Project report  
27-06-2017  
Natural Computing

*Tackling AI and control theory problems using Deep Reinforcement Learning*

* *Eastern Screech Owls*

Jordi Beernink, RoelBouman and Jeffrey Luppes

Contents

[Introduction 3](#_Toc486370631)

[Literature and Theory 3](#_Toc486370632)

[*Deep Q-Networks (DQN)* 4](#_Toc486370633)

[*Dueling Network DQN* 5](#_Toc486370634)

[*Cross Entropy Method for Reinforcement Learning* 5](#_Toc486370635)

[Experimental Setup 5](#_Toc486370636)

[*The Cart-Pole balancing problem* 6](#_Toc486370637)

[*The up swinging pendulum problem* 7](#_Toc486370638)

[Results 9](#_Toc486370639)

[*The Cart-Pole balancing problem* 9](#_Toc486370640)

[*The up swinging pendulum problem* 13](#_Toc486370641)

[Conclusion 14](#_Toc486370642)

[References 15](#_Toc486370643)

[Appendix A: Evaluation 16](#_Toc486370644)

[Appendix B: Self-Evaluation Jeffrey 17](#_Toc486370645)

[Appendix B: Self-Evaluation Jordi 18](#_Toc486370646)

[Appendix C: Self-Evaluation Roel 19](#_Toc486370647)

[Appendix D: Github repository 20](#_Toc486370648)

# Introduction

In recent years, deep learning has successfully become a staple method in various domains, such as object recognition with visual attention, Computer Aided Diagnosis and Detection in medical imaging (CADe), and many other regression and classification problems. More recently, deep learning has been combined with reinforcement learning resulting in the creation of the new field of Deep Reinforcement Learning [1,2]. This newly developed method combines Q-learning[3], a Reinforcement Learning technique, with Deep Neural Networks, and was trained to play classic Atari 2600 games and acquired great acclaim in the AI community. This trained Deep Q-network, as the new Reinforcement Learning agent is called, was able to surpass human performance across several of 49 different games. More recent applications involve high-dimensional robot control, solving physics-based control problems, and the playing of the traditional Go game.[4]

On the 27th of April 2016, a large scale platform for Reinforcement Learning called OpenAIGym[5] was launched. OpenAI gym provides a Python environment for the fair comparison of Reinforcement Learning techniques. The platform supports many classical problems (‘Environments’) from Reinforcement Learning theory, such as balancing a pole atop a minecart, balancing a pendulum from a random starting position, and driving a mine cart up a hill from a local minimum using momentum. More advanced problems are also presented, such as several of the classic Atari 2600 games, several games from the PyGame platform, and the classic shooter DOOM.

Deep Q-Learning methods, and other (Deep) Reinforcement Learning techniques, have already proven to be highly applicable to several of the problems represented on the OpenAI Gym platform.

The goal of this project is to compare and evaluate several (Deep) Reinforcement Learning techniques using the OpenAI Gym platform. Especially well known problems from control theory, such as balancing a pole atop a minecart, and swinging up a pendulum, are covered in great detail. Several methods have been used to attempt to solve these problems, and different architectures and settings are covered, so as to thoroughly compare these methods. Preliminary work on harder learning objectives, such as Atari2600 games was briefly done, but proved to be unfeasible given the limited resources. The possible improvements and conclusions of the research will be discussed in detail.

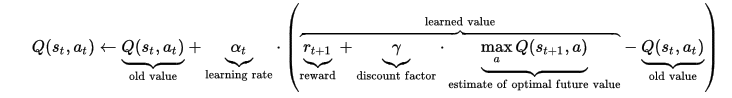
# Literature and Theory

*Reinforcement Learning*

Reinforcement Learning is a field of machine learning specialized in the controlling of actions in an environment. An Agent, which is often a highly specialized program or algorithm, determines which actions to take so as to maximize the reward given a certain state of the environment. This can effectively be seen as a Markov Decision Process, or MDP. Reinforcement Learning is an unsupervised learning technique, as there is no notion of what the true best action given a certain state is, it only approximates. Often in Reinforcement Learning, the entire state-action space is too large to effectively calculate using brute force methods. Because of this limitation, other techniques for approximation have to be used.

*Q-Learning*

Q-Learning is a basic technique in reinforcement learning that attempts to find the ideal action for a Markov Decision Process.[3] It adapts simple dynamic programming principles to take into account the possible cumulative reward when deciding which action to take. Given a certain state from the set *S,* and *A,* the set of action possible in a state, the algorithm tries to maximize the value of function *Q*. The value of *Q* is updated as the state-action space is explored, and it thus simply iterates while updating. Updating is done according to fequation 1, where *s*t and *at* indicate the specific state and action from *S* and A, respectively, at step *t*. This formula is more often known as the Bellman equation.



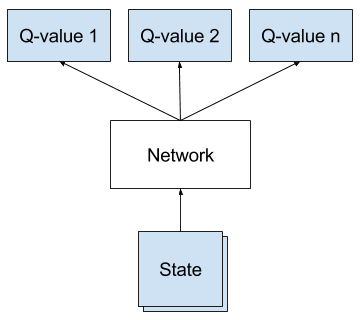
*Equation 1: The iterative updating function for Q, or Bellman equation, used in Q learning. [Source: Wikimedia Commons]*

Note that the formula also contains a learning rate and discount factor, the first indicates the factor by which the new Q value is updated, the latter can control the importance of both direct and long-term rewards. It should be noted that the initial value of Q, before exploration of the state-action space, is determined by the controller of the algorithm, often it is set high, as to encourage initial exploration of the space.

In practice, the state-action space size greatly limits the usefulness of the algorithm, as the tables often used for lookup in dynamic programming quickly outgrow the limits of physical memory.

