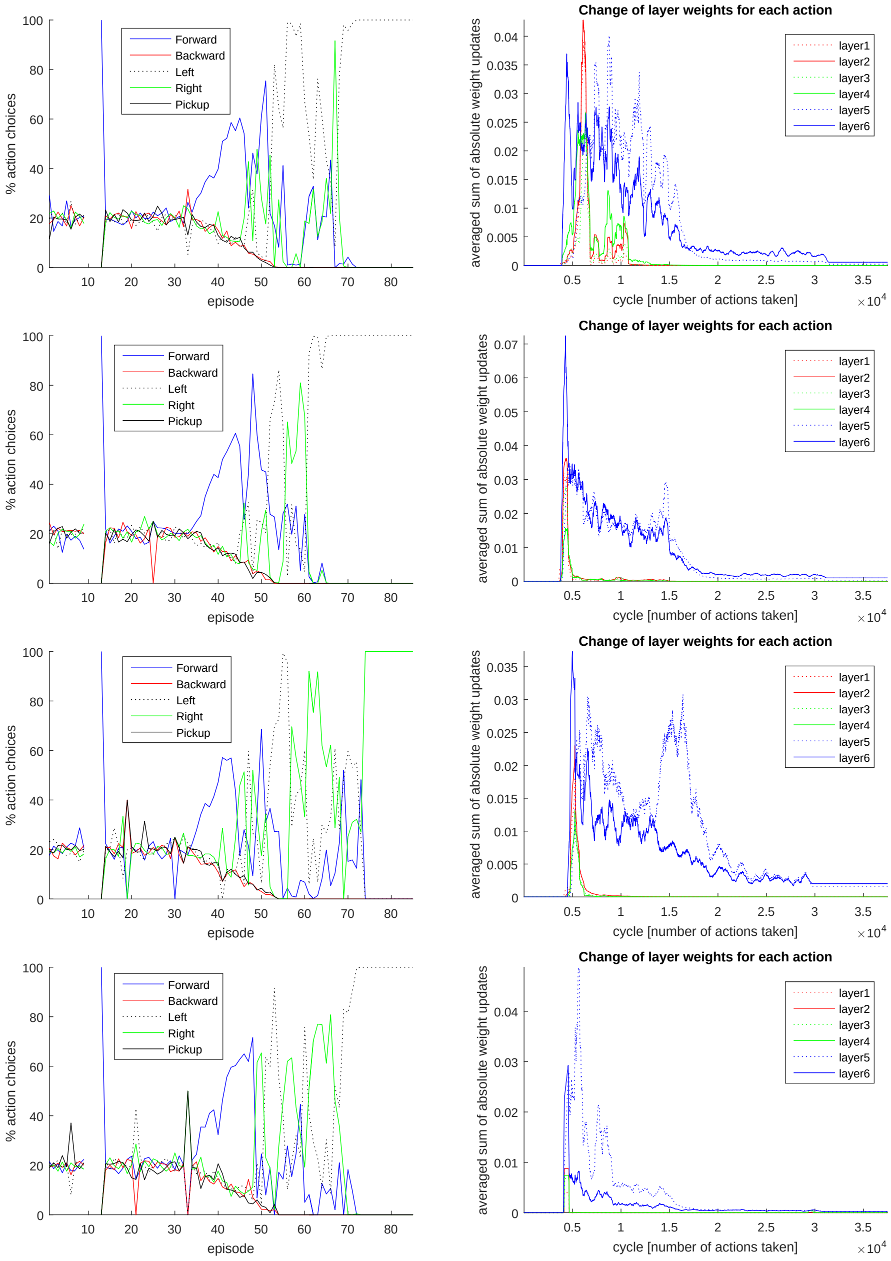


Figure 4.8: Results of increasing the number of first-layer features. Left column has the percentages of actions in each episode, right column has the convergence graphs. The rows from top to bottom represent 8, 16, 32 and 64 first-layer features.

