Exploring High Dimensional, Sparse Reward Problems Using Deep Learning and Neuroevolution

Mats Tveter

Thesis submitted for the degree of
Master in Robotics and Intelligent Systems
60 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2021

Exploring High Dimensional, Sparse Reward Problems Using Deep Learning and Neuroevolution

Mats Tveter

Abstract

The field of machine learning has seen a rapid progression in the last decade, heavily contributing to the computer's ability to solve complex problems. However, as the field progresses, so does the complexity of the challenges that need to be solved to advance the field further. One of these challenges is dealing with a huge amount of data and extracting useful information; another challenge is solving tasks that require several "correct" actions in sequence to complete the task. These challenges have been solved separately, but more complex problems might require a singular solution to solve both challenges simultaneously. This study investigates how to solve complex, high-dimensional, and sparse reward problems by combining two separate algorithms that excel at each subpart of the problem. More specifically, deep learning and neuroevolution. Deep learning extracts information from the high-dimensional data, and neuroevolution solves the sparse reward problems.

This thesis aims to show that a combination of deep learning and neuroevolution on high-dimensional and sparse reward problems outperforms either machine learning technique alone. The experiments were run in a reinforcement learning environment, modified to fit the needs of this dissertation. Numerous experiments were conducted with different goals and sub-goals to thoroughly investigate and explore the capabilities of neuroevolution in combination with deep learning. The deep learning model receives the high-dimensional data, and through the process of learning, discovers how to extract compact, useful information from this data. The neuroevolution algorithm utilizes the compact, extracted representation provided by the deep learning model to determine the best possible action given the input.

The results show that deep learning and neuroevolution can perform better when combined than each of them alone on a high-dimensional and sparse problem. Neuroevolution performs well on problems with sparse rewards but struggles with high-dimensionality, and deep learning can easily handle high-dimensionality but struggles with sparse rewards. This combination of deep learning and neuroevolution can be a good performing and viable option in multiple fields where similar problems appear. This research contributes to the field by doing a comparison between deep reinforcement learning, neuroevolution, and neuroevolution+deep learning, and a thorough survey of the field.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Goal of the thesis 2

1.3 Outline 3

2 Background **5**

2.1 Machine Learning 5

2.1.1 Types of Machine Learning 5

2.1.1.1 Supervised Learning 5

2.1.1.2 Unsupervised Learning 7

2.1.1.3 Reinforcement Learning 7

2.1.2 Neural Networks 10

2.1.3 Deep Learning 13

2.1.3.1 Convolutional Neural Network 15

2.1.3.2 Deep Reinforcement Learning 17

2.1.3.3 Challenges in Deep Learning 18

2.2 Evolutionary Computation 20

2.2.1 Evolutionary Algorithm 20

2.2.2 Neuroevolution 24

2.3 Deep Learning and Neuroevolution 26

2.3.1 Dividing the problem 26

2.3.2 DL-component 27

2.3.3 NE-component 31

2.3.4 RL-problems solved 33

2.3.5 Summary and contribution 34

3 Tools and frameworks **35**

3.1 Environment 35

3.1.1 Bipedal Walker 35

3.2 Algorithms 37

3.2.1 SAC 37

3.2.2 NEAT 39

3.2.3 DL 42

3.3 Hardware 46

4 Implementation **47**
4.1 Environment . . . 47
4.1.1 Sparse rewards . . . 49
4.2 Dataset . . . 51
4.3 Algorithms . . . 51
4.3.1 SAC and NEAT . . . 51
4.3.2 DL . . . 52
4.4 Experimental Setup . . . 53
4.5 Experimental Challenges . . . 55

5 Experiments and results **59**
5.1 Non-sparse Fitness Function . . . 59
5.1.1 Sensor Values . . . 60
5.1.1.1 State and Terrain Information . . . 60
5.1.1.2 Only State Information . . . 61
5.1.2 Image Values . . . 63
5.1.3 Summary - Non-Sparse Reward . . . 66
5.2 Sparse Fitness Function . . . 67
5.2.1 Sensor Values . . . 67
5.2.1.1 State and Terrain Information . . . 67
5.2.1.2 Only State Information . . . 69
5.2.2 Image Values . . . 70
5.2.3 Summary - Sparse Rewards . . . 72

6 Discussion **73**
6.1 General discussion . . . 73
6.2 Comparing the results of the algorithms with a non-sparse fitness function . . . 73
6.3 Investigating possibilities of a DL combination . . . 75
6.4 Investigating the performance of the algorithms with high-dimensional input . . . 76
6.5 Investigating performance of the algorithms with a sparse reward function . . . 77
6.5.1 State and terrain sensor input . . . 77
6.5.2 Only state sensor input . . . 77
6.6 High-dimensional, sparse reward problem . . . 78
6.7 DL Combination . . . 79
6.8 Limitations . . . 80

7 Conclusion **83**

8 Future Work **85**

Bibliography **94**

Appendix **95**
A Additional Experiments . . . 95
A.1 NEAT - Fully Connected vs. Unconnected . . . 95
A.2 NEAT - Pretrained NEAT . . . 95

B Supplementary Plots 96
B.1 Sparse Fitness Sensor Levels 96
C Raw plots 100

List of Figures

2.1 Supervised Learning 6
2.2 Reinforcement Learning 8
2.3 Artificial and Biological Neurons 10
2.4 Neural Network 11
2.5 DL vs. Conventional Machine Learning 14
2.6 Deep Learning Face-classifer 14
2.7 VGG16 Architecture 15
2.8 Convolutional Layer 16
2.9 Filter Abstraction 16
2.10 Max-Pooling 16
2.11 Evolutionary Algorithm 21
2.12 Variation Operators 23
2.13 SANE and ESP 25
2.14 DL and NE - Dividing the problem 27
2.15 NE+DL Combination 27
2.16 UL-ERL 28
2.17 Variational Autoencoder 29
2.18 VAE+NE 29
2.19 Various NE+DL Combination 30

3.1 Bipedal Walker 36
3.2 Tanh 40
3.3 Evolutionary Operators NEAT 41
3.4 Genotype to Phenotype 42
3.5 DL Architecture 45

4.1 Bipedalwalker Frame 49
4.2 Sparse Reward Function 50
4.3 Gaussian Filter example 55

5.1 Experiments Overview 59
5.2 RL - State and Terrain - Non-sparse Reward 60
5.3 NEAT - State and Terrain - Non-sparse 61
5.4 RL - State - Non-sparse Reward 62
5.5 NEAT - State - Non-sparse Reward 63
5.6 RL - Image - Non-sparse Reward 64
5.7 NEAT/RL - Image Down-sample - Non-sparse Reward . . . 65
5.8 RL/NEAT + DL - Image - Non-sparse Reward 66

5.9 RL - State and Terrain - Sparse Reward 67
5.10 NEAT - State and Terrain - Sparse Reward 68
5.11 RL - State - Sparse Reward 69
5.12 NEAT - State - Sparse Reward 70
5.13 NEAT+DL and SAC - Image - Sparse Reward 71

List of Tables

2.1 Input/Output NE Summary 32
2.2 RL Environments 33

3.1 Bipedal Sensors 36

4.1 Sparse Fitness Table 50
4.2 Fitness and Threshold 54

Acknowledgement

First and foremost, I would like to thank my main supervisor, doctor Tønnes Frostad Nygaard for assisting me at every stage of this research project, with continuous support and guidance and invaluable weekly meetings with productive conversations. Special thanks to my co-supervisor, associate professor Kai Olav Ellefsen, for sharing important and essential knowledge in the field and with many helpful and valuable contributions.

Thanks to friend and fellow student Mathias Kvamme Nilsen for motivating, helping, and pushing each other in this endeavour, but with many daily distractions, mainly through quizzes. I would also like to thank the Robotics and Intelligent Systems research group and its staff for making a great learning environment the past two years.

Finally, I would like to thank my friends, family, and especially my girlfriend Aisa Meisler for supporting me through all of this.

Chapter 1

Introduction

1.1 Motivation

Recent years have seen an impressive and rapid progression in solving highly challenging tasks that have challenged AI researchers for decades. One of the fields in machine learning which has shown particularly impressive results is deep learning. With deep learning, computers have mastered complex tasks from a wide range of domains, from image and speech recognition to game playing, robot control, and systems for self-driving cars.

Deep Learning (DL) is particularly good at extracting structures from large amounts of data, forming meaningful, compressed internal representations from high-dimensional inputs. For instance, DL can learn from a large dataset of pictures what types of features are characteristic of a dog or a car. DL has achieved state-of-the-art performance in image recognition [12, 32, 57] and speech recognition [25, 37, 45]. However, deep learning is not without its weaknesses, and many open challenges restrict the ability of deep learning algorithms to solve certain kinds of problems, and the complexity of these models limits our capability to understand and trust the decisions made by deep learning [35]. One of the challenges is solving Reinforcement Learning (RL) problems with sparse rewards [43]. In RL environments, the environment provides the algorithm with a numerical reward, which indicates whether the action it took was good or bad. In more complex, sparse environments, this reward is delayed, meaning that the algorithm has to do several "correct" actions in sequence before receiving feedback from the environment. By introducing other machine learning techniques with different strengths and functionality, it is possible to assist deep learning in overcoming some of these open challenges. An interesting idea, which has not yet been widely explored, is, therefore, to combine neuroevolution (NE) and deep learning (DL).

Neuroevolution (NE) is a sub-field of Evolutionary Algorithms (EA) and is a type of population-based search algorithm, where a population of neural networks solving a problem is maintained, which allows for a more diverse search for candidate solutions [54]. Through some of the attributes inherited by EAs, NE has the potential to overcome some of DL limitations by exploring multiple solutions simultaneously and solving sparse reward problems. In contrast to RL and DL methods that heavily rely on continuous rewards and feedback from the environment to improve and learn, NE only cares about the accumulated reward over an entire episode, allowing it to overcome sparse reward problems. Evolution is used to adjust the population in NE, which is computationally expensive. This is the reason why, as with most evolutionary algorithms, NE struggles to improve effectively with high-dimensional data [1].

A promising way to combine NE and DL is to let deep learning do what it excels at, learn and make predictions from a large amount of high-dimensional data, and train a small action-selection component using NE. [1, 11, 17, 29, 43]. This thesis will explore ways to combine deep learning and neuroevolution, aiming to solve problems that neither could solve well by themselves by uniting their advantages.

1.2 Goal of the thesis

The overall goal is to get an overview and to thoroughly explore the combination of neuroevolution and deep learning.

Goal 1 : Surveying the related work of neuroevolution in combination with deep learning.

The first goal of the thesis is to survey the use of neuroevolution and deep learning combined. Exploring the variations in the different papers, focusing on the different algorithms, the division of work between the algorithms, representations, and problems solved using this combination. To the best of the author's knowledge, a thorough survey has not been done before in this particular sub-field. Hopefully, this can give a good insight into this field and reveal potential new challenges and possibilities utilizing this combination.

Goal 2 : Investigate the performance of a deep learning and neuroevolution combination in a high-dimensional, sparse reward environment.

This goal was chosen to get a better understanding of a DL+NE combination and its capabilities. DL manages and extracts useful information from the high-dimensional data, and NE chooses the best actions based on this information, solving sparse reward problems. The problem is high-dimensional, mimicking data received in the real world, and with most real-life tasks being sparse. This problem is highly interesting, testing the capabilities of DL+NE and helping to close the gap between solving problems in simplistic environments with continuous feedback to highly complex, noisy, uncertain real-world environments.

Milestones

The second goal is divided into separate milestones, leading up to the final experiment that is a high-dimensional and sparse reward problem.

Milestone 1 : Compare the performance of deep RL and NE algorithms in a simple, non-sparse environment.

Milestone 2 : Investigate and possibly verify how a combination with deep learning and the algorithms can successfully work in the simple, non-sparse environment.

Milestone 3 : Test the algorithm's performance with high-dimensional input data.

Milestone 4 : Test the algorithm's performance with sparse reward.

The first milestone is comparing the performance of the chosen RL and NE algorithm in a simple, non-sparse environment. Secondly, finding a way to combine DL with these algorithms and test whether such a combination will work. The third milestone investigates the performance when the input to the algorithms increases in dimensionality, together with the original normal fitness function. The last milestone is to change the reward function from a normal fitness function to a sparse fitness function and test the performance of the algorithms. In this experiment, the input dimensionality is low, so that the main focus is the performance with the new fitness function. With all these milestones completed, the second goal of this thesis should be achievable.

1.3 Outline

Chapter 2 introduces the theory used in this thesis and related work in the field. Chapter 3 gives information about the environments and algorithms used. Chapter 4 describes the author's changes and contributions to the already described algorithms and environments, along with experimental setup and challenges.

Chapter 5 presents the experiments as well as the results. Chapters 6 and 7 include discussion and the conclusion of the thesis and lastly chapter 8 presents possible avenues for future work.