# *Deep Q-Networks (DQN)*

Deep Q-Networks are perhaps the most notable of the recently developed Reinforcement Learning techniques, having provided a method of overcoming the memory limitations of Q-learning. The Google subsidiary DeepMind has achieved great performance with their Q-networks[1,2]. The Deep Q-Network method uses Deep Neural Networks for the approximation of Q. A neural network represents the Q-function, and takes state-space input to calculate a value for Q corresponding to a certain state-action combination. In practice, several types of networks can be used for representing the Q function. The network takes the state as input, optionally with information regarding previous states and action, and predicts a Q value for each possible action. A simple illustration, as presented by the famous DQN paper, can be found in figure 1.



*Figure 1: Architecture of a Deep Q-network, the specific architecture used for predicting multiple Q values can be credited to Google.*

# *Dueling Network DQN*

In recent years several improvements have been made upon the staple Deep Q-Network technique. One of these is the Double Deep Q-Network, which combines Deep Reinforcement Learning with Double Q-learning[6]. In the Double Q-learning variant, the action selection is decoupled from the evaluation, which leads to less overoptimism due to estimation errors.

The architecture proposed in Double Deep Q-Networks was further extended into an architecture, called a dueling network architecture, where the state value function and the state-dependent action advantage are completely decoupled.[7] This procedure leads to the development of a more robust agent, which in turn leads to a better policy evaluation when many actions are similarly valued, which was shown to be a weakness of Deep Q Networks.

# *Cross Entropy Method for Reinforcement Learning*

The Cross Entropy method is an algorithm for global optimization.[8] It has been shown to perform quite well on several reinforcement learning problems, and it has successfully been applied to the NP-hard Tetris game.[9] The algorithm generates random policies in the form of several Gaussian distributions, and updates these distributions to best resemble the optimal policy after trial and error. Practical implementations sample from the distributions with noise, so as to stop early convergence, and store the distributions by modeling them with a neural network.

*Deep Deterministic Policy Gradient (DDPG)*

In its beginnings, DDPG was quite similar to Cross Entropy based methods, as the underlying class of Policy-Gradient algorithms upon which DDPG is based also assumes a noisy probability distribution over actions. A problem in Reinforcement learning Policy-Gradient based methods greatly suffer from is the credit assignment problem, which states that it is hard to determine which of a series of actions had the most impact towards the reward. To alleviate this problem, like Dueling Network DQN’s, the state value function and the state-dependent action advantage are decoupled by using modeling using two neural networks, one for each. The decoupling is a little more advanced than in Dueling Network DQN’s, as it can more accurately model the time factor.

# Experimental Setup

For the development of the architecture, the well-developed Keras library[10] for the Python language was used. Keras is a library developed for rapid prototyping of Neural Network applications, and it offers an extra layer of abstraction over either the Theano or TensorFlow libraries. Many different Neural Network structures can be tested using the Keras library, without the need to build the architecture from the ground up.

The inclusion of Deep Reinforcement Learning into the process is done using the Keras-rl library[11], which is an extension upon Keras focusing on Reinforcement Learning. It has several Reinforcement Learning techniques available, and can easily be extended. In addition, the Keras-rl library has built-in support for the OpenAI Gym platform, which was elaborated on earlier, and can thus easily be used on various challenges and easy implementation of a reward system.

From the OpenAI Gym platform, the well-known control theory problems Cart-Pole and the up swinging pendulum will be covered.

# *The Cart-Pole balancing problem*

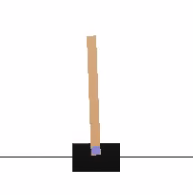


Figure 2: A visualization of the Cart-Pole problem from Control Theory.

A pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The system is controlled by applying a force of +1 or -1 to the cart, thus there are two possible actions. The pole starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every time step that the pole remains upright. The episode ends when the pole is more than 20.9 degrees from vertical, or the cart moves more than 2.4 units from the center. This in turn leads to the cart going off-screen. The OpenAI Gym platform defines the problem as solved when an average reward of 195.0 over 100 consecutive episodes is attained[12]. Four features are passed to the Agent; the cart position, cart velocity, pole angle, and pole velocity at the tip.

***Architecture of the CartPole Experiment***

The CartPole experiment is quite simple, a similar Neural Network architecture is used for each of the different agents. A schematic overview of the architecture can be found in figure 3.

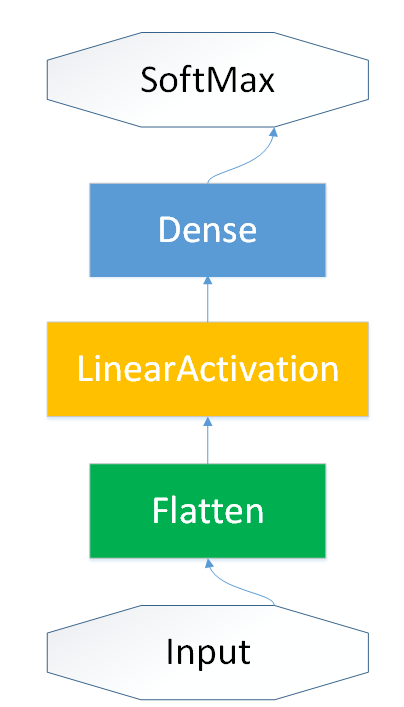


Figure 3: Basic Neural Network architecture for the Cart-Pole Experiment.

The output of the CartPole simulator is flattened, or appended to form a single feature vector. This is followed by a Dense layer of 16 neurons with a Linear activation function. This is then followed by a SoftMax function is directly used in the CEM agent. In the case of the dueling DQN agent, two networks are used.

The CEM agent starts with a warm up session of 2000 steps. The training run will be 100.000 steps; this will make it possible for each architecture to converge to the ideal score. Lowering this amount to 10.000 or even 1.000 will greatly hinder the convergence and will give a biased view of some of the specific architectures. In contrast to the CEM agent, the Dueling Network DQN agent will start with a warm up of a 1000 steps.