4.4. Follow-up tests

In the previous section it was shown that the largest network could not learn a successful behavior. This section shows the follow-up tests that are worth mentioning. The other tests are variations on and combinations of network size, filter size, phase size, complexity of the reward function and image size. They all yielded the same results as found in Figure 4.8.

4.4.1. Longterm epsilon-greedy learning

The test that showed different behavior is an epsilon-greedy test with a constant epsilon of 10%. The network has the same size as in the test with 16 first-layer features and the replay memory is initialized to a size of 50k experiences. The replay memory is implemented as a circular buffer. The learner was stopped after 717 episodes (approximately 8 days). The resulting statistics are shown in Figure 4.9.

The first 10 episodes are the same as in previous results, followed by initializing the the convolution layers with weights learned with unsupervised learning. The episodes after that are of epsilon-greedy learning with reinforcement learning.

There are clearly two phases recognizable: before and after cycle 82295, which is in episode 187. Before this point the behavior is mostly random, nearly equal action percentages per episode. After this point the action-value function favors the pick-up action. The post-episode-187 behavior does not seem to follow a pattern, except for choosing single actions repeatedly. The success rate drops from 18% to 11%.

Cycle 82295 has a spike in weight updates of the lower 5 layers (Convergence graph of Figure 4.9). Before only the output layer shows large spikes, after only the two highest layers show spikes.

Figure 4.10 shows what happened with the network. The neural activations of the convolution layers are nearly equal for the first layer and 100% equal for the second layer. This is very different from what is learned by the auto-encoders (Figure 4.11). The neural activations clearly show that some features have converged to color and edge detectors. These features show white on a black background or vice versa. The quality of the reconstructions in Figure 4.11 is to be expected with these low amounts of filters.

4.4.2. Supervised Learning of Visual Features

To continue the search for how many features would be needed and to test if the proposed architecture would be able to recognize a book, a supervised learning test was executed. Trained with 5000 images, labeled by a boolean for whether it shows a book or not, and then tested with 500 images.

The labels were extracted by converting the image from RGB to HSV followed by a threshold for the red color of the books. If a book is visible, there should be at least 10 pixels of that color in the image. A simple and efficient operation that can be done by a smaller network. The neural network consist of 1 convolution layer with 64 filters (8×8 with stride 4 in both directions) followed by two fully connected layers of 200 neurons and 2 neurons (one for no book and one for recognizing a book).

With 40 episodes and a test after every episode the success rate stabilized at 69%. In every episode all 5000 samples are used once for updating. Increasing the amount of features again was blocked by hardware limits. The program uses more than 16GB RAM when loading the data and trying to initialize a network with 128 filters in the first layer.