Chapter 2

Background

This chapter aims to give an introduction to the field of machine learning as well as more in-depth knowledge about thesis-related background information in the fields of deep learning, reinforcement learning, and evolutionary algorithms. The last section of this chapter presents a survey of the related literature in the sub-field of neuroevolution in combination with deep learning, with comprehensive information about the algorithms, problems, and contributions.

2.1 Machine Learning

Machine learning is a part of computer science where the objective and goal is to construct algorithms that can improve automatically through experience/examples. The field of machine learning is not a new idea or field; it has been around since the 1940s, with the popularity decreasing and increasing over the last decades. However, in recent years, popularity has exploded, and companies and researchers employ it to all types of problems and disciplines. The reason why it has seen such an increase in popularity is directly linked with the rapid progression and availability in processing power, which furthermore allows more complex and challenging problems to be solved in a reasonable amount of time. Machine learning is a multidisciplinary field with aspects from statistics, mathematics, biology, and computer science and is most closely related to computational statistics. Machine learning is a sub-part of the greater field of Artificial Intelligence and is usually divided into three sub-fields, (1) Supervised Learning, (2) Unsupervised Learning, and (3) Reinforcement Learning.

2.1.1 Types of Machine Learning

2.1.1.1 Supervised Learning

The majority of practical usage in today's society, without including the scientific research field, uses a supervised learning approach [33]. This approach aims to learn the mapping function f from the input x to the

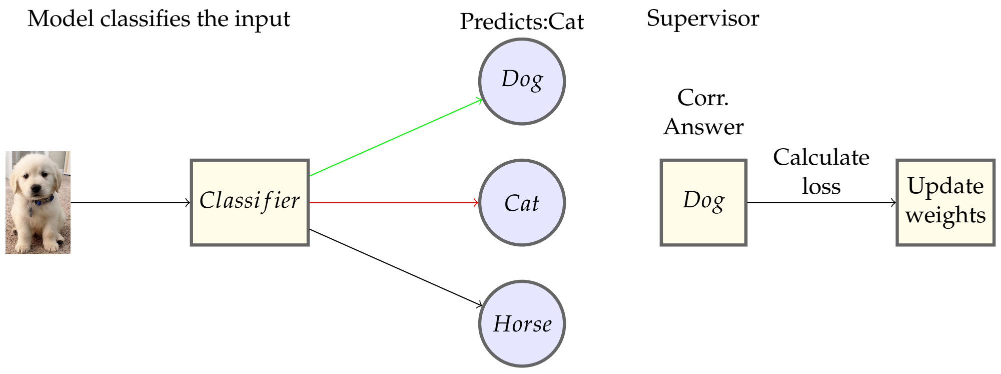


Figure 2.1: Example of supervised learning

output Y, seen in equation 2.1. In other words, it receives an input x and attempts to find the correct features or information that links it with the output Y.

$$f(x) = Y \tag{2.1}$$

The algorithm needs to approximate this mapping function well so that when presented with new input data x, it manages to predict the correct output Y. It is called supervised because the process of learning from a dataset can be seen as a teacher supervising a student. The "supervisor" provides the input data and the correct output data through the dataset. The algorithm is given an input, and based on this input, predicts an output. It is then given the correct answer and has to adjust its mapping function according to how well or close its prediction was to the actual correct answer, depicted in figure 2.1. If the model was wrong and adjusts its weight accordingly, the goal is that it manages to predict the correct answer when presented with a similar input in the future.

Supervised learning can be divided into two different problems:

1. Classification - Output value is a category, such as "Disease" and "no disease," or "cat" and "dog." The number of categories can range from one to infinite.

2. Regression - Output value is usually a real numbered value. Examples include Price, Weight.

When training a machine learning model, and especially through the use of a dataset, a common problem is over and underfitting. This can also relate to other machine learning problems often described by other terms. These two terms relate to how well the model can generalize to new input and the model's performance with the supplied dataset. Over-fitting is a problem where the model is able to perform very well with the given training dataset, but when presented with new unseen data, it struggles to predict correctly. This is because it has learned the noise in the training data and not the underlying effect. Under-fitting is when the performance is low with the training dataset as well as with new unseen data, this can be caused by lack of data, too little training, or if the machine learning model is too simple. The goal is to train the model to hit the optimum place between over and underfitting, good performance on the training dataset, and good performance with new unseen data. This is not easy to achieve, but a common practice for accomplishing this is dividing the dataset into three separate parts, training set, validation set, and test set. The training set is used when training the model, the validation set is used to validate the model throughout the training, and finally, the goal of the test set is to see how well it generalizes to new data by using never-before-seen data.

2.1.1.2 Unsupervised Learning

Unsupervised learning is utilized when there is only input data x and no correct output variables, in other words, an unlabeled dataset. The goal is to let the algorithm find the underlying structure or distribution in the data. These are called unsupervised because no teacher is telling the algorithm what is right and wrong, and it is up to the algorithm itself to find meaningful and interesting structure in the data.

Unsupervised learning can be divided into several different problems. The most common are:

1. Clustering - The algorithm divides the data with the same structure into clusters, in other words, grouping similar data points. "Clustering people who buy the same things."
2. Association - Finding structures or rules that govern a large part of the data, "People who buy X, also buys Y."

Unsupervised learning can typically be used before supervised learning when performing exploratory analysis on the data, identifying features in the data. Clustering algorithms can also be used to compress data [58].

Semi-supervised Learning

Semi-supervised learning is a combination of unsupervised learning and supervised learning, where a partially labeled dataset is used. The goal of this combination is to see how combining different types of labeled and unlabeled data can change the learning process and design algorithms that can take advantage of this combination [65]. This is used when the labeled data is scarce and expensive.

2.1.1.3 Reinforcement Learning

In reinforcement learning, an *agent* is learning how to take the correct actions in an environment with which it can interact with. The environment provides the agent with a reward and a new state based on the action taken by the agent. In this type of machine learning, no supervisor tells the agent what to do but instead guiding it through rewards provided by the environment. The agent's goal is to maximize its reward through actions in the environment. There might not be an optimal action to take in these types of problems, which is one of the reasons why it can not be solved by supervised learning. The figure below 2.2 shows how a typical reinforcement learning algorithm operates, where there is a state, the agent chooses an action based on the state, this results in a new state, and the agent is given a reward from the environment.

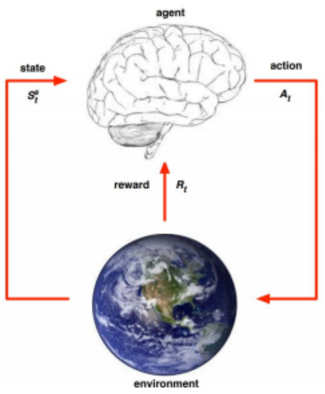


Figure 2.2: Reinforcement Learning iteration, State, agent, action, reward and environment [47]

Markov Decision Process

Reinforcement learning problems are often modeled as Markov Decision Processes (MDP). A key aspect in MDPs is the *Markov Property*. "If an environment has the Markov Property, then its one-step dynamics enable us to predict the next state and expected next reward given the current state and action [56]." This property is essential in RL because this allows the agent to make decisions and actions based solely on the current state the agent is in. An RL task that satisfies this Markov Property is called an MDP. If the environment has a finite set of states and actions, it is called a Finite Markov Decision Process (FMDP). An MDP models the RL environment and contains (1) A set of possible world states s, (2) A set of possible actions A, (3) Reward function R, and (4) a description T of each action's effect in each state.

Q-learning

The goal in Reinforcement learning is to solve the MDP, which involves finding the best actions to take under what circumstances to accumulate the highest expected total reward in all successive steps. Q-learning is an algorithm that attempts to find the optimal policy to solve an MDP, and this is done by learning a Q-function, given in equation 2.2. A policy in reinforcement learning is a term describing the agents strategy to pursuit its goal.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{2.2}$$

This is an action-value function that tells how good it is to take this action, and this function directly approximates the optimal action-value function Q^*, independent of the policy being followed [56]. The actions taken by the q-learning agent are given in equation 2.3

$$a(s) = \text{argmax} Q_\theta(s, a) \tag{2.3}$$

Policy Gradient

Policy Gradient is another method used for solving MDPs. The goal of the policy gradient method is to update θ to values that make π_θ the optimal policy, where π is the policy and θ is the parameters of the policy. The The algorithm aims to maximize the expected return, seen in equation 2.4.

$$J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{E}[R(\tau)] \tag{2.4}$$

Model-based agent vs. Model-free agent

Model-based agent vs. Model-free agent is a distinction made in RL algorithms, and this can relate to how an agent chooses its action. Model-free agents take their actions based on the stimuli received, and the actions are done almost automatically. Model-based agents are more goal-directed, and the agent knows the value of the goals and the relationship between actions and consequence [56].

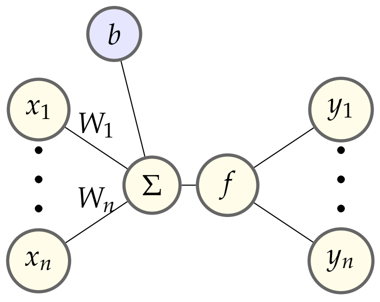
Getting enough information in model-based algorithms is challenging in more complex, high-dimensional environments. An inaccurate model of the environment almost always leads to sub-optimal actions. Therefore model-free-based algorithms have achieved the most success in those environments [11]. The goal of model-based algorithms is to solve mainly two challenges in model-free algorithms,(1) the requirement of a vast amount of training data and (2) transfer the policy to different tasks in the same environment.

Off-policy vs On-policy

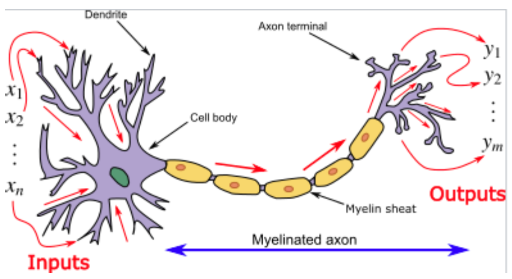
This term explains how the agent updates its policy. On-policy is dependant on the current policy being used, while off-policy works independently of the current policy. In other words, on-policy updates its policy according to how the current policy performs, while off-policy gathers information from past policies, often using a replay buffer. This allows the agent to know the latest experiences in addition to older experiences that may have been collected using an older policy and improves the policy according to this buffer.

2.1.2 Neural Networks

Neural networks (NN) or Artificial Neural Networks (ANN) are some of the most commonly used architectures in machine learning [33]. This architecture is inspired by the inner workings of the human brain, more precisely, the neurons and connections between them. Figure 2.3 depicts the relationship between a biological neuron and an artificial neuron. The goal is to mimic how the brain works but vastly simplified in complexity. The advance in computing power allows us to increase the number of connections and neurons in a NN progressively toward the hundred billion neurons in the human brain. NN has both served as a way to better understand how the human brain works and as a basis for efforts creating artificial intelligence. NN can be used for multiple different applications, but it tends to fall into one of these categories: (1) function approximating, (2) classification, or (3) data processing.



(a) Artificial Neuron: Inputs:x_1 to x_n, Outputs:y_1 to y_n, Activation Function:f Bias:b



(b) Biological Simple Neuron: Inputs via the Dendrites, Outputs through Axon [61]

Figure 2.3: Artificial and Biological Neurons

Functionality

Neural network is a network of connections and neurons; each neuron receives a signal, does some processing, and then sends the signal to all neurons connected to it. The signal that goes "through" a connection is usually a real number. The output of a neuron is calculated by some non-linear function of the sum of its input. The connections are called edges, and the edges have a weight attached to it so that it can strengthen or weaken the signal; this weight is adjusted when the network is "learning." The edges are updated using gradient descent, more thoroughly explained in the sub-section Gradient Descent (2.1.2). This is an iterative optimization technique for finding a local minimum. The bias term is used to move the activation function to the left or right on the graph. An example of a complete neural network is illustrated in figure 2.4.

Layers

A neural network usually consists of three types of layers, an input layer, an output layer, and 0 to n number of hidden layers. The input layer receives input data, this is usually a real number, and depending on the type of data or classifier, these numbers represent different objects, such as the pixel value in an image classifier or the number of bedrooms in a house-price predictor. The hidden layers are located between the input and output layers and are responsible for doing non-linear transformations of the inputs. The output layer is responsible for converting the values from the hidden layer to object-specific outputs, such as classes in a classifier or a real value number in a regression problem.

Activation Function

The activation function is responsible for deciding if the neurons should propagate its value to the next layer. The most straightforward example of an activation function is a threshold function; if $Y >$ threshold, signal next layer, else send 0 to the next layer. The activation functions are usually chosen based on the architecture and type of classification.