***Execution of the CartPole Experiment***

The first section of the experiment will focus on comparing different amounts of stacked layers in combination with three different activation functions: sigmoid[13], linear and relu[14], in different layer stacks. The architecture is identical for both the CEM and Dueling DQN agents. By increasing the amount of layers both the time, mean reward and standard deviation can be controlled to achieve the ideal balance between performance and training time. The scores will be based on rerunning each specific architecture five times and grabbing ten random test episodes from each run. This will result in an unbiased score that will be compared with the other combinations in that specific architecture and agent.

The second part consists of selecting the best three architectures per agent and increasing the neurons of these layers by doubling the amount per increment, this goes from 16 up to 256. This prevents bloated training times for intensive architectures with for example, ten layers.

The last part of this experiment will grab the overall best scoring method per agent and adds a dropout layer to the neural network while having only 16 neurons to prevent a bloated training time. By adding a dropout layer, possible overfitting might be prevented. Whether networks easily overfit in these cases may not easily be determined, as the problems themselve are fairly simple. A schematic overview of the network including the PreLU and dropout layer can be found in figure 4. The PreLU layer is added, as a PreLU layer with dropout has been proven to work in practice on many machine learning benchmark contests.

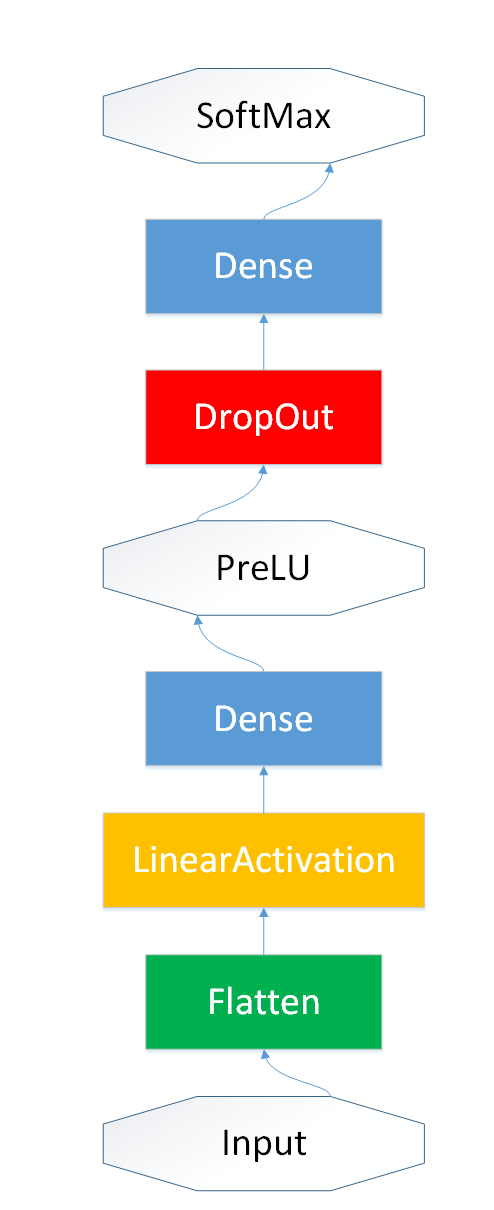


Figure 4: Neural Network architecture including dropout and PreLU layers for the Cart-Pole Experiment.

# *The up swinging pendulum problem*

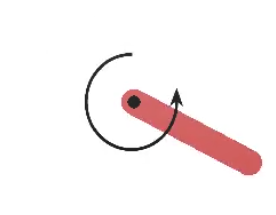


Figure 5: A visualization of the up swinging pendulum problem from Control Theory.

This environment is based on the inverted pendulum swing up problem. The pendulum starts in a random position, and the goal is to swing it up so it stays upright. This problem is currently in an unsolved environment, which means it does not have a specified reward threshold at which the problem is considered solved. This means that an arbitrary amount of episodes can be used to achieve the highest possible score [15]. Three features are passed to the Agent; the cosine of the angle, the sine of the angle and the pendulum velocity at the tip. The exact reward can be found in equation 2.

Clearly, this minimizes the amount of actions taken, and aims to keep the pendulum upright with as little movement as possible. Note that the score is negative, and it is ideal to have this be as close to 0 as possible.

***Architecture of the Pendulum Experiment***

The experimental setup of this experiment is quite like that of the Cart-Pole experiment, at least in terms of network architecture. Instead of a CEM or Dueling DQN agent, a DDPG agent will be used to best test the effects of the continuous approach. As two networks are needed, one for the state value function, and another for the state-dependent action advantage, two are presented in the following section. It must be noted that a Ornstein Uhlenbeck process is used for the stochastic element in the DDPG agent.

The state value function network is similar to the network used for the Cart-Pole experiment, with the exception that the final layer is a Linear layer instead of a Softmax layer, the resulting architecture is visualized in figure 6.

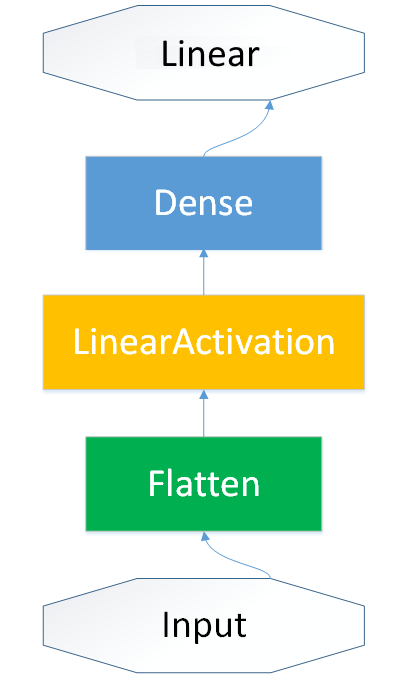


Figure 6: Basic Neural Network architecture for state value function in the Pendulum Experiment.