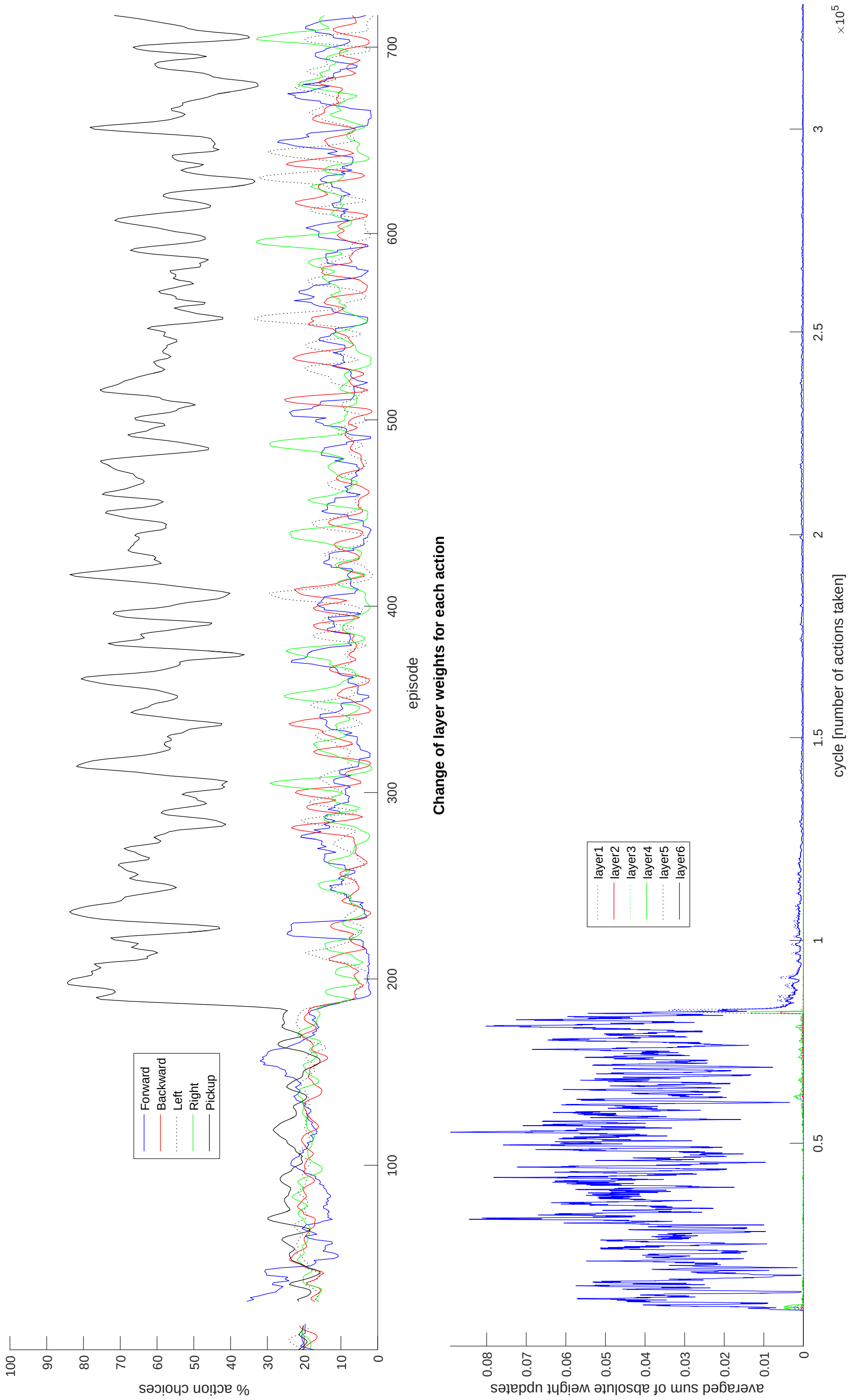


Figure 4.9: Results of continuously learning with an epsilon-greedy policy. The first graph shows the percentages of actions in each episode averaged locally over 10 episode, the other is the convergence graph.

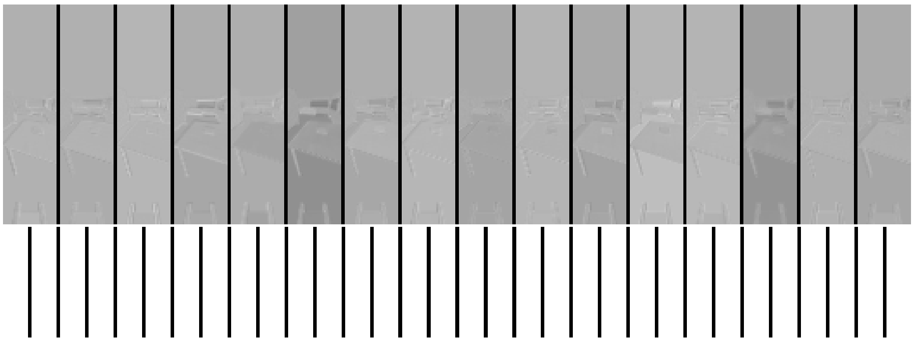


Figure 4.10: Outputs of the first and second convolution layer learned with reinforcement learning. From top to bottom: neural activations of the first layer and neural activations of the second layer. There is one image per convolution filter.