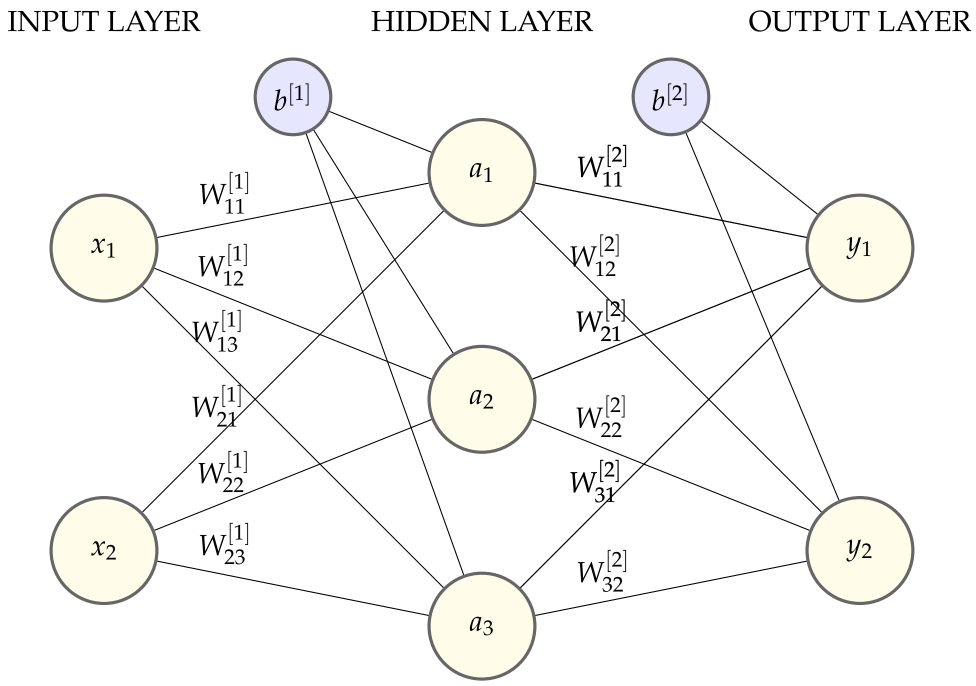


Figure 2.4: Neural Network, with 2 inputs, 3 hidden layer neurons, 2 biases and 2 output neurons

Calculations

To calculate the output of the nodes a_k in the hidden layer, the sum of the nodes is calculated in the last layer, multiplied with the weights in the edges, and add the bias term. This result is then passed through a non-linear activation function g, seen in equation 2.5.

$$a_k = g(\sum_{j=1}^{n} x_j W_{jk} + b^{[1]}) \tag{2.5}$$

And correspondingly the output neurons, seen in equation 2.6.

$$y_k = g(\sum_{j=1}^{n} a_j W_{jk} + b^{[2]}) \tag{2.6}$$

After the outputs are calculated, the correct answer is provided to the algorithm if it is supervised learning, and the algorithm uses an error function to calculate how far off the answer was in relation to the correct answer. This error value is used to iteratively update the weights by using an optimization technique, layer by layer, towards the input nodes; this is called backpropagation.

Loss Function

Loss functions are used as an evaluation method for how well the model fits the dataset, and it is the basis for iteratively improving the model by reducing the error.

Gradient Descent

Gradient descent is one of the most commonly used optimization techniques and is used to minimize a function by finding the direction of the steepest descent. This is done by taking the negative of the gradient. Each time the gradient is calculated and the weights and biases are updated, the neural network is hopefully but not necessarily a step closer to the optimum value.

Summary

NN consists of neurons, connections, and layers. Through these quite simple building blocks, it can learn how to approximate functions, doing simple classification and regression. This architecture can be extended by increasing the depth, which is done by adding more hidden layers. This allows the NN to learn more complex functions and vastly increases the possibilities of a NN.

2.1.3 Deep Learning

Deep learning is a sub-field of machine learning that has seen an explosion in popularity in recent years, and this is essentially due to the technological advancements that only increase in rate each year. With faster and more computational power available, a highly complex model is more feasible. GPUs, in particular, is a technology that allows a significant speed-up when training deep learning models due to the way deep learning models are implemented by using matrices. Deep learning is typically implemented like a NN, but the numbers of neurons, layers and parameters are vastly increased. These architectures may look interchangeable on the surface, but with this increase in complexity, the attributes and functionality of the model change. A simple NN is transmitting the input data through the model and producing an output. DL, on the other hand, with the increase in depth and number of neurons, is able to extract complex features information automatically from the input data and form a relationship between input stimuli and neural responses that is also found in the human brain.

Deep learning has solved some of the most challenging problems in machine learning that have been unsolvable by other methods [33]. Some examples of its achievements is state-of-the-art image recognition [12, 32, 57] and speech recognition [25, 37, 45]. One of the most useful attributes of a deeper neural network is its ability to extract features directly from raw data, without pre-processing, such as images or audio files. Conventional machine learning techniques [4] on the other hand, has limited capability of extracting useful information from raw data, and doing so required considerable domain expertise to design a feature extractor that can transform the data before this data could be used in a classifier to detect patterns [4, 33]. Deep learning models are able to do all this by themselves, receiving raw data, propagating the data and increasing the abstraction through the layers, and extracting the useful representations that are needed for classification [33]. The difference between how a deep learning model works compared to conventional machine learning is showed in figure 2.5. Therefore deep learning can also be referred to as a representation learning algorithm due to the ability to learn how it can extract features/representations. Deep learning is a varied field that encapsulates all models that take advantage of the depth and complexity explained here. Examples of this are Deep Reinforcement learning, which is able to solve more complex MDP's and Deep Convolutions Networks that are specialized for image classification.

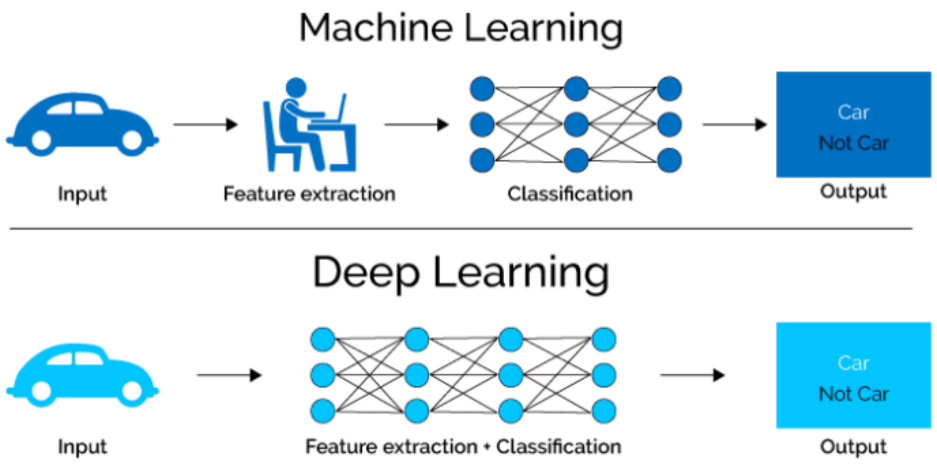


Figure 2.5: Conventional ML techniques vs. Deep Learning [34]

To better understand how deep learning can extract useful features by increasing its abstraction level through the network, imagine a face classifier, like the one depicted in figure 2.6. The classifier aims to receive raw data in the form of pixel values from images and classify the persons in the image. The first hidden layer has learned to look for edges, and the second hidden layer has learned to look for arrangements of edges, here in the form of facial features, such as noses, eyes, eyebrows, and mouths. The third hidden layer has learned to look for full shapes, in this case, entire faces. Here the abstraction layer increases through the network, which can explain what is going on inside this model and how it can make predictions; these abstraction levels are also visualized in figure 2.9. This example also describes one of the most desirable properties of deep learning, that the network itself learns what attributes and features to look for in the input data when performing classification, without the need for human interaction.

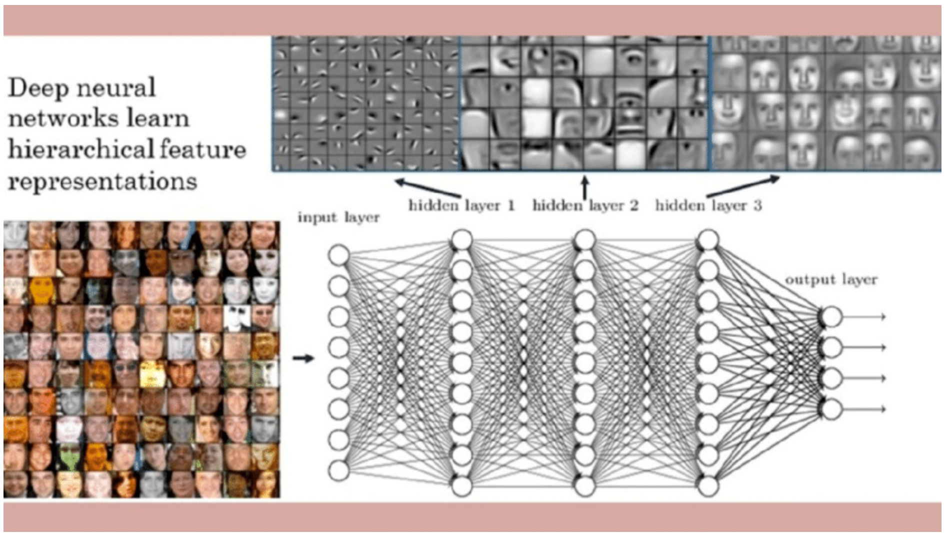


Figure 2.6: Face-classifier using Deep Neural Networks [33]

2.1.3.1 Convolutional Neural Network

Convolutional Neural networks or CNN is used heavily in image recognition applications and is directly inspired by visual neuroscience [33]. CNN is dominating the fields in computer vision for almost all recognition, and detection tasks [33]. It is very similar to a NN because it is made up of neurons with learnable weights and biases. The difference in CNN is the assumption that the inputs are images, and with this follows the option to encode certain properties into the architecture. The architecture in a CNN consists of three main components; convolutional layers, pooling layers, and fully connected layers. An example of a convolutional neural network can be seen in figure 2.7, with convolutional layers (blue), max-pooling layers (yellow), and the last three layers are a fully connected neural network architecture.

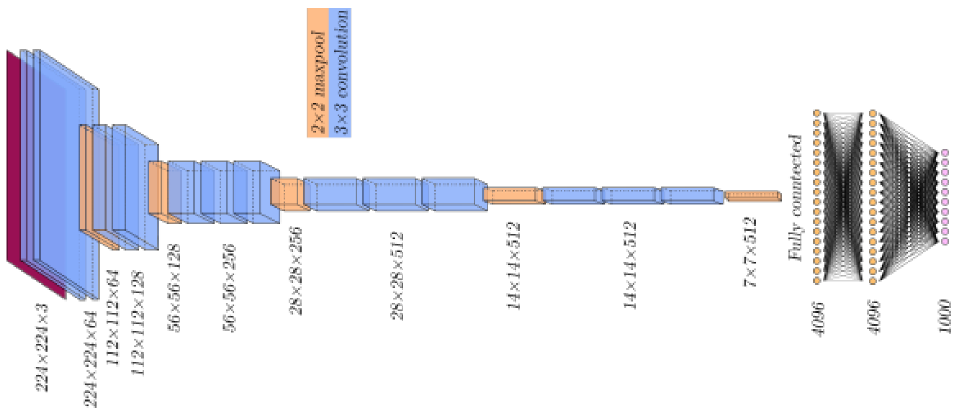


Figure 2.7: VGG16 CNN architecture [50]

Convolutional Layer

The convolutional layer, portrayed in figure 2.8, consists of multiple learnable filters; the filters are usually small (5x5x3) and extend to the full depth of the input volume. In the forward pass, the filter is convoluted along the entirety of the input image. The dot product is calculated between the filter and the input at every position. Each dot product is stored in what is called an activation map. This activation map is related to each neuron's receptive field, in other words, what part of the image each neuron looks at. Each layer can have multiple filters, and correspondingly several activation maps, and as the learning progresses, each filter learns to look for different objects in the image that will activate the filter. Backpropagation is as simple as in a regular deep neural network, updating the weights in the filters. The abstraction level increases with the depth of the network, so the filters look for more advanced shapes, visualized in figure 2.9.

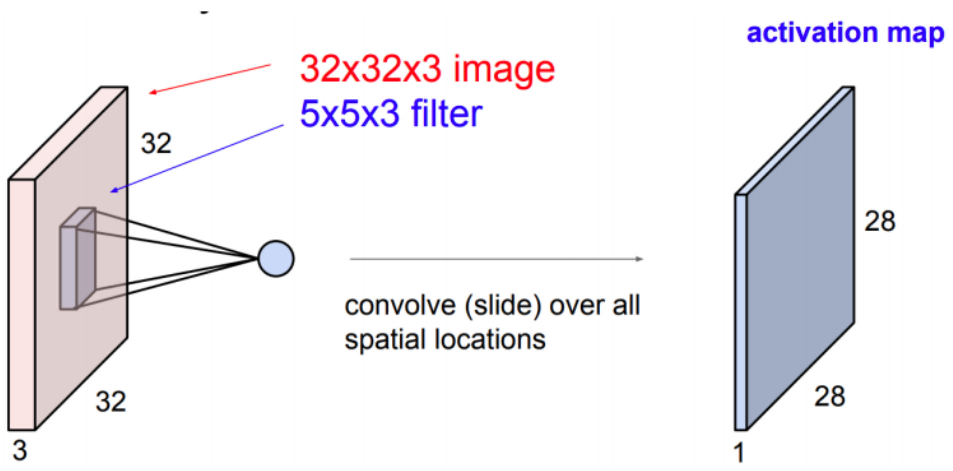


Figure 2.8: Convolutional layer, with a 32x32x3 input image, one 5x5x3 filter, that results in one activation map [26].