The second network, modeling the state-dependent action advantage, is similar to the first, with the exception that a larger number of layers is used. Note that throughout the experiments the second network is not modified. Modifying the second network in addition to the first would yield too many options to reliably test within reasonable time. The architecture of the second network can be found in figure 7.

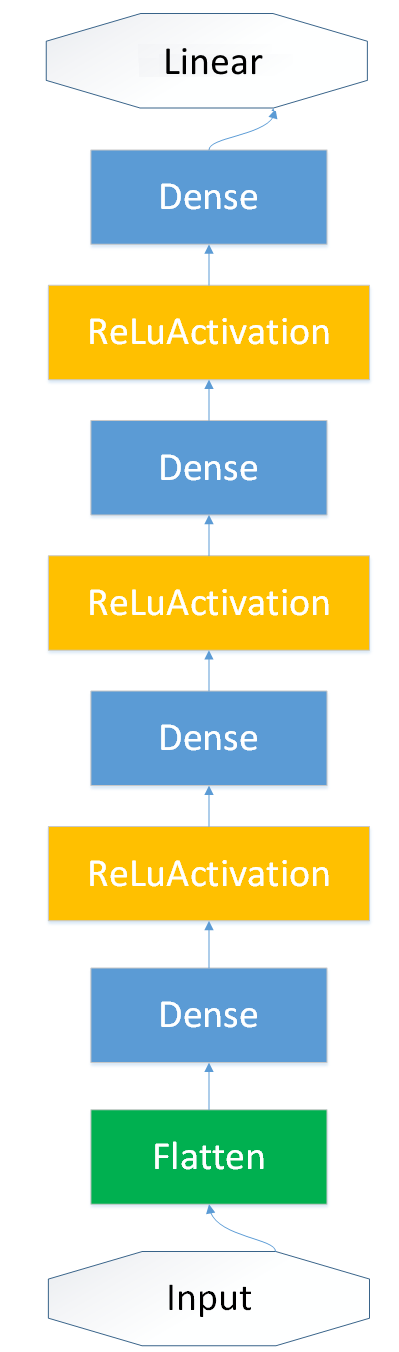


Figure 7: Basic Neural Network architecture for state-dependent action advantage in the Pendulum Experiment.

The warmup in this experiment consists of 5000 steps. As the problem can not be solved according to the specifications, the training time can be limited more efficiently. As such, 5000 steps are used for testing and training purposes, ensuring that a decent action policy must be found in time.

***Execution of the CartPole Experiment***

The first section of the experiment will focus on comparing different amounts of stacked layers in combination with three different activation functions: sigmoid[13], linear and relu[14], in different layer stacks. Note that the architecture is only changed for the state value function network. The scores will be based on rerunning each specific architecture five times and grabbing ten random test episodes from each run.

The second part consists of selecting the best three architectures per agent and increasing the neurons of these layers by doubling the amount per increment, this goes from 16 up to 256.

# Results

# *The Cart-Pole balancing problem*

***The optimal activitation function and amount of layers for CEM***

The Cart-Pole problem is bounded, which means there is a maximum score of 200 that can be obtained. To study the performance of different network architectures the amount of layers is varied for ReLU, Sigmoid and Linear activation functions. The resulting mean scores, including standard deviations, and the respective training times for each possible combination can be found in Table 1.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Layers** | **RELU** | **Sigmoid** | **Linear** | **RELU Time** | **Sigmoid Time** | **Linear Time** |
| **1** | 129.6 +/- 26.49 | 98.9 +/- 75.18 | 123.4 +/- 63.57 | *20.226* | *19.587* | *17.231* |
| **2** | 176.6 +/- 40.43 | 73.2 +/- 11.01 | 133.4 +/- 60.43 | *19.652* | *22.047* | *18.534* |
| **3** | 164.4 +/- 22.01 | 81.6 +/- 17.79 | 164.1 +/- 21.34 | *19.899* | *23.578* | *19.344* |
| **4** | 142.6 +/- 53.89 | 9.5 +/- 0.5 | 182.4 +/- 12.67 | *22.914* | *24.732* | *21.031* |
| **5** | 109.3 +/- 46.67 | 9.5 +/- 0.5 | 189.3 +/- 18.4 | *23.432* | *25.232* | *22.983* |
| **6** | 77.9 +/- 30.7 | 9.1 +/- 0.7 | 162.6 +/- 40.21 | *25.327* | *25.823* | *23.653* |
| **7** | 94.3 +/- 56.74 | 9.4 +/- 0.49 | 189.9 +/- 16.22 | *24.973* | *26.321* | *23.843* |
| **8** | 61.3 +/- 20.52 | 9.3 +/- 0.46 | 175.5 +/- 25.74 | *26.483* | *27.164* | *24.721* |
| **9** | 169.9 +/- 38.26 | 9.2 +/- 0.54 | 158 +/- 38.39 | *28.293* | *28.432* | *25.926* |
| **10** | 62.2 +/- 33.53 | 9 +/- 0.63 | 169.8 +/- 30.13 | *28.765* | *29.568* | *28.322* |

*Table 1: mean scores, including standard deviations, and the respective training times for each combination of 1-10 layers and a ReLU, Sigmoid or Linear activation function for a network using in conjunction with the CEM agent.*

Several things can be concluded from table 1. The Sigmoid activation function is significantly outperformed by both the ReLU and Linear functions across all layer sizes. Furthermore, the size of the network does not seem to significantly affect the network. It is likely that all of these observations stem from a similar origin. In CEM the network is used to model distributions and their parameters. Due to the small search space, it is likely that only few layers are needed to accurately model these distributions, so increasing the amount of layers beyond the first few offers no improvement. For Linear activation functions, between 4-7 layers seems to be optimal, while 2-3 seems optimal for ReLU activation functions. The Linear function based model seems to consistently perform better, and significantly outperforms the ReLU function on similar network sizes. It is furthermore hypothesized that the Linear function outperform both others because of its property to more accurately model consistent variables needed for distributions. The sigmoid layers are probably too unstable for proper modeling. The rectification in the ReLU function probably also distorts this property of the Linear function.