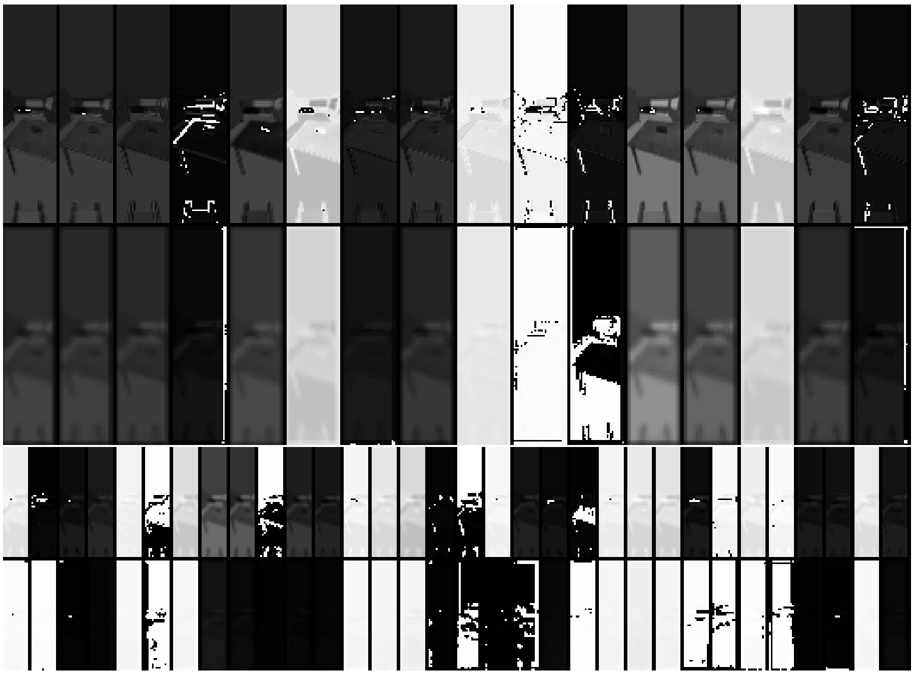


Figure 4.11: Inputs and reconstructions of the second and third convolution layer learned with auto-encoders. From top to bottom: neural activations of the first layer, reconstruction of this by the second layer, neural activations of the second layer, reconstruction of this by the third layer. There is one input and reconstruction per convolution filter of the preceding layer.

5

Discussion

Previous chapter showed that the goals of scaling up Q-learning with artificial neural networks (section 3.1) were not attainable with the proposed test environment and network architecture (section 3.3). In this chapter the reasons and mechanisms behind this learning behavior are discussed. This is discussed in order of how information flows through the learning system: simulation, experience, experience replay, learning, resulting behavior.

5.1. Scaling Up

The images produced by the simulation are of greater dimension and information density than what was produced in the Atari experiments of Mnih et al.[12]. The dimensions are raised from 84×84 with 4 possible values to $512 \times 128 \times 3$ with 256 possible values. If the input dimensions are the only difference, the results should have been similar. But, the simulation behind the images puts more and different information in the images that also needs to be extracted to properly complete the learning task.

With the Atari game 16 convolution filters in the first hidden layer are enough. With the 3D simulation 64 filters is not enough. Increasing the amount of initial filters even further is not possible with the computer used in this project, because the case with 64 filters needed 6.5 days to complete a full test of learning and evaluation.

How much features would be needed is hard to determine with the results, because the resulting convolution filters never converged to the desired features (this will be discussed further in the next section). But, 64 filters is a reasonable amount. There are about 6 different textures of significance in the simulation (book, wall, table, etc.). Then multiply that by about 10 filters per texture for differences in orientation and distance.

In the Atari game simulation all that needs to be extracted by the neural network are relative 2D locations and 2D orientations of several objects. These are recognizable with a handful of filters per object type. Actions have a resulting motion of objects in the same reference frame as the pixels in the image.

In the simulation of this experiment the information to be extracted is much greater in volume and complexity. Relative to the robot the objects have a 3D position and 3D orientation. This information can be extracted with object specific filters for different relative positions and orientations. Also some objects of significance, like the table or the sofa, are larger in the image than the size of one filter. On top of that, actions cause motion in a reference frame different from the pixels in the image.