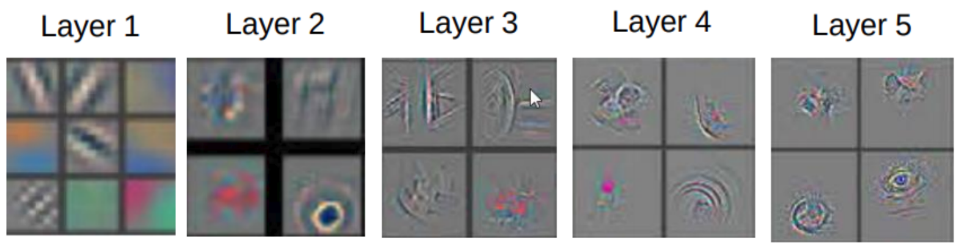


Figure 2.9: Filter abstraction in CNN [64]

Pooling Layers

The pooling layers, illustrated in figure 2.10, are responsible for downsizing the activation map. There is a couple of reason why this is useful, (1) the number of parameters decreases with a smaller activation map and (2) decrease the location sensitivity of features in an image, which results in activation maps being more robust against the changes in the position of the features in the image, called local translation invariance. Common pooling layers are max-pooling and average-pooling. Most CNN architectures include pooling layers, but some researchers wish to avoid pooling operations and have found ways to eliminate these layers by substituting them with convolutional layers [51].

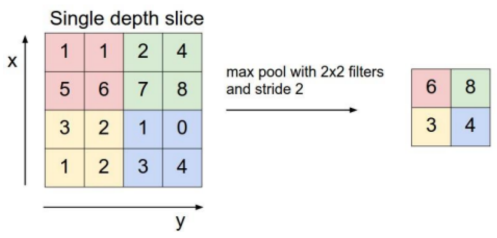


Figure 2.10: Max-pooling, with stride = 2 (stride is the spatial step length in the convolutional operation) [26]

Fully Connected Layers

Fully connected layers are a standard fully connected NN, receiving features produced by convolutional layers and pooling layers. This layer is responsible for transforming the received features into useful and problem-specific outputs. An example of this could be a classifier, where the last layer is responsible for transforming the features into distinct classes.

Dropout Layers

Dropout layers is a concept introduced to prevent overfitting and help the model generalize better; this can also be viewed as requiring each neuron to "learn" more. The dropout layer works by introducing a dropout rate, which is the percentage chance of an input to a neuron set to 0. So if the dropout rate is 0.5, there is a 50% chance that the input to each neuron is set to 0. The actual sum of the inputs remains unchanged because it also scales up all other non 0 inputs with a scale fitting with the $1/(1 - rate)$.

2.1.3.2 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) is the combination of deep learning and reinforcement learning. This combination aims to create algorithms that can achieve their task in more complex environments. Reinforcement learning has achieved some success in various domains [38], but these domains are handcrafted, low-dimensional, or fully observed state spaces. The problem is getting enough information about the environment so that the agent can learn to take the correct actions. The motivation behind DRL is that deep learning can extract useful features from high dimensional input, and an RL agent can react to this compressed representation of the environment.

DRL was first introduced in 2013 in Mnih et al [39] and later extended [38]. In these papers, methods are presented using a deep CNN trained using Q-learning, end-to-end, reinforcement learning. The goal is to derive an efficient representation of the environment from high-dimensional data so that the agent can utilize these observations to generalize past experiences to new situations. The deep convolutional neural network is responsible for extracting useful features from the raw pixel data and approximate an optimal action-value function. These methods were able to surpass the performance of all earlier algorithms and achieve human-level performance in the 49 Atari games they tested on and were using the same architectures and hyperparameter in every game, which shows that their approach is very robust. However, the Q-learning algorithm has a problem with overestimating the bias (overestimates action value under certain conditions). Hasselt, Guez, and Silver [21] improved and extended the approach and addressed the overestimated biases using the double Q-learning algorithm [20]. Other improvements of the Q-learning algorithms have also been shown to produce good results.

In Hessel et al. [22] a combination of six of these improvements are integrated into a single algorithm which achieves state-of-the-art performance in Atari games.

DRL is not without its problem and drawbacks, one being that they require vast amounts of data to work well. Another problem is that they are quite slow learners, and the reason for that is that it needs to learn how to extract features combined with learning a policy. These problems have been addressed by introducing transfer learning, where the feature extractor is trained in a supervised fashion by watching human play games [7, 23, 49]. This makes the agent perform well from the start of learning and can continue improving.

The combination of RL and DL has achieved some great results in more advanced games like Go [49], Chess [48] and Shogi [48].

2.1.3.3 Challenges in Deep Learning

Exploration vs. Exploitation

One of the biggest challenges in machine learning, in general, is the trade-off between exploration and exploitation [42, 60]. Exploration is where the algorithm explores new solutions, and this may lead to finding more optimal solutions and can lead to a way out of a local optimum. On the other hand, the algorithm also needs to exploit already known solutions, optimizing them to see their full potential. So it is a trade-off between doing what it knows gives it reward and works and try to find new and better things that can accumulate a higher reward.

A classical exploration-exploitation problem is the Multi-armed bandit problem. Where there is k number of one-armed bandits(slot machines), the goal is to find out in which sequence to play the machines to accumulate the highest reward. Each machine provides a random reward from a probability distribution specific to that machine. The trade-off is exploiting the machine that has the highest expected payoff while exploring the other machines to get information about their expected payoffs.

Sparse rewards

A sparse reward is when an algorithm has to do many "correct" actions in sequence before getting a reward for doing so. Most deep learning algorithms still struggles with this [43], although some researchers have found ways to increase exploration in these algorithms to find solutions [9].

An example of the difference between non-sparse reward and sparse reward environment can be explained using the hotter, colder game. Where a hider has hidden something, and a searcher is trying to find the lost object. The searcher is given instructions by the hider, with cold being further from the object and warmer being closer. A non-sparse version of this can be that the searcher takes one step, then gets feedback from the hider on whether the step/action resulted in getting closer to the object. A sparse version can be that it requires the searcher to take 10-15 steps before the searcher gets feedback whether it resulted in getting closer to the object than in the beginning. This makes the problem increasingly complex, the searcher do not know which specific actions that resulted in getting closer or farther away from the object. In order to solve this game, the searcher can not be fully dependant on the feedback, and have to balance between exploring and exploiting.

Data driven

Deep learning requires large amounts of data to learn even the easiest abstraction relationships and to do simple classification tasks [35]. With this requirement, it is not difficult to see why these models have been increasingly popular and well-performing in the last decade, with data being more available than ever.

Black box

Deep learning has millions or even billions of parameters, and it is often useful to know how these systems work; of now, it is still mostly considered a "black box." [35] However, researchers have taken steps to get a better understanding of these systems and displaying what each neuron has learned to detect [41].

Transfer learning

Transfer learning is a problem in deep learning. This means that the model might struggle to perform or solve the problem if the goal or environment is slightly changed from the original goal or training environment. With this being a problem, it might seem that the solutions given by the DL model are extremely superficial, and the question is: what has it actually learned [35].

Images for DRL

High-dimensionality remains one of the critical challenges for DRL to explore in high-dimensional environments effectively [42]. This especially relates to pixel-to-action mapping. However, there have been proposed solutions to this, which do not require any changes in the actual algorithms. This solution uses data augmentation techniques on the input images which resulted in state-of-the-art performance over several benchmarks [28].

Challenge

Deep learning has achieved some great results doing what it is good at, but it is also useful to be aware of problems where DL does not perform well, like problems with sparse reward. Instead of tweaking and changing DL to fit every problem, a combination with other approaches might be a better solution. By picking algorithms that excel at different parts of a problem, a better solution can be achieved.

2.2 Evolutionary Computation

Evolutionary Computation (EC) is an area of research within computer science. This field draws its main inspiration from natural evolutionary processes. The fundamentals of this field relate to the powerful components of evolution and the usage of trial and error to solve problems. Evolutionary Algorithms (EA) is the sub-field describing the algorithms used for solving problems using evolution. Genetic programming, a sub-class of the larger class of EAs, has produced a wide range of competitive results to human-level achievements in many different fields, including controllers, game playing, algebra, and designing circuits [30].

2.2.1 Evolutionary Algorithm

There are several different variations of Evolutionary Algorithms, where goals, representation, and implementation differ. Common for all of these algorithms is the idea behind them: a population of individuals, where there is a natural selection to pressure the population to become better (survival of the fittest) [10]. In addition to achievements described in the previous paragraph, evolutionary algorithms have also achieved some surprising results in recent years, managing to perform competitively with DRL in high-dimensional Atari environments [46, 55]. EAs are highly parallelizable due to the population-based search, making it time-efficient (real-time) and able to create different strategies simultaneously [54]. EAs have shown to be good to solve problems with sparse rewards [1, 11, 27], and can be used alone or in combination to solve those problems such as an Evolutionary RL algorithm proposed in [27].

An EA consists of these main components: representation, fitness function, population, variation operators, parent and survivor selection, and a termination criterion. Concerning the workflow of a typical EA, these components are illustrated in figure 2.11.

Representation

Representation in EA's is more commonly called genotype. The genotype is a set of genes describing each individual, and the phenotype is what could be developed based on the genotype. How this representation is described is problem-dependant.

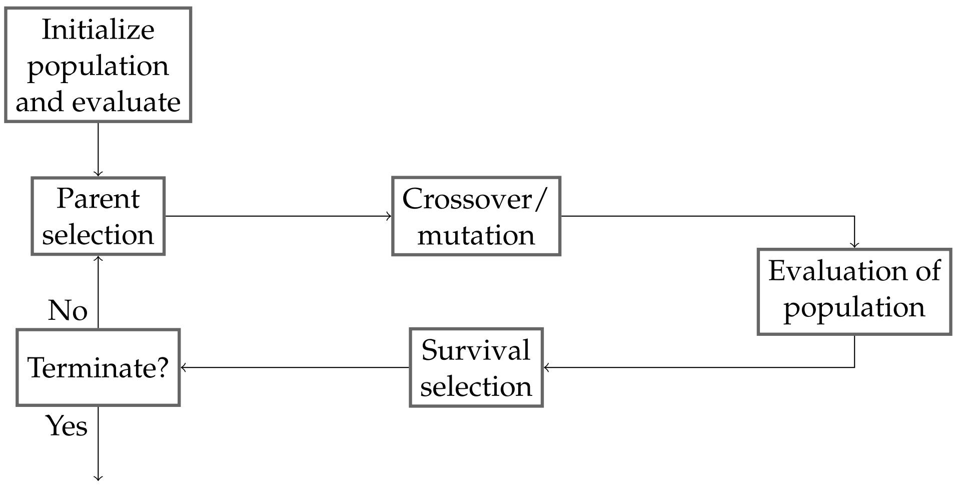


Figure 2.11: Standard layout of an Evolutionary Algorithm

Fitness Function

The fitness function forms the basis of selection in that it promotes improvement. The fitness function represents the problem that needs to be solved and defines what improvement is. This fitness value is given to individuals based on its ability to solve the problem and is further used in choosing which individuals to include in the next generation.

Population

The population is a set of candidate solutions to the problem. The initial population is usually randomly seeded or chosen from a set of known solutions. The population size can either be fixed or very unusual dynamic. There are two main ways for the population to evolve, evolving the entire population to create the next generation or, evolve individuals within the population, allowing individuals to be kept into the next generation. One generation refers to one run of the algorithm and also to the population at the beginning of that run.

Parent Selection

Parent Selection is the process of choosing which individuals that should be parents for the next generation. Together with survivor selection, this is responsible for pushing towards quality improvement (better fitness). There are different approaches to selecting which individuals should be parents; usually, the selection is probabilistic. The high-quality solutions have a higher chance of becoming parents, but the low-quality solutions are given a small chance to avoid local optimum and increase diversity.

Variation Operators

Crossover The offspring is created by combining the parents into one or two offsprings. What parts and how they are combined are dependent on random selection. The motivation behind crossover is that you are combining individuals with desirable features, and by crossover, combine these features. This operation is exploitative in that it makes a transition somewhere between the parents in the search space [14, 62].

Mutation The role of the mutation operator is to create small, random changes to a genotype. This operation is exploratory because it introduces a random change to the population [14, 62].

Survivor Selection

The individuals allowed to continue into the next generation are chosen based on fitness and quality. This selection is done after the offspring has been created, and this selection is often deterministic. The two most common ways to select is fitness based(selecting top quality solutions) or age-based(selecting only from offspring)

Termination Criteria

There are no guarantees in reaching an optimum in EA's, so there has to be other criteria to stop the algorithm. Examples of this could be time, the number of fitness evaluations if no fitness improvement or under a threshold, or where diversity drops below a threshold value.