***The optimal number of neurons per layer for CEM***

Several numbers of neurons have been tested for the most promising layer sizes found in the previous section. For ReLU with 2 layers, and Linear with 5 or 7 layers, the number of neurons has been varied from 16 to 256 neurons. The resulting mean scores and standard deviations can be found in table 2.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CEM | Layers | RELU | Sigmoid | Linear | RELU Time | Sigmoid Time | Linear Time |
| 20 | 1 | 129.6 +/- 26.49 | 98.9 +/- 75.18 | *123.4* +/- 63.57 | *20.226* | *19.587* | *17.231* |
| 40 | 2 | 176.6 +/- 40.43 | 73.2 +/- 11.01 | *133.4* +/- 60.43 | *19.652* | *22.047* | *18.534* |
| 60 | 3 | 164.4 +/- 22.01 | 81.6 +/- 17.79 | *164.1* +/- 21.34 | *19.899* | *23.578* | *19.344* |
| 80 | 4 | 142.6 +/- 53.89 | 9.5 +/- 0.5 | *182.4* +/- 12.67 | *22.914* | *24.732* | *21.031* |
| 100 | 5 | 109.3 +/- 46.67 | 9.5 +/- 0.5 | *189.3* +/- 18.4 | *23.432* | *25.232* | *22.983* |
| 120 | 6 | 77.9 +/- 30.7 | 9.1 +/- 0.7 | *162.6* +/- 40.21 | *25.327* | *25.823* | *23.653* |
| 140 | 7 | 94.3 +/- 56.74 | 9.4 +/- 0.49 | *189.9* +/- 16.22 | *24.973* | *26.321* | *23.843* |
| 160 | 8 | 61.3 +/- 20.52 | 9.3 +/- 0.46 | *175.5* +/- 25.74 | *26.483* | *27.164* | *24.721* |
| 180 | 9 | 169.9 +/- 38.26 | 9.2 +/- 0.54 | *158* +/- 38.39 | *28.293* | *28.432* | *25.926* |
| 200 | 10 | 62.2 +/- 33.53 | 9 +/- 0.63 | *169.8* +/- 30.13 | *28.765* | *29.568* | *28.322* |

*Table 2: mean scores, including standard deviations, and the respective training times for ReLU with 2 layers, and Linear with 5 or 7 layers, the number of neurons has been varied from 16 to 256 neurons.*

From these results it can be verified that the Linear activation function indeed performs best. A Linear activation function with 64 Neurons per layer and 5 layers consistently outperforms all others, and scores near perfectly on the problem.

***DropOut in CEM***

Further experimenting was done with the addition of DropOut layers. For the Linear activation function with 16 Neurons per layer and 5 layers, a PreLU + DropOut layer was added. The resulting mean scores can be found in table 3.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| CEM Model | *16* | *32* | *64* | *128* | *256* | *Dropout 16* | *Mean* |
| 2 Layer Relu | 176.6 +/- 40.43 | 181.3 +/- 18.9 | 191.8 +/- 8.87 | 100.8 +/- 41.07 | 170.2 +/- 45.77 |  | 164.14 +/- 31.01 |
| 5 Layer Linear | 189.3 +/- 18.4 | 197.5 +/- 5.12 | 198.7 +/- 3.29 | 188.1 +/- 15.16 | 122.5 +/- 44.71 | 135.6 +/- 58.32 | 171.95 +/- 17.34 |
| 7 Layer Linear | 189.9 +/- 16.22 | 168.2 +/- 38.86 | 175.8 +/- 26.05 | 130.7 +/- 69.62 | 166.2 +/- 29.83 |  | 166.16 +/- 36.11 |

*Table 3: The effects of adding a PreLU+DropOut layer on CEM performance.*

Note that the testing was done on the initial 16 neuron length layers. This probably causes a network that is too sparse, and thus is not able to accurately model the distributions, hence the low mean scores. To test dropout more accurately, the number of neurons should be doubled when applying p=0.5 dropout, as is recommended in literature.

***The optimal activitation function and amount of layers for DQN***

The same network sizes and activation functions that were tested for CEM were tested for DQN. The results can be found in Table 4.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| DQN | Layers |  |  |  | RELU Time | Sigmoid Time | Linear Time |
| 20 | 1 | 199.7 +/- 0.32 | 198 +/- 4 | *8.6* +/- 0.49 | *123.325* | *133.514* | *135.678* |
| 40 | 2 | 195.8 +/- 11.95 | 137.8 +/- 24.95 | *11.3* +/- 2 | *133.459* | *149.383* | *141.637* |
| 60 | 3 | 190.6 +/- 17.96 | 147.3 +/- 5.25 | *9.2* +/- 0.87 | *142.453* | *162.883* | *149.099* |
| 80 | 4 | 170.3 +/- 23.79 | 47.3 +/- 76.35 | *9* +/- 0.89 | *151.863* | *173.522* | *156.827* |
| 100 | 5 | 158.9 +/- 32.74 | 108.5 +/- 35.11 | *9.3* +/- 0.64 | *159.732* | *195.743* | *169.139* |
| 120 | 6 | 180.9 +/- 23.59 | 9.3 +/- 0.46 | *9.4* +/- 0.66 | *168.384* | *208.412* | *176.995* |
| 140 | 7 | 193.4 +/- 9.44 | 112.9 +/- 20.18 | *8.9* +/- 0.7 | *183.432* | *227.002* | *186.488* |
| 160 | 8 | 188.5 +/- 14.6 | 50.2 +/- 62.99 | *9.6* +/- 1.02 | *201.282* | *256.843* | *189.775* |
| 180 | 9 | 160.4 +/- 20.74 | 9 +/- 0.89 | *9.6* +/- 0.8 | *199.637* | *275.118* | *199.013* |
| 200 | 10 | 199.2 +/- 0.46 | 9.4 +/- 0.49 | *9.5* +/- 0.92 | *206.954* | *293.954* | *206.912* |

*Table 4: mean scores, including standard deviations, and the respective training times for each combination of 1-10 layers and a ReLU, Sigmoid or Linear activation function for a network using in conjunction with the DQN agent.*

It seems that the Linear activation function based networks are unable to accurately model the Q function, and are therefore outperformed by Sigmoid and ReLU activation functions, who perform about as well. It seems that stacking sigmoid layers is not beneficial, which is to be expected due to the more unstable thresholding method. For this problem, it seems that simple single layer networks with either ReLU or Sigmoid activation functions perform just as well, and none can be deemed significantly better than the other based on these results. It must be noted that the training times are 7-10 times greater than in the CEM case, which is a clear disadvantage, nonetheless, DQN slightly outperforms CEM without much optimization.