Those 64 convolution filters are what is at least necessary to create all the relevant visual features. But, the first neural layers also create irrelevant visual features due to their learning type: unsupervised learning. Double the amount of filters might have done it, maybe even more are needed. All the extra filters would converge to the extra textures in the simulation: paintings, the outside (through the windows), the wheels of the robot and more. It is also possible that the unsupervised layer learns multiple features for the same situation, but then with a few pixels shifted.

5.2. Experiences and Experience Replay

Near the end of the learning phases of the tests the replay memory has about 30k experiences. This is 60 episodes of 500 steps or less if it is a success. According to the results about 30% of the random-action episodes end in success. But, not all experiences are acquired with a random action. Therefore about 8 successes is a normal amount if the learner does not learn the right behavior.

5.2.1. Sampling

Only 8 most relevant experiences in 30k is not enough to learn from. The chance of randomly taking one of these is too low and multiple uses are needed to get the action-value function approximate the right values.

There are primarily too many irrelevant features. Primarily, because after that success experience is learned well enough the experiences that precede become more relevant for Q-learning. Furthermore, even when that moment arrives the chances of randomly picking those preceding experiences is just as low.

In the Atari-game tests of [12] was this not a problem, because the rewards (game score increments) always come shortly after the actions that caused them. In this experiment's simulation the rewards sometimes come within 2 actions (collision) and sometimes come only after 100 actions. It is even possible the reward will never be given, e.g. when the episode reaches its maximum length of 500 actions. Therefore the long delayed rewards contribute to the low percentage of relevant experiences.

A more reasonable amount of relevant experiences would be around one per batch update. The random batches take 32 experiences. Unfortunately raising the chance from 8 in 30k to 1 in 32 is not possible in this case without compromising the randomness of the batch.

5.2.2. Learning

Learning from experiences of low relevance is a main problem in this experiment. These experiences have no reward and more important, these have initially no value. On average the low relevance experiences overpower the relevant experiences in learning.

The results show that with every iteration the randomly initialized weights become smaller until they produce a fully activated or fully deactivated output of a layer. This happens until a success experience is taken from the memory for replay. This causes a spike in learning updates, but these are then countered by another 1000 updates that diminish the weights.

Visualizing the weights and activations shows that the convolution filters do not show any structure when learned through reinforcement learning. Neither does the lowest fully connected layer. The neural outputs combine to all activated or all deactivated, blocking the information flow to the higher layers. The higher fully connected layers have therefore no useful input and so the biasses adjust to make the network produce the same output in every run: The most valuable action averaged over all experiences.

This comes from the Q-learning learning rule (Equation 2.8). The result-state value is calculated with the random weights, which produces a low value to start with. This state value is reduced with the discount factor to become the desired action-value in the origin state. This forces the weights to become smaller to produce smaller value approximations.

5.3. Resulting Behavior

In the evaluation phase of the tests the learned behavior always comes down to always taking the action with the highest expected value. Most often this is either the turn (left/right) or pick-up action. With more successes the pick-up action is chosen. With less successes the turn-action is chosen. Left or right doesn't make a difference, except for if one side has more collisions during turning. Stepping (forward/backward) is almost never chosen, because these actions result more often in collisions.

This happens because, as mentioned in subsection 5.2.2, the visual features for every input go to zero and therefore the action-values are based on bias. The result is an action with the least amount of punishment.

Contradicting the behavior above, at the end of the mixed phase and the first half of the exploitation phase the behavior shows to be much more intelligent. Collision avoiding and even some sort of exploration. Although, the most important actions, like the final pick-up, are one of the randomly chosen actions. In the second half of the exploitation phase the amount of different actions diminish. The 25,000th iteration (half-way the exploitation phase) has shown to be the boundary where the higher layers get no information from the input anymore.

This slightly intelligent behavior surfaces because the fully-connected layers are still partially able to learn. This is possible because the random structures recognized by the lower layers barely change or change slowly. What changes is the feature strength.

The exploration phase or random-action phase yields mostly all success of the test. Trying to combine the continuous action-randomness and exploitation in one long-term test did also result in bad weights and neural outputs.