Example

Image an example where the goal is to evolve a robot's morphology (shape, structure). The morphology consists of a body, legs, and arms, similar to a humanoid robot. The objective is to make an efficient morphology optimized for walking in dynamic environments. For the simplicity of this example, how it walks or is controlled will be left out.

The *representation* of each individual can be a vector describing its morphology, where the shape of the body is fixed as a cube, but the size can vary. The number of legs can be changed, and leg and arm lengths can also be varied.

$$Genotype = [nr\_legs, length\_legs, length\_arms, size\_body] \tag{2.7}$$

The *population* is randomly initialized, consisting of n number of robot individuals represented as the vectors in equation 2.7. The different robots are tested in the environment, and the $fitness$ of each robot is evaluated. In this case, the fitness function is how far the robot can walk without falling over. This fitness value is the foundation for the *parent selection*, where the top-performing robots have a higher chance of being selected as parents. The parents are then combined using *crossover* at a random position in the genome; elements of each parent are combined to create new offspring, as illustrated in figure 2.12. *Mutation* creates a small change in the offspring's genotype, which could be a small variation in the length of the leg, also displayed in figure 2.12. The new robots are evaluated and given a fitness score; based on this, the *survivors* are selected, the robots that should be saved in the new generation. In this stage, the best performing robots are chosen as survivors, and the other robots are discarded. Suppose the termination criteria are not met, select new parents, and continue the algorithm. The *termination criteria* in this case can be a threshold value; if a robot walks over a set distance, combined with a non-improvement threshold over a number of generations and a maximum number of allowed generations.

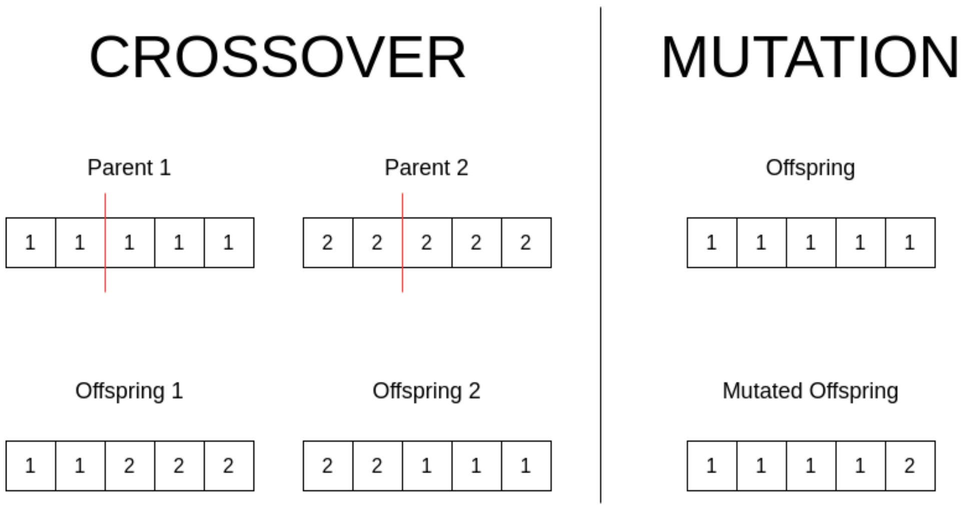


Figure 2.12: Variation Operators

General Problems in EA

Exploration vs. Exploitation Exploration versus exploitation is also a problem in EA's, where there has to be a balance to find good solutions, although some variations exist where they solely depend on exploration. In Genetic Algorithms (GA), excessive exploration will limit the diversity of solutions and induce premature convergence, and excessive exploitation will lead to a slow convergence [5]. Premature convergence is when an algorithm reaches a sub-optimal solution where the parent can not produce offspring that are better or superior to themselves—slow convergence results in a slow and time-consuming process.

High-dimensionality In complex RL environments or problems where there is high-dimensional data such as raw visual input, EAs struggles to improve effectively. The bigger the input, the bigger the network processing the data has to be, the larger the genome has to be, which equals slower evolution [1].

2.2.2 Neuroevolution

Neuroevolution is a sub-field of Evolutionary Computation, which employs EA's to optimize neural networks. The goal is to apply evolution to either evolve, both topology and weights, or just one of them. The population in neuroevolution can consist of neurons, weights, or full neural networks. Neuroevolution is known to perform well as a robot controller, with a long history of success in the field of evolutionary robotics [54].

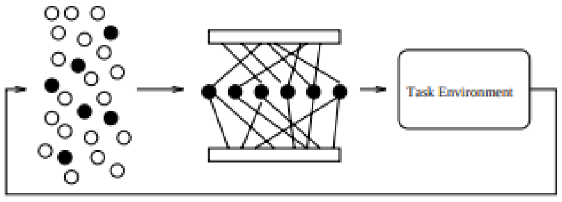
Evolution

Through the history of neuroevolution, several different approaches has been proposed on how to evolve a neural network[54]. From only evolving weights[15, 44], evolving the topology[63] or a combination TWEANN(Topology and Weight Evolving Artificial Neural Networks)[2, 40, 52, 53], to evolving neurons [15, 40] or evolving how to update the weights[13]. Some algorithms have a fixed topology, and some are randomizing through generations, keeping the ones that produces good results.

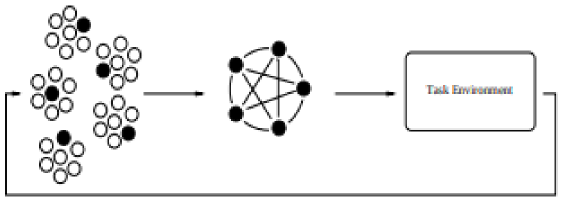
Types of NE-algorithms

There have been proposed several different algorithms for evolving neural networks. In [44] a fixed topology is applied, and the weights are evolved. This can be a good solution if this topology is known to be best for the given problem. SANE [40], showed in figure 2.13a, evolves the topology based on a population of nodes. The population consists of nodes of different types and specialization, and the main goal is to combine nodes that have optimized for different aspects of a NN. This creates a symbiotic relationship between the nodes. The goal of each neuron is to establish a connection with other neurons in the population to form a functioning NN. ESP [15], displayed in figure 2.13b, is an extension of SANE, with a population of nodes, but it enforces sub-population. This involves recombination that can only be done with other members of its own sub-population, and a neural network is created by randomly choosing nodes from each sub-population. This method used a fixed topology, only evolving weights, but if the algorithm did not manage to solve the task and got stuck, it restarted with a random number of hidden neurons between a preset range.

The Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [19] algorithm evolves its topology and weights by using a covariance matrix. A population of new individuals is generated by sampling a multivariate normal distribution. When all individuals are evaluated and given their fitness score, a mean of the multivariate normal distribution is calculated as a weighted average of the highest performing individuals in the population. The steps taken to update the covariance matrix are in the direction of previously successfully search steps. Other approaches



(a) Symbiotic, Adaptive Neuro-Evolution (SANE) [15]



(b) The Enforced Sub-Populations Method (ESP) [15]

Figure 2.13: SANE and ESP

evolving both topology and weights uses a population of full NN, like GNARL [2], NEAT [53] and HyperNEAT [52].

There are a few differences between these; GNARL initializes the population randomly, with a random number of hidden nodes. Half of the population is chosen as parents. The offspring is created by copying the parent and mutated according to the severity of how bad or good the fitness score was. NEAT is starting with only input and output nodes, no hidden nodes. Through mutation, which can add either a new connection between nodes or adding a new node, it gradually increases the size and complexity of the networks. This makes sure that it only evolves a suitable network for the task at hand and does not get an over complexified result. The special thing about this is the innovation number or historical markings that are used when doing crossover and mutation. HyperNEAT is exploiting geometry to evolve large-scale neural networks. HyperNEAT extends NEAT to evolve CPPN or Compositional Pattern Producing Networks. In this method, CPPN uses indirect encoding to encode NN. "The main idea is that we can generate a pattern of weights based on a function of the geometry of the inputs and outputs [54]." This makes it possible to have an encoded CPPN with only a few dozen weights to represent a NN with millions of connections. HyperNEAT was the first system to achieve direct pixel-to-action in Atari games [54].

NE and Sparse rewards

Neuroevolution inherits EA's ability to overcome delayed or sparse rewards by using a fitness metric that looks at the accumulated reward after an entire episode rather than continuous feedback after every action, often used by RL algorithms. Neuroevolution through the selection operators gradually pushes the population towards regions of the policy space that equals higher episodic returns. In addition, NE uses a population-based search that can effectively explore and exploit the action space, which leads to diverse policies that effectively explore the domain [27].

Computational Demanding

Neuroevolution being a part of EA, also struggles with high-dimensional data and is far more computationally demanding than gradient descent approaches like deep learning [11, 29]. Challenging RL environments or problems with raw visual data often require large networks for dealing with high-dimensional data. Up until now, approaches dealing with scaling NE to these problems have been focusing on indirect encoding, where small networks are mapped to large complex structure [29]. Other approaches involve using a pre-processor to transform high-dimensional raw data into low-dimensional features.

Summary

Evolutionary algorithms and Neuroevolution are powerful tools, especially as optimization techniques. The population-based search provides some desirable properties, such as parallelization and the capability to evolve several different strategies simultaneously. EAs and Neuroevolution work well in problems with sparse rewards and tasks where multiple viable solutions are desired. However, due to the computational demanding task of solving continuous RL environments, a way of scaling NE to this problem has to be found. This can be done in two approaches: reduce the dimensionality by compressing the representation of the neural network controller, or use a form of pre-processor like deep learning, to transform the high-dimensional raw data into low-dimensional features.

2.3 Deep Learning and Neuroevolution

DL and NE are both powerful tools when used on problems that can exploit their best qualities, but knowing their weaknesses is equally important when choosing an algorithm for a given task. DL is generally known for performing poorly in environments where the rewards are delayed, also known as sparse. On the other hand, NE performs poorly when dealing with high-dimensional data. The motivation behind combining DL and NE is based on their best qualities, and by picking algorithms that excel at different parts of a problem, a better solution can be achieved. One of DL's best qualities is extracting features directly from raw input, such as images or sound. NE is excellent at solving tasks with different strategies simultaneously and can solve tasks with sparse rewards. Thus combining DL and NE can produce a model capable of handling high-dimensional data, extracting useful information from complex environments, and a controller that can solve problems with sparse rewards. A few interesting questions need to be answered to thoroughly understand how these algorithms can be merged in a useful way.

2.3.1 Dividing the problem

It is first essential to determine how the workload should be distributed between the two algorithms. As mentioned earlier, the DL component should do most of the computational demanding tasks, dealing with the high-dimensional complex data from the environment. Exactly what type of work this DL component should do depends on whether it is model-free or model-based; in other words, a transformer or a model predictor. The NE component should receive a compact representation from the DL component and determine what types of actions are optimal based on the input. This workload division aims to maximize the potential of this union, utilizing the different strengths from each of the algorithms. Figure 2.14 depicts a simplified overview of how the problem could be divided amongst NE and DL.

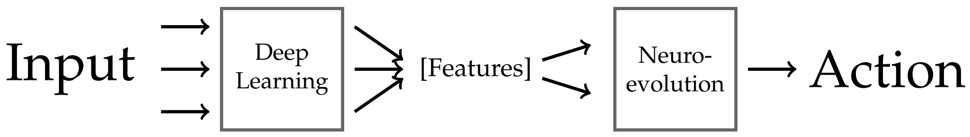


Figure 2.14: Work-load division

2.3.2 DL-component

The next step is specifying the DL component. It is first helpful to divide it into model-based and model-free. Then specify what type of problem the DL component is solving, which DL architecture, and if specified, why this architecture was chosen.

Model-Free

The goal for the DL component in model-free algorithms is to transform the high-dimensional data into low-dimensional compact features that the controller can use. Possibly one of the most intuitive approaches is using DCNN for video feature extraction, in that CNN is known to be good at extracting features from images, sound, and video. In Poulsen et al. [43] this approach was applied, showed in figure 2.15. Two different networks were implemented: shallow (6-layers) and deep network (12-layers). These networks were trained using a supervised learning approach. The results of the comparison showed that a deeper network is more robust and produced better results overall. In addition to this, they experimented with different types of feature representations to see which type of representations yielded the best results when combined with the NE component.