***The optimal number of neurons per layer for DQN***

Similar to the CEM experiments, several layer sizes have been tested for the best performing structures found in the previous section, again varying from 16 to 256. The results of these tests can be found in table 5.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *DQN* | *16* | *32* | *64* | *128* | *256* | *Dropout (16)* | *Mean* |
| 1 Layer Relu | 199.7 +/- 0.32 | 197 +/- 4.58 | 196.5 +/- 5.5 | 199 +/- 3 | 199.5 +/- 1.5 | 166.8 +/- 9.18 | 193.08 +/- 2.98 |
| 10 Layer Relu | 199.2 +/- 0.46 | 187.5 +/- 14.79 | 132.2 +/- 6.87 | 90.3 +/- 84.82 | 197.2 +/- 5.93 |  | 161.28 +/- 22.57 |
| 1 Layer Sigmoid | 198 +/- 4 | 199.5 +/- 0.81 | 198.2 +/- 2.79 | 195.6 +/- 5.43 | 179.7 +/- 44.07 |  | 194.2 +/- 11.42 |

*Table 5: mean scores, including standard deviations, and the respective training times for ReLU with 1 and 10 l*a*yers, and Linear with 1 layer, the number of neurons has been varied from 16 to 256 neurons.*

As is to be expected, little changes when we change the number of neurons. Performance was already near maximum capacity, and it does not significantly change when increasing the layer size.

***DropOut in CEM***

Much like CEM experimenting was done with the addition of DropOut layers for DQN. For the ReLU activation function with 16 Neurons per layer and 1 layers, a PreLU + DropOut layer was added. The resulting mean scores can be found in table 6.

|  |  |  |
| --- | --- | --- |
|  | Relu 1 Layer No DropOut | Relu 1 Layer DropOut |
| Mean Score | 199,7 +/- 0,332 | 166,8 +/- 9,185 |

*Table 6: The effects of adding a PreLU+DropOut layer on DQN performance.*

Again, much like was the case for CEM, the addition of a DropOut layer significantly decreases performance, probably due to the same factors that influenced the low score in CEM.

# *The up swinging pendulum problem*

***The optimal activitation function and amount of layers for DQN***

Much like was the case for CEM and DQN, several activation functions and number of layers have been tested and are thoroughly compared. The resulting mean scores can be found in table 7.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| DDPG | Layers | Relu | Sigmoid | Linear | RELU Time | Sigmoid Time | Linear Time |
| 20 | 1 | -1530.2 +/- 77.48 | -1480.1 +/- 64.21 | -1445.4 +/- 48.07 | *86.128* | *85.776* | *85.793* |
| 40 | 2 | -1487.9 +/- 44.07 | -1453.3 +/- 51.12 | -1480.5 +/- 24.32 | *86.171* | *86.129* | *85.843* |
| 60 | 3 | -1442.9 +/- 77.06 | -1480.3 +/- 49.6 | -1394.7 +/- 434.34 | *85.989* | *86.024* | *86.132* |
| 80 | 4 | -1484.8 +/- 54.15 | -1418.3 +/- 74.25 | -1474.1 +/- 27.42 | *85.853* | *86.039* | *86.187* |
| 100 | 5 | -1479.1 +/- 44.58 | -1520 +/- 32.33 | -1451.9 +/- 33.05 | *85.986* | *86.051* | *86.342* |
| 120 | 6 | -1421.3 +/- 97.96 | -1380 +/- 47.57 | -1431.7 +/- 69.25 | *86.432* | *86.347* | *86.398* |
| 140 | 7 | -1455.9 +/- 73.93 | -1447.1 +/- 57.74 | -1484.4 +/- 166.17 | *86.862* | *86.515* | *86.432* |
| 160 | 8 | -1436.86 +/- 69.68 | -1461 +/- 61.37 | -1397.2 +/- 64.18 | *86.342* | *86.472* | *86.445* |
| 180 | 9 | -1483.7 +/- 72.48 | -1342.5 +/- 149.04 | -1468.4 +/- 33.7 | *87.203* | *86.746* | *86.663* |
| 200 | 10 | -1466.2 +/- 42.36 | -1528.6 +/- 164.15 | -1458.3 +/- 81.86 | *87.109* | *86.763* | *86.701* |

*Table 7: mean scores, including standard deviations, and the respective training times for each combination of 1-10 layers and a ReLU, Sigmoid or Linear activation function for a network using in conjunction with the DDPG agent.*

Scores for this problem are much closer to each other, most likely due to the relatively harsh scoring metric. Nonetheless, some methods perform significantly better than others. Most notable is the architecture using the sigmoid activation function, where the architectures with 5 and 9 layers perform quite well, with small standard deviations for the 5 layer architecture. It is likely that in each case, the DDPG method uses a lot of actions, and therefore scores relatively low. It must be noted that some of the Linear structures, most notable those with 3 and 8 layers, don’t perform significantly worse. It is important to note the standard deviation for the 3 layer Linear model, as it is extremely high. Perhaps a more stable model with 3 layers and a Linear can be found using more neurons.