5.4. Unsupervised Learning Layers

The unsupervised learning layers were initially added to reduce the state space for reinforcement learning, to create visual features of high abstraction that can be used in several different simulation objectives and to produce an initially more stable input for reinforcement learning.

The last of these three clearly comes back in the results and discussion. Reinforcement learning of the lower visual features includes them in the weight-reducing iterations (subsection 5.2.2). It would have resulted in a very low chance of generating a filter update to learn useful features and a very high chance of learning with random filter updates. Unsupervised learning creates at least features that are usable with the given visual input. These are learned from the start, with every iteration and provide a much more stable input for higher layers.

Reducing the state space for reinforcement learning could only be achieved through stacking the convolution layers and using the stride property to skip positions. With the about 128 features needed in the first auto-encoder (section 5.1) the state space is expanded after the first layer to 2.6 times the dimensionality of the input($128 \times 512 \times 3 = 196{,}608$ to $31 \times 127 \times 128 = 503{,}936$).

Unfortunately this experiment did not get to testing multiple simulation objectives to test the re-usability of the visual features created with the auto-encoders. The second object to put the book on the bookshelf would have increased the difficulty for the learner while it could not even do the first task.

5.5. Feeding Relevant Data

Now that it is clear that there is not enough relevant input data and that there are more irrelevant features to be learned than initially expected, there are some options considered but not executed due to some design rules.

For example on-policy learning instead of off-policy. Early experiences have a higher chance of multiple replays. Newer experiences have less influence on learning, while they could be valuable. Also this would help learning a rewarding experience multiple times with preceding experiences together. It would have a larger impact on the convergence than one rewarding experience at a random learning cycle. But, the disadvantages of on-policy learning will still make it worse. Starting from a random behavior, and when the action-randomness goes to zero, the learner will be stuck optimizing a non-optimal behavior.

Selective replay memory is also considered. Based on rewards given, experiences and preceding experiences could have been given a higher probability of replay. But this would also contradict off-policy learning. The experiences with higher rewards are highly correlated. As written in the paper about Neural Q-learning [12], this is something to avoid.

Another advantage of off-policy learning with equal replay chances is exploration behavior. If the robot is unable to see its target it should explore the room. These are experiences far before any final reward, but are also important to learning successful behavior.

5.5.1. Possible improvements

The RL learning algorithm will not be changed. Replay batches are kept full random with every experience having equal chance of replay. That is for reinforcement learning, not for the auto-encoders. It is possible for the auto-encoders to receive more rewarding data. This is a way to force the neural network to learn more relevant features without having to increase the amount of neurons.

There is one great advantage of feeding high-value inputs to the convolutional auto-encoders: the target is more often in view than out of view while the background can be different. Which should lead to more or better filters to describe that specific object. More important, it increases the percentage of relevant visual features.

To do this, one would need to make a formula to rate the relevance/usefulness of an experience depending on the corresponding reward and what is previously learned. Based on that relevance one can, for example, determine the chance of the experience to be picked for the random learning batch.

A disadvantage would be that there are other relevant features of the room that might not be seen by the auto-encoder. For example the auto-encoders will only learn about the table on which the target book lies, but not about the sofa on which the other book lies, because the random actions never got to getting the reward for that book. This can be can be countered by extending the pre-learning phase until most situations have a success.

There is also the question of how much data is enough and not too much. What amount of preceding experiences need to be included in auto-encoder learning, because some rewards have effect on state values many steps before the reward.

Approaching the problem from the other side is also possible. Instead of managing what is fed to the learning batch, an attempt could be made to increase the effectiveness of a success on the learning system. The method called eligibility traces could help with this (last paragraph of subsection 2.2.3). To accomplish this, one has to design an eligibility trace system that works with the replay memory system.

Such an eligibility trace system would have a large impact on the learners update frequency as it would need many more calculations, because the variables in a neural network do not always represent the same world state transition as is the case in a look-up table. Therefore, to calculate the updates the algorithm has to recalculate the value of next state ($\max_a Q(s_{t+1}, a)$ in Equation 2.8) with every step back in the trace. The length of the trace is therefore also the multiplier for how much time it will take to finish one learning cycle.