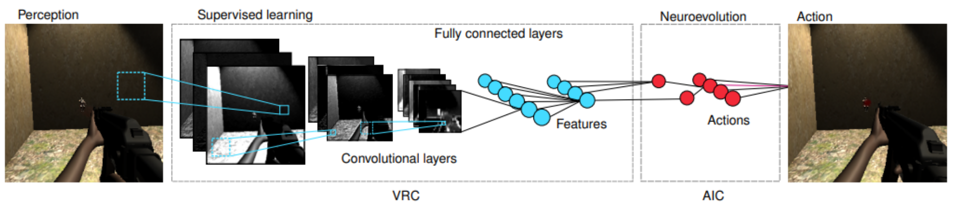


Figure 2.15: Combination of DL and NE [43]

In Koutnik et al. [29] a Max-Pooling CNN (MPCNN) architecture, displayed in figure 2.16, was used, with an algorithm called UL-ERL (Unsupervised Learning - Evolutionary Reinforcement Learning). This method combined evolved action learning with an unsupervised learning compressor. The compressor was evolved to maximize the variance in its output, and a fitness function forced the MPCNN to output feature vectors that are spread out in feature space. This results in a small set of useful features used by the small evolved RNN controller. In this approach, evolution is used for both feature extraction with the DL component and action selection by a small RNN controller. Instead of supervised learning, which requires task-specific domain knowledge, the DL component was evolved. It might not be an easy task to choose what constitutes a class in a given scenario in an RL setting. Instead of letting the algorithm figure this out on its own, not putting on any restraints which may be over-complex or not complex enough for the task at hand. This has its strengths and drawbacks; the strength is that there is no need to investigate the domain to create training sets. It evolves the needed architecture for that given problem, but evolving large structures is computational demanding, as mentioned before.

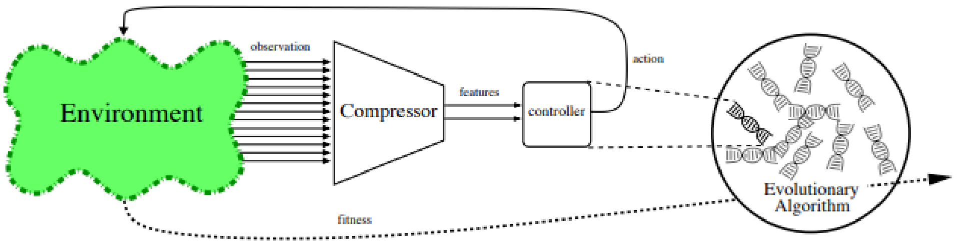


Figure 2.16: Unsupervised Learning – Evolutionary Reinforcement Learning (UL-ERL) framework [29]

Another good architecture at transforming high-dimensional data to low-dimensional compact representations is a Variational Autoencoder (VAE). A VAE consists of three components, encoder, chokepoint, and decoder, and the main goal is to compress data (encoder), create a compressed representation of the data (chokepoint), and decompress the data (decoder), shown in figure 2.17. The objective of the VAE is to reconstruct the input image as output. The training of a VAE is done by calculating the loss between the input image and output image of the model. Recreating the input from the compressed representation in the chokepoint shows that the compressed representation consists of enough information to represent the input accurately. In other words, no information is lost in the compression of the data, and the data in the chokepoint can be used to represent the input.

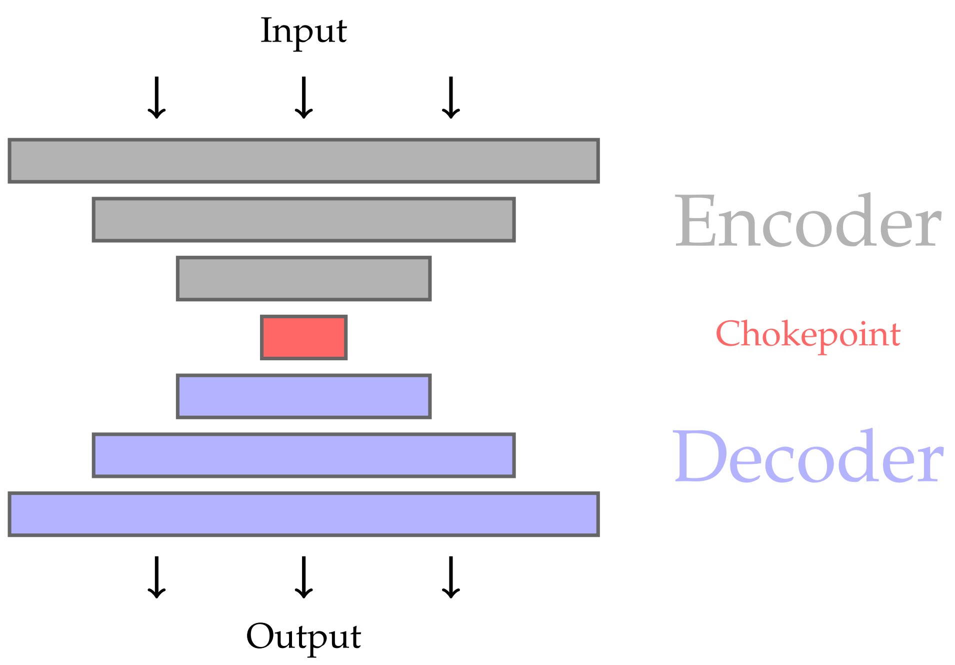


Figure 2.17: Simple variational autoencoder

Two relevant papers have proposed methods using VAE [1, 17]. One is a model-free approach, the other a combination. In Alvernaz et al. [1] a VAE was trained online during game-play and used backpropagation to update the weights, seen in figure 2.18. This method allows the VAE to continuously update its weight while playing; in the first 30 generations, exploration was rewarded high for the NE component (finding images that the VAE had not seen before) to acquire a diverse representation of the environment.

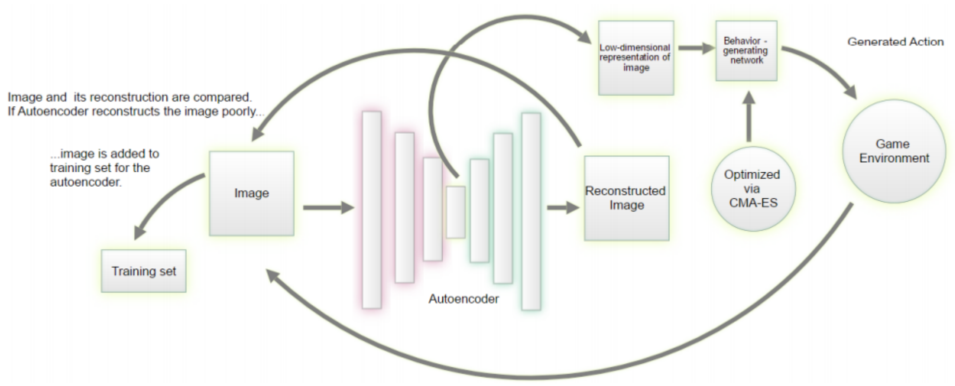
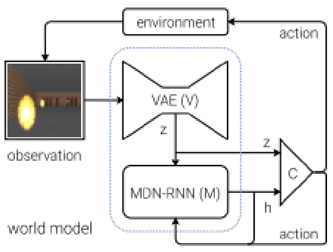

Low-dimensional representation of image
Behavior - generating network
Generated Action
Image and its reconstruction are compared. If Autoencoder reconstructs the image poorly...
...image is added to training set for the autoencoder.
Image
Reconstructed Image
Optimized via CMA-ES
Game Environment
Training set
Autoencoder

Figure 2.18: Combination of VAE and NE [1]

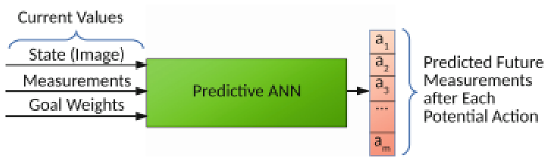
Model-Based

The goal of the DL component in model-based algorithms is to create an accurate predictive model of the environment and solve the main problems with model-free algorithms: (1) requirement of large data-sets and (2) not being able to transfer its policy to different tasks in the same environment. Creating an accurate predictive model of the environment can be a problem in high-dimensional environments. In Ellefsen et al. [11] they greatly simplified the prediction of future states by only predicting a few key measurements that are required to choose the optimal action given the state. Their approach is an extension of an algorithm called DFP (Direct Future Prediction) [8]. The process can be seen as transferring the RL problem into a supervised learning problem of predicting future states. The environment predictor used in this paper is a deep network, receiving data from a CNN architecture processing the image from the environment and two fully connected networks that are (1) a measurement module and (2) a goal module. This architecture is displayed in figure 2.19b.

A second paper mentioned earlier that used a VAE is Ha et al. [17]. Here, a VAE is used to compress image inputs and propagate a compact feature representation both to the controller but also to an evolved RNN, seen in figure 2.19a. The RNN is tasked with predicting the compact feature representation based on actions taken by the controller. This paper uses a combination between model-based and model-free, in that the controller both relies on the information from the environment and also takes into account the consequences of actions. The representation given by the RNN is used to generate an environment in which the agent can be trained inside and then transfer back to the real-world model.



(a) Flow diagram in [17] using VAE, RNN, NE



(b) DL component in Model-Based approach[11]

Figure 2.19: Two different approaches of combining deep learning and neuroevolution

2.3.3 NE-component

With either a compressed representation of the data or a predictive model of the environment ready to be used, a controller reacting to these representations is needed. As mentioned in the NE section, there are a lot of different algorithms to choose from, and in this section, the goal is to get an overview of the algorithms used in the relevant papers. NEAT is used in [11, 43] and CoSyNe [16] which is a extension of ESP is used in [29]. The two last papers used a CMA-ES algorithm to evolve their controller [1, 17]. Similar for all of the above implementations is that the controller outputs a vector; the elements in the vector are either actions or goals to pursue. With the algorithms defined, a table summarizing the input and output vectors of the NE components is displayed in table 2.1 on the next page.

Input/Output NE-component

Paper	Input	Output

Ellefsen et al. [11]	[m1, m2, m3]
	
	m1 = ammunition
	m2 = health
	m3 = Nr. enemies killed
		Goal-vector:
		
		[g1, g2, g3]:
		g1 = ammo-objective
		g2 = health-objective
		g3 = Monster killing
		
		Vector in range[-1,1]

Ha et al. [17]	[zt ht], where zt=vector with compressed input from the VAE, ht = hidden state from the RNN predictor.	at = W_c[zt ht] + bc W_c and b_c map input vector to output Based on the problem: Car-racing outputs = [steering left/right, acceleration, brake]
Koutnik et al. [29]	3-d compact feature vector, from the MP-CNN	Actions [o1, o2, o3]: avg(o1,o2) = steering o3=brake/throttle

Alvernaz et al. [1]	128-float values, representing the environment, compressed representation.	Actions [a1, a2, a3, a4]:
		a1 = left
		a2 = right
		a3 = forward
		a4 = Nr. times to repeat actions

Poulsen et al. [43]	Angular representations:
	[angle1, angle2, distance, in sight]
	
	Visual Partitioning Representation:
	25-dimensions, where each element is a place on the screen, element containing 1 shows where the enemy is
		Actions [a1,a2,a3,a4]:
		a1 = up
		a2 = down
		a3 = left
		a4 = right

Table 2.1: Summary of inputs and outputs

2.3.4 RL-problems solved

A variety of different RL-problems have been solved using this combination, the different environments is displayed in table 2.2.

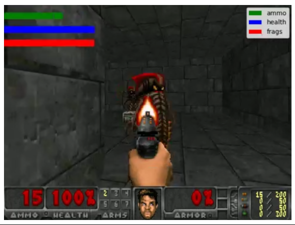
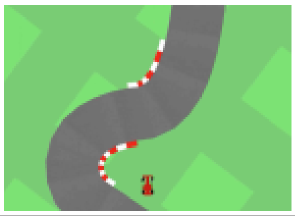
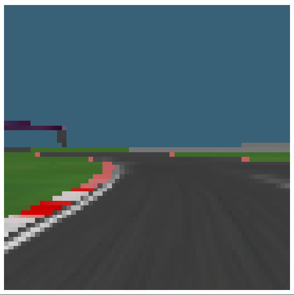
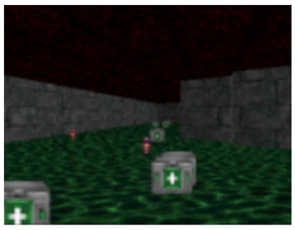
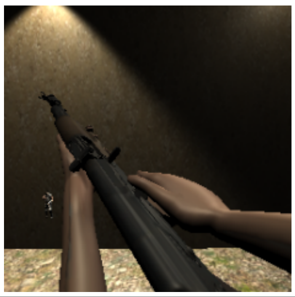
Screenshot	Experiment	Paper	Model-based/ Model-Free	DL	NE
	VizDoom	[11]	Model-Based	DL	NEAT
	VizDoom/ Car-Racing	[17]	Model-Based and Model-Free	VAE RNN	CMA-ES
	TORCS	[29]	Model-Free	MPCNN	CoSyNe
	VizDoom	[1]	Model-Free	VAE	CMA-ES
	FPS-Shoot	[43]	Model-Free	CNN	NEAT

Table 2.2: Summary of the different RL environments in related papers

2.3.5 Summary and contribution

Finally, it is interesting to see the contribution of the different papers and what underlying problems they were trying to solve. In Ellefsen et al. [11], using an extension of the DFP algorithm were able to show that their implementation allowed for the model to be successfully transferred back to the environment even when the optimal policy was changed. Their approach was also able to change the goals on-line, based on changes in the environment, which is highly interesting with real-world applications in mind, where the environment is ever-changing, and the goals will differ based on the feedback from the environment.