***The optimal number of neurons per layer for DDPG***

For the most notable structures with low standard deviations, the number of neurons per layer has been varied from 16 to 256. This yields several scores, all of which can be found in table 8.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *DDPG* | *16* | *32* | *64* | *128* | *256* | Mean |
| Sigmoid 6 Layers | -1380 +/- 57.57 | -1460.2 +/- 75.9 | -1466.8 +/- 49.73 | -1494.4 +/- 77.43 | -1521.2 +/- 91.29 | -1464.52 +/- 70.38 |
| Linear 3 Layers | -1394.7 +/- 434.34 | -1395.6 +/- 155.5 | -1491.8 +/- 41.61 | -1404 +/- 106.3 | -1331.7 +/- 445.79 | -1403.56 +/- 236.71 |
| Linear 8 Layers | -1388.3 +/- 64.18 | -1469.3 +/- 44.58 | -1413.8 +/- 50.83 | -1319.6 +/- 441.79 | -1454.4 +/- 76.76 | -1409.08 +/- 135.63 |

*Table 8: mean scores, including standard deviations, and the respective training times for Sigmoid with 6 layers, and Linear with 3 or 8 layers, the number of neurons has been varied from 16 to 256 neurons.*

From table 8 it can be concluded that for the 6 layer Sigmoid and the 8 layer linear model, more neurons do not seem to improve the score. The 4 layer linear model does not seem to have any merit based on this data, as it is quite unreliable due to its high standard deviations, and its generally low scoring results.

# Conclusion

Several things can be concluded from this study. For the simple Cart-Pole problem, Both CEM and DQN based methods were able to perform near perfectly. For a simple problem such as this, CEM is preferable, as it can train in little time, and performs just as well as DQN. For this specific problem, neither increasing or layer size, or the addition of PReLU+DropOut layers seemed to positively impact performance. For CEM, linear layer structures, such as ReLU layers or normal Linear layers perform well, while for DQN Sigmoid and ReLU layers far outperform the Linear layers. The close ties between various architectures verify that a wide range of architectures must be explored to find the optimal one. Often the best choice is made with Occam’s razor, choosing the simplest one. In other cases, optimization techniques such as Ant Colony optimization or Genetic Algorithms might be useful for the construction of an optimal network architecture.

DDPG has been successfully applied on the up swinging pendulum problem. Many different network architectures have been tested. The most succesful architectures are moderately deep, using 6-8 layer 16 neurons networks, which use either Linear or Sigmoid activation functions. It is hypothesized that the DDPG agent takes too many actions, which negatively impacts the score on the problem a more conservative agent might be more useful. Many successful agents suffer from taking a great number of actions, as can clearly be seen in studies of reinforcement learning on the Atari2600, where agents show erratic behavior in various games. A more simplistic approach might be more controlled, and might be preferable in conditions where the taking of actions is costly, such as in robot control or applied AI.

Several interesting conclusions have been drawn from these experiments, and these provide valuable insights into reinforcement learning, which might extend to more useful platforms, such as applied control theory.

# References

1. Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." *Nature* 518.7540 (2015): 529-533.
2. Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).
3. Watkins, Christopher JCH, and Peter Dayan. "Q-learning." *Machine learning* 8.3-4 (1992): 279-292.
4. Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." *Nature* 529.7587 (2016): 484-489.
5. Brockman, Greg, et al. "OpenAI gym." *arXiv preprint arXiv:1606.01540* (2016).
6. Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep Reinforcement Learning with Double Q-Learning." *AAAI*. 2016.
7. Wang, Ziyu, et al. "Dueling network architectures for deep reinforcement learning." *arXiv preprint arXiv:1511.06581* (2015).
8. De Boer, Pieter-Tjerk, et al. "A tutorial on the cross-entropy method." *Annals of operations research* 134.1 (2005): 19-67.
9. Szita, István, and András Lörincz. "Learning Tetris using the noisy cross-entropy method." *Learning* 18.12 (2006).
10. Chollet, François. "Keras." (2015).
11. Plappert, Matthias. "keras-rl" (2016).
12. Barto, A., Anderson, C., & Sutton, R. (2015). *CartPole version 0*. Retrieved May 23, 2017, from OpenAI Gym: https://gym.openai.com/envs/CartPole-v0
13. Gibbs, M. (2000). Variational Gaussian process classifiers. *IEEE Transactions on Neural Networks, 11*(6), 1458-1464. Retrieved June 25, 2017, from http://ieeexplore.ieee.org/document/883477/
14. Nair, V., & Hinton, G. (2010). *Rectified Linear Units Improve Restricted Boltzmann Machines.* Computer Science. Toronto: University of Toronto. Retrieved June 25, 2017, from http://machinelearning.wustl.edu/mlpapers/paper\_files/icml2010\_NairH10.pdf
15. OpenAI Gym. (2015). *Pendulum Version 0*. Retrieved May 23, 2017, from OpenAI Gym: https://gym.openai.com/envs/Pendulum-v0

# Appendix A: Evaluation

The project could be divided into three main phases. The first phase was the orientation in the literature concerning deep reinforcement learning and applying it to a game environment. Everyone had to read several literature articles and needed to have a basic understanding of the different methods, such that everyone could implement the different methods into neural networks or reinforcement learning methods. The second phase was exploring the possibilities of deep reinforcement methods in specific environments. Jeffrey and Jordi focused on experimenting with the simple environments of OpenAI gym such as the Pendulum and CartPole. Roel focused on trying to create an Atari environment that could be used for the final experiments of the project. Sadly, due to memory leaks in the ALE Atari Emulator, it was impossible to run an Atari environment for quite some time. Much work was put into fixing these problems, and it eventually worked. Afterwards it was concluded that the training times for Atari games took up to several days, and it was unfeasible to test on the available hardware. This meant that the simple environments, that Jeffrey and Jordi experimented with, needed to be used. The last phase was creating a project around these simple environments. Experiments with standard control theory problems proved to be fruitful, as the lower training times allowed for thorough testing. We would like to be able to experiment more with more advanced hardware, to actually tackle more advanced problems.