6

Conclusion

The goal of this research was to try and take a step towards the making of a robotic brain that can both handle large visual inputs and maintains a high level of adaptivity. The proposed challenge included a basic but visually complex 3D simulation in which images for learning are produced. These images are much larger than in preceding researches towards the learning of visual features.

The proposed learning method includes scaled-up versions of Neural Q-learning[12] and Stacked Convolutional Auto-Encoders[11]. Unfortunately it was not able to handle the complex visual data and delayed rewards of reinforcement learning. At least not within the time and hardware limits.

The main problems are caused by the increases in difficulty: long-delayed rewards and large visual inputs for the learner with high pixel information density. Due to the hardware limits it is not possible to determine which is the real main problem. First, long-delayed rewards. These cause a build-up of initially irrelevant experiences in the replay memory for Neural Q-learning. The random initialized weights are still trained with these experiences, causing the weights to converge to smaller values with every iteration. Only because the Q-learning applies bootstrapping.

Two methods have been applied to counter the above problem. One is reward shaping. Small rewards were given with every action that leads closer to the goal. This was not enough to overcome the learning noise of the random initialization.

Second is the use of stacked convolutional auto-encoders. This solved the problem partially. These unsupervised learning layers do not depend on if an input is task-relevant. All inputs have equal relevance. That did not help with the size and information density of the inputs. The amount of convolution filters needed to recognize all features is higher than was possible to test within the time and hardware limits. 64 features are what is needed at least when doing reinforcement learning. But, unsupervised learning needs more, because it also learns less useful visual features.

More neurons could have been possible if a more efficient implementation was used, e.g. calculate it on GPU instead of CPU. Recommendations for future research are 1) to first find the right amount of first-layer visual features, get the correct hardware, and then continue. 2) To solve the problem of the high number of initially irrelevant experiences. Such a design needs to consider that any increase in correlation between learning samples affects the off-policy learning property of the Neural Q-learning algorithm. Adding eligibility traces for neural Q-learning should be considered.

Bibliography

[1] Suzanna Becker. Unsupervised learning procedures for neural networks. *International Journal of Neural Systems*, 2(01n02):17–33, 1991.

[2] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 2012.

[3] Chaitanya Ekanadham, S Reader, and H Lee. Sparse deep belief net models for visual area V2. *Advances in Neural Information Processing Systems*, 20, 2008.

[4] Xiaoxiao Guo, Satinder Singh, Honglak Lee, Richard L Lewis, and Xiaoshi Wang. Deep learning for real-time atari game play using offline monte-carlo tree search planning. In *Advances in Neural Information Processing Systems*, pages 3338–3346, 2014.

[5] Simon S Haykin. *Neural networks and learning machines*, volume 3. Pearson Education Upper Saddle River, 2009.

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

[7] Pieter P Jonker, Robert PW Duin, and Dick de Ridder. Pattern recognition for metal defect detection. *Steel Grips*, 1(1):20–23, 2003.

[8] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154. IEEE, 2004.

[9] Varun Raj Kompella, Leo Pape, Jonathan Masci, Mikhail Frank, and Jürgen Schmidhuber. Autoincsfa and vision-based developmental learning for humanoid robots. In *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pages 622–629. IEEE, 2011.

[10] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[11] Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. Stacked convolutional auto-encoders for hierarchical feature extraction. In *Artificial Neural Networks and Machine Learning–ICANN 2011*, pages 52–59. Springer, 2011.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[13] Roland E Suri and Wolfram Schultz. Learning of sequential movements by neural network model with dopamine-like reinforcement signal. *Experimental Brain Research*, 121(3):350–354, 1998.

[14] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* Cambridge Univ Press, 1998.

[15] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic robotics.* MIT press, 2005.

[16] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[17] Cornelius Weber and Jochen Triesch. A sparse generative model of V1 simple cells with intrinsic plasticity. *Neural Computation*, 20(5):1261–1284, 2008.