In Ha et al. [17] the focus was to generate an entire model of the environment, using visual sensory inputs and creating this model using an RNN architecture. This approach allowed for training the agent inside the generated environment and successfully transfer the learned policy back to the original environment. While this method allows for training the agent with less computational resources with a simpler predictive model, this model is subject to errors and enables the agent to cheat and exploit problems within the generated model, which will not be good when transferred back to the real environment.

Koutnik et al. [29] experimented with evolving the DL part of the combination by using a UL-ERL algorithm that evolved a CNN capable of extracting features without the need for supervised learning. This approach has its advantages and disadvantages; letting the algorithm itself figuring out what features it needs to solve a problem can make the model more effective, and this approach could easily be fitted to be an end-to-end approach to new problems. The disadvantage, however, is that evolving large structures is slow and computationally heavy.

In Alevernaz et al. [1], their goal was to show that the DL, or more specifically the VAE model, could successfully compress the 3D input so that an EA can be effective. Although they did not manage to outperform a deep-q network, this approach showed that VAE successfully decompresses the visual input to compact features, and this can be promising with the VAE being able to be trained on-line while the game is playing. Different representations of data have been shown in table 2.1, and choosing which representation is best for a given problem is not an easy task.

In Poulsen et al. [43] their main contribution was comparing the different representations, angular and visual partitioning, given by the DL component to the NE, and see how this affected training time and result. They also experimented with different depths of their DL model, comparing the difference. This can give useful insight on choosing the representation given to the NE component. However, this looks pretty problem-specific, and it is not apparent how this research can be applied to problems that are not very similar to this.

Chapter 3

Tools and frameworks

This chapter attempts to give an introduction to the different tools and frameworks used throughout the thesis. The background chapter lays the theoretical groundwork for the work in this thesis while this chapter focuses on the environments and algorithms applied.

3.1 Environment

Every living organism is interacting with its environment and uses the feedback from the environment to improve its actions to better survive and thrive. Similar artificial environments are created to solve various RL problems. These environments generally model a simplified version of a more complex problem. Depending on the problem, these environments give the agent situated in this artificial environment and controlled by the algorithm feedback that can help it solve the problem. There is a huge variety of RL environments available, and for this thesis, a few important aspects are considered in choosing the environment. These aspects are linked to the problem this thesis is trying to address: (1) It has to be sufficiently complex so that DL and NE can benefit from a combination, (2) the ability to work with high dimensional data, and (3) the ability to work with both non-sparse and sparse reward, this implies that the problem and the reward can be made sparse.

3.1.1 Bipedal Walker

The environment chosen for the experiments in this thesis is Open AI's gym environment Bipedal Walker v3 [3]. The Bipedal Walker is a two-dimensional environment consisting of an agent walking horizontally, portrayed in figure 3.1. In the environment, the agent experiences gravity and friction, similarly to the real world. The anatomy of the agent consists of a rigid five-sided body and two identical legs as seen in figure 3.1. The legs are divided into two joints and are arranged as front-leg (light brown color) and back-leg (dark brown color). The environment itself produces 24 observations or sensor outputs. The sensor outputs consist of different measurements, such as velocity in the x and y direction, the state of the agent and its legs, and ten lidar readings that provide different distance measurements. The complete sensor output can be seen in table 3.1. The chosen algorithm receives the observations from the environment and should produce an action vector consisting of 4 values. These four values range between -1 and 1 and are the torque applied to the four different parts of the legs, more precisely: knee1, hip1, knee2, and hip2. The maximum number of time-steps in the bipedal walker environment is 1600; time-steps are related to the number of actions received by the environment from the algorithms.

Environment Goal

The goal is to walk as efficiently and fast as possible to accumulate the highest possible reward. There are two different versions of the Bipedal Walker environment, normal and hardcore. Normal has a flat ground, while hardcore has obstacles, holes, and inclines. The threshold for considering the environment solved is a mean reward of 300 over 100 episodes. The reward function is further discussed in the next section.

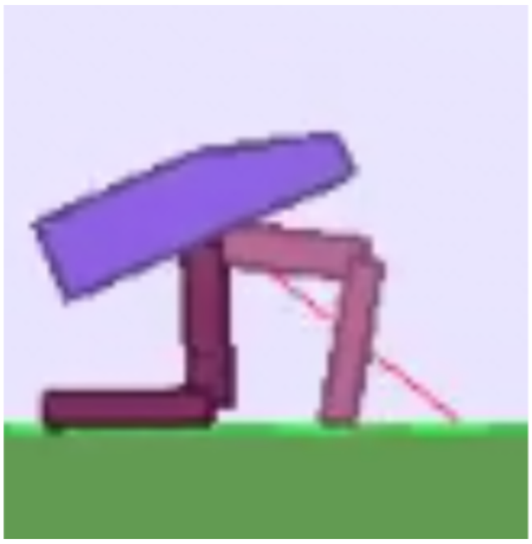


Figure 3.1: Bipedal Walker [3].

Num	Observation	Min	Max	Mean
0	hull_angle	0	2*pi	0.5
1	hull_angularVelocity	-inf	+inf	-
2	vel_x	-1	+1	-
3	vel_y	-1	+1	-
4	hip_joint_1_angle	-inf	+inf	-
5	hip_joint_1_speed	-inf	+inf	-
6	knee_joint_1_angle	-inf	+inf	-
7	knee_joint_1_speed	-inf	+inf	-
8	leg_1_ground_contact_flag	0	1	-
9	hip_joint_2_angle	-inf	+inf	-
10	hip_joint_2_speed	-inf	+inf	-
11	knee_joint_2_angle	-inf	+inf	-
12	knee_joint_2_speed	-inf	+inf	-
13	leg_2_ground_contact_flag	0	1	-
14-23	10 lidar readings	-inf	+inf	-

Table 3.1: Bipedal Sensors [3].

Reward Function

The reward is given from the environment; every time the agent performs an action a, it receives a numerical reward value from the environment. This value indicates if the action was good or bad. If the agents accumulate sufficient speed without falling over, it will reach a total reward of 300+, which is the goal of this environment.

$$r(t) = \alpha(x(t) - x(t-1)) - \beta \sum_{i=0}^{3} |a_i| - 100(g_t) \tag{3.1}$$

Equation 3.1 shows: $g_t = 1$ if the body or hull hits the ground, and 0 otherwise. $r(t)$ is the received reward at time-step t. $x(t)$ is the x position at time-step t and $x(t-1)$ is the last x position at time-step $t-1$. α and β are scaling variables.

The reward is accumulative, meaning that the agent receives a reward for each action/time-step, and the total reward is all the reward accumulated from one run in the environment. If the bipedal walker falls over, it receives a reward of -100.

3.2 Algorithms

3.2.1 SAC

Soft Actor-Critic (SAC) [18] is the main RL algorithm used in this thesis, chosen from initial experiments on the original implementation of the gym environment bipedal-walker. Actor critic algorithms learn both policies and values. The actor learns the policy while the critic criticise the actor's choice. The main part of the algorithm utilizes the Stable-Baselines [24] python package. During the initial experimentation, SAC has shown to be a fast algorithm to find a viable solution and completing the environment without problems. SAC is defined for RL tasks involving continuous actions, and it is an off-policy, model-free, deep learning algorithm. SAC aims not only to maximize the lifetime rewards but also the entropy of the policy. Entropy can be seen as the randomness in the policy. It wants to maximize the entropy of the policy to encourage more exploration explicitly. This encourages the algorithm to assign equal probabilities to actions with the same Q-value, which ensures that the algorithm does not repeatedly select the same actions. The objective function consists of both the reward term r and the entropy term $\mathcal{H}$:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\rho_\pi}[r(\mathbf{s}_t,\mathbf{a}_t) + \alpha\mathcal{H}(\pi(|\mathbf{s}_t))] \tag{3.2}$$

Equation 3.2 shows: α is a temperature variable that controls the relative importance of the entropy against the reward. SAC uses three different types of networks, (1) state-value function $V_\psi(\mathbf{s}_t)$, (2) soft Q-function $Q_\theta(\mathbf{s}_t,\mathbf{a}_t)$ and (3) policy function $\pi_\phi(\mathbf{a}_t|\mathbf{s}_t)$. Each of the separate networks optimizes different error function, shown in equation 3.3, 3.4, 3.5 and 3.6.

Value network:

The state-value network is trained to minimize the squared residual error:

$$J_V(\psi) = \mathbb{E}_{\mathbf{s}_t\sim\mathcal{D}}[\frac{1}{2}(V_\psi(\mathbf{s}_t) - \mathbb{E}_{\mathbf{s}_t\sim\pi_\phi}[Q_\theta(\mathbf{s}_t,\mathbf{a}_t) - \log\pi_\phi(\mathbf{a}_t|\mathbf{s}_t)])^2] \tag{3.3}$$

Equation 3.3 shows: $\mathcal{D}$ is the distribution of the replay buffer. The target is to minimize the error between the prediction of the Q-function plus the entropy of the policy function π.

Q-network:

The Q network is trained to minimize the soft Bellman residual, showed in equation 3.4:

$$J_Q(\theta) = \mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t)\sim\mathcal{D}}[\frac{1}{2}(Q_\theta(\mathbf{s}_t,\mathbf{a}_t) - \hat{Q}(\mathbf{s}_t,\mathbf{a}_t))^2] \tag{3.4}$$

where

$$\hat{Q}(\mathbf{s}_t,\mathbf{a}_t) = r(\mathbf{s}_t,\mathbf{a}_t) + \gamma\mathbb{E}_{\mathbf{s}_{t+1}\sim p}[V_{\bar{\psi}}(\mathbf{s}_t + 1)] \tag{3.5}$$

Equation 3.5 is calculating the squared difference between the prediction of the Q-function and the immediate reward plus the discounted expected reward for the next state. This is done over all action, state pairs. In equation 3.5 the term $V_{\bar{\psi}}$ is a target value network where ψ is the moving average of the value network weights. This moving average has shown to be good to stabilize the training [18].

Policy-network

Lastly, the final network aims to minimize the expected KL, or Kullback-Leibler divergence noted as D_{KL}

$$J_\pi(\psi) = \mathbb{E}_{\mathbf{s}_t\sim\mathcal{D}}[D_{KL}(\pi_\psi(\dot{|}\mathbf{s}_t))||\frac{exp(Q_\theta(\mathbf{s}_t,\dot{)}}{Z_\theta(\mathbf{s}_t}] \tag{3.6}$$

This optimization, showed in equation 3.6, aims to make the distribution of the policy function similar to the distribution of the exponentiation of the Q function normalized by a function Z. Minimizing the objective can be done in multiple ways. The authors of the paper used a reparameterization trick to make sampling from the policy a differentiable process, and this results in avoiding problems with back-propagating the errors in the neural network.

Reward

Scaling the reward down can lead to a nearly uniform policy that fails to exploit the reward signal, leading to substantially lower performance [18]. With the reward scaled up, the policy can become nearly deterministic, with a fast learning curve at first but leading to poor local minima due to insufficient exploration. The only hyperparameter that needs tuning in SAC is the reward scaling [18].

Summary

SAC is an actor-critic, off-policy deep learning algorithm, using entropy to force the agent to maximize reward simultaneously with exploration; in other words, it solves the environment while trying out random new actions. SAC's advantages are that it focuses on exploring new solutions and gives up solutions that are unpromising and the ability to assign equal probability to the actions that seem equally attractive for solving the problem. This is particularly useful in environments where several different solutions are desirable.

3.2.2 NEAT

The neat-python package [36] that is based on the original paper from Stanley et al. [53] is used for the experiments in this thesis. This section aims to describe the specific parts of NEAT that stand out regarding EA and NE described in the background section.

Initial Population

There are several different approaches to initializing the initial population used in neuroevolution; while input and output nodes are fixed, the number of starting hidden nodes and connections can be set. Some algorithms start with a random population, but this can lead to several problems, one being the input not being connected to the output. The main goal of neuroevolution is to find a minimalistic solution to a problem. Therefore creating random individuals with many potentially unnecessary connections and nodes forces the network to spend time getting rid of these. NEAT has as a design principle to start small, with only input and output nodes, no hidden nodes, and evolve from there, leading to minimalistic solutions [53]. A few initial experiments in this thesis focused on the difference between an "unconnected" or "full" starting architecture. "Full" means that all input nodes are connected to all output nodes, and "unconnected" means that there are no connections at all. The number of hidden nodes is set to zero to keep the networks as minimalistic as possible. Depending on the problem, "full" and "unconnected" can be beneficial, "full" takes more time initializing but can create connections that might not be necessary. In more challenging problems, like using images in the bipedal-walker environment, "full" seems to have a bigger chance of leading to a solution rather than being stuck. "Unconnected" often lead to minimalistic networks, less time initializing, faster generations, and in easier problems can lead to a faster solution. Therefore in the main experiments in this master thesis, the full implementation has been used—both in regard that it performs overall better and that K.O. Stanley et al. [53] apply full connect when doing experiments in the original paper. One experiment has been conducted showing the difference between these, and it can be found in Section A.1 in the appendix.