The contributions for the last phase were as followed. Roel focused on the theoretical and methodical aspect of the report as well as the interpretation of the results. Jordi looked at some possible experimenst that could be performed on the CartPole and Pendulum environment. Afterwards the different architectures and parameter tuning was also performed by Jordi. Jeffrey focused on the creating and maintaining a repo for all the progress and worked on the approach of the project.

# Appendix B: Self-Evaluation Jeffrey

**The project**

Undoubtly the project could have gone better. I feel that we were focusing too much on tiny details instead of trying to get a proof of concept up and running as soon as possible. Our initial goal – Doom – was too high. The software is buggier than anything I have ever worked with in an academic setting, and we encountered a lot of issues that could count on zero community support. I feel that this was reflected in our process, as some of us were already working on the final report when we weren’t even running experiments yet. We could have used a little bit more direction and self-discipline. Yet, I think we achieved good results and I have no hard feelings towards the others.

**Personal Performance**

In this group with Jordi (Business Intelligence) who’s a great organizer and Roel (AI) who’s great at having a good grasp on relevant literature, I felt that I from my mixed AI and CompSci background I took on a role of “the CS guy” at first. During the first few weeks I mostly tried to get things up and running and spent many days trying out various set-ups. At first I focused on Ubuntu, and later tried to get as many examples up and running on Windows. The later worked in the end, but it turned out that my solutions were not portable to the others. We then focused on the Atari and Box2D platforms. These too I eventually managed to get working, but not in a portable and stable manner. Roughly half of the Gym environments were not runnable. Roel had gotten a lot working independently, but nothing seemed to work on Jordi’s distro. At that point, I had logged about 80 hours on troubleshooting, which means a lot in exam season! I also set up a Dropbox folder and a Github repository for the group, enabling us to work together and track commits. I remained the Repo organizer.

I then tried to turn away from being “the CS guy” and turned to the report writing, something Jordi had been working on. We switched around, as he could use the programming experience and I hardly ever write reports like these. I ended up rewriting a lot, and did a lot of work on excel in the last few days to generate analysis, working in close cooperation with Roel and Jordi who were running the experiments.

What I brought to the team was a lot of experience debugging and knowledge of various tools. I only wrote some pieces of code, while I could probably have gotten results a lot faster than those who ended up writing more code. I did not want to remain in my comfort zone, so I ended up doing a lot of things that others tend to do.

I honestly wish I could have given it more time, but a couple of problems (e.g. my dad coming down with pneumonia, long story) took a cut out of my time.

# Appendix B: Self-Evaluation Jordi

Overall, I am not satisfied with how this project went with both my own input as the others. Because of my own indecisiveness, we suffered from time constraints at the last minute. Which resulted in my opinion in an incomplete project. Personally, I should have cut the knot earlier with the project so that the entire experiment process could have started earlier and not two days before the deadline. The biggest problem during this entire process was my own procrastination concerning the direction of the project. The original objective was to get a Doom environment working and try to create a neural network that was able to mimic human input. After three weeks of testing to get the environment to work, to no avail, we shifted to Atari. However, after another two weeks this also did not reach anything. Therefore, in a span of a couple of days the entire project needed to be built from the ground-up. This conflicted with people having other appointments and not having the time to clarify the tasks of each person. As a result, this led to people following their own vision of the project. This resulted in a lot of loss of time at the end of the project, which could have been better spend on fine-tuning the report.

An example of this was the literature review and experiment section. These were not finished in sequential order. This meant that the experiment section was finished before the literature review. As a result, the experiment section, a segment that costed me an entire three days to finish, was considered majorly flawed. As a result, this had a negative effect on the motivation on the entire team and led to fixing the experiment section at the last minute. In short, I was indecisive and did not show the initiative to lead the team into finishing this project on time and prevent spending a lot of time on environments that did not show any chance of working after one week. If I could redo this project, I would have focused on something simple and worked this out completely and in the finest details from day one with the team. Afterwards everyone would have the same vision of the project and it would be finished on time. This could have resulted in a more finished project.

My points to work on is showing initiative in the first week, steering the project in the right direction from day one, and not focusing completely on orientation. The second point to work on is showing interest in what people are doing as to get a better idea of what people are specifically doing and why they are doing it. This could result in a better understanding of the project and could help in improving the report.

# Appendix C: Self-Evaluation Roel

The project was a double-edged blade. On one hand, I have learned a lot about Reinforcement Learning, the UNIX platform, OpenAI, Python, Keras and Emulators. On the other hand, I feel many of my efforts went to waste due to various reasons.

I spent a lot of time on fixing various platforms, which were hardly supported to begin with. Sadly I could count on little technical expertise within the group, I was on my own a lot. Because of my technical focus, the others focused on algorithmic issues and experimentation, which could not get my full attention. Near the end Jeffrey provided a lot of extra input, which was very helpful. I feel a lot of mistakes have been made because my colleagues were not knowledge about the matter, nor did they seem to have done any significant literature research, something which showed from the by Jordi produced results section, which needed extreme rewriting last-minute due to mistakes that could have been easily prevented by simply reading about the methods we used. Communication about these problems with Jordi was especially hard, as he rarely seemed to understand what was wrong, and I had to do a fair amount of lecturing on basic experimental machine learning principles.

I believe that, in the end, we delivered a decent report, and the project matter actually covers important aspects of reinforcement learning. It is a shame that much of my time went to waste on trivial matters which shouldn’t have taken up so much time. Nonetheless, the experience has been educational, which is what matters most.

# Appendix D: Github repository

A Github repository containing all of the code used in this project and instructions on installation can be found at <https://github.com/jeffluppes/NaturalComputingDNN> . A link to a virtual machine running Ubuntu with all of the packages installed can also be found in the repository README. The password to this VM is “asdf”. This should allow any reader to test and evaluate the code.