Activation function

One activation function has been used in the experiments of this thesis, Tanh, which is shown in figure 3.2. This activation function keeps the output values that range between -1 and 1.

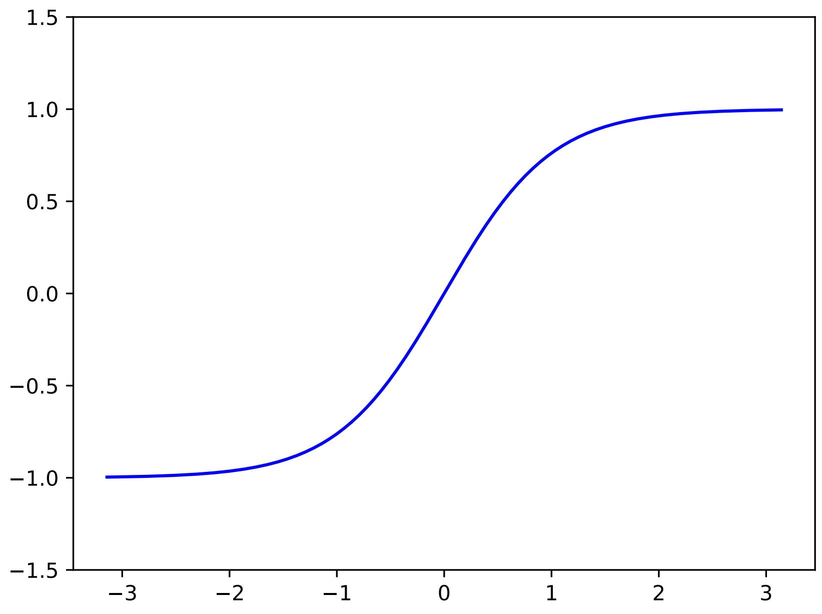


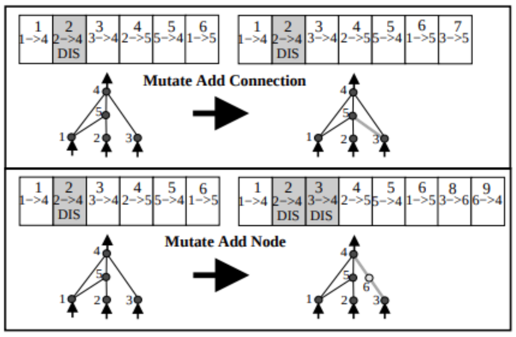
Figure 3.2: Activation function tanh

Crossover

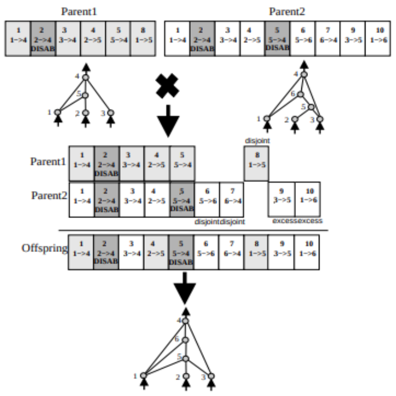
In neuroevolution, where we are often dealing with complex structures, there is no intuitively good approach to perform crossover without ruining the performance of the two networks. Because of this, some researchers have given up this option [54]. NEAT solved this by adding a ”global innovation number" [53]. These numbers are markers that identify each gene’s original historical ancestor, which gives an overview of the chronological order of every added gene in an individual. This gives NEAT two different matching genotypes, meaning that the innovation numbers of the parents are lined up, or disjoint (middle nodes that do not match)/excess (leaf nodes that do not match) depending on the differences of the two parents. Crossover is performed by choosing random matching genes from the parents, done by using the innovation numbers, and all of the excess and disjoint genes are added from the parent with the highest fitness of the two. An example of crossover can be seen in figure 3.3b.

Mutation

In neuroevolution, the mutation is performed by changing weights, adding a node, a connection, or a combination of these; an example of mutation is depicted in figure 3.3a.



(a) Mutation



(b) Crossover

Figure 3.3: Evolutionary operators for NEAT [53]

Speciation

Performing mutations and crossover in NE usually do not give the network an instantly better performance, rather the opposite. The network needs time to optimize the newly added changes to see the full potential of this mutation or crossover. To solve this, NEAT [53] uses speciation. Speciation divides the individuals in a population into discrete niches, and the individuals only compete with individuals in their own niche rather than the entire population. This speciation process can result in better diversity by giving the different individuals more time to optimize. The speciation can be seen as a way of protecting the innovation of new behaviors.

Selection

The selection process evaluates each of the individuals in the chosen environment and looks at the fitness score. The selection process is then based on these hyperparameters, how many species are there, how big the difference should be between individuals to be constituted as a separate species, and, finally, how many individuals in each species should be chosen as parents. The parents are chosen, with the best performing individuals having a higher chance of being chosen within each species.

Genotype

The genotype, depicted in figure 3.4, in NEAT, is two genes, node genes and connection genes. The node gene specifies the type of node (input, hidden, output), the innovation number (historical markings), activation function, bias, aggregation (sum of input, or multiplication of input), and response (all the attributes of a node: activation function, bias, and aggregation). The connection gene specifies the innovation number, the weight, and if it is enabled or not, input and output. A network can be created from this representation, which is the phenotype.

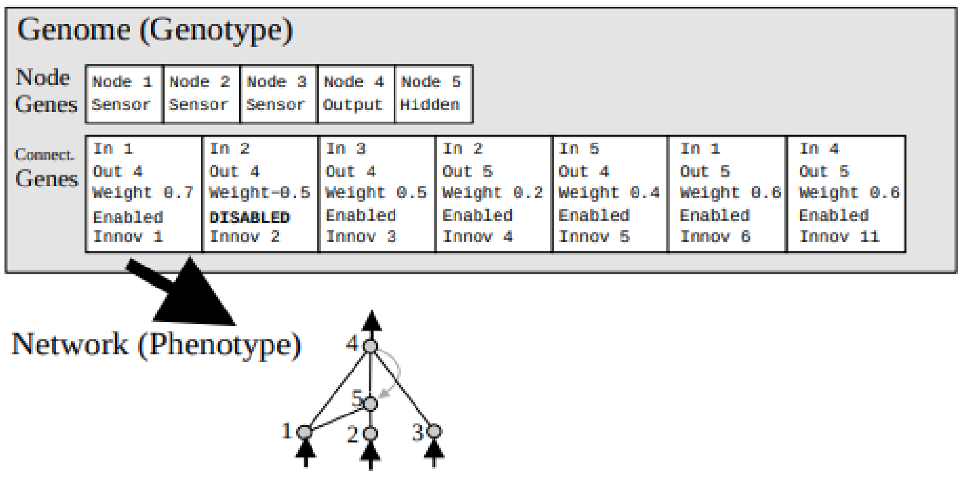

Genome (Genotype)
Node Genes
Node 1 Sensor
Node 2 Sensor
Node 3 Sensor
Node 4 Output
Node 5 Hidden
Connect. Genes
In 1 Out 4 Weight 0.7 Enabled Innov 1
In 2 Out 4 Weight-0.5 DISABLED Innov 2
In 3 Out 4 Weight 0.5 Enabled Innov 3
In 2 Out 5 Weight 0.2 Enabled Innov 4
In 5 Out 4 Weight 0.4 Enabled Innov 5
In 1 Out 5 Weight 0.6 Enabled Innov 6
In 4 Out 5 Weight 0.6 Enabled Innov 11
Network (Phenotype)
4
5
1
2
3

Figure 3.4: Genotype to Phenotype mapping example [53]

Parallelization

As mentioned in the background section, EAs are highly parallelizable; the number of workers is limited to the number of threads and cores and can also be applied to cloud computing, a network of computers working on the same task. This results in a significant speedup when running EAs on computers with many cores.

3.2.3 DL

Most of the deep learning theory was described in the background section 2.1.3, so this section will primarily describe how the deep learning model was trained and the process of choosing the architecture, optimizer, and loss function. The choice was conducted experimentally. Different permutations of optimizers, loss functions, and architectures were cross-tested to find which combinations resulted in the best possible model with the dataset from Bipedalwalker. The selection of optimizers, loss functions, and architectures are briefly explained in the following sections.

Keras

Keras [6] is a deep learning API running on top of the machine learning platform Tensorflow. This python package is well documented and widely used. This package was used to create and train the model.

Optimizers

Optimizers are the glue that ties the loss function together with the model and its parameters; where the loss function tells which way to go, the optimizers take the steps and update the model accordingly. This section will describe the optimizers used in this thesis shortly.

Momentum is a term used below. This term can be seen as how big a step is taken in the steepest direction. Like a ball rolling down a hill, it builds up momentum. This extra momentum value can, in some cases, lead to faster convergence. Nesterov accelerated gradient, added to a momentum term, can be seen as a more intelligent momentum term, meaning that when it reaches near a minimum value which is the goal, it slows down. Like momentum but with a look-ahead function.

Stochastic gradient descent (SGD) is based on gradient descent, explained in the background section. Instead of updating for every data point as done in gradient descent, the stochastic gradient descent algorithm randomly chooses a few samples from the dataset, saving a lot of time for a large dataset. RMSprop and Adadelta came about the same time, and both tried to solve the problem with fast vanishing learning rates. Adagrad, a widely used optimization technique, suffers from this. It performs small changes(low learning rate) for features that appear frequently and significant changes (high learning rate) for features that appear infrequently. The learning rate is based on the squared of all past squared gradients. Adadelta and RMSprop solve this by only storing a fixed number of past gradients. These gradients then decay according to an average of these gradients, multiplied with a momentum value. Like Adadelta and RMSprop, Adam has an adaptive learning rate and, in addition, has an exponentially decaying momentum value. Nadam is similar to Adam but using Nesterov accelerated gradient.

Loss-Functions

Loss functions are used as an evaluation method for how well the model fits the dataset. The model in this thesis is not classifying the data but predicting numbers, meaning that this is a regression problem. The loss functions chosen are some of the most commonly used in these kinds of machine learning problems.

Mean Squared Error (MSE) loss function takes as the name implies the mean and squares the error between the predicted and actual value showed in equation 3.7. This squaring gives it the property that far-off predictions get penalized heavily while close predictions get a small penalty. It exhibits problems when nearing the minima, where the gradient might be too small.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2 \tag{3.7}$$

Huber loss, seen in equation 3.8, includes a δ term that controls how small an error has to be to take the quadratic of the error. Depending on this δ value, Huber loss is Mean Absolute Error (MAE) and becomes (MSE) when the error is small. MSE when $\delta \sim 0$ and MAE when $\delta \sim \infty$. MSE has a problem with small gradients, and MAE can have problems with large gradients, meaning it might overshoot and miss the minima. Huber loss can work well in these cases, with small and large gradients, but the δ term needs to be tuned, which can be a challenging and tedious task.

$$L_i = \begin{cases} (z^(i))^2 & \text{for} |z^(i)| \leq \delta \\ 2\delta |z^(i)| - \delta^2 & \text{for} |z^(i)| > \delta \end{cases} \tag{3.8}$$

The logcosh loss function showed in equation 3.9 takes the logarithm of the hyperbolic cosine of the prediction error. This makes it very similar to MSE but will not be affected so strongly if there are a few predictions that are far off. It has all the advantages of Huber Loss but is twice differentiable everywhere.

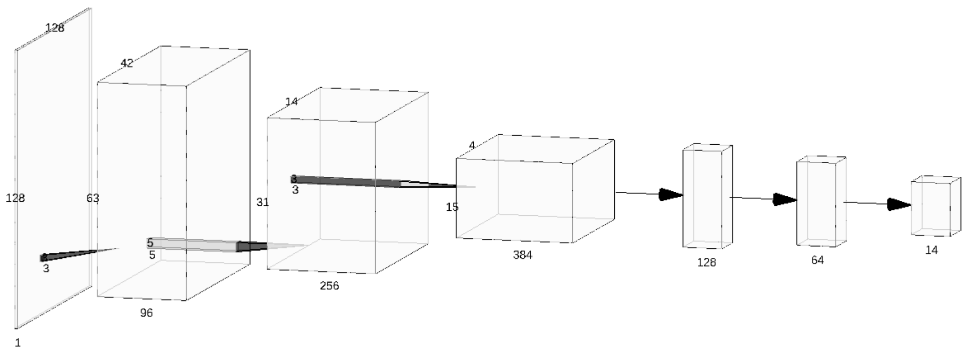
$$L(y, y^p) = \sum_{i=1}^{n} log(\cosh(y_i^p - y_i)) \tag{3.9}$$

Cosine Similarity is a measurement of similarity between two n-dimensional vectors, seen in equation 3.10, where a and b are vectors. In this case, the similarity between the predicted vector and the actual vector. It is defined as the cosine of the angle between the vectors. Giving a positive value of 1 if the vectors are parallel in the same direction, 0 if they are orthogonal, and -1 if they are parallel but pointing in different directions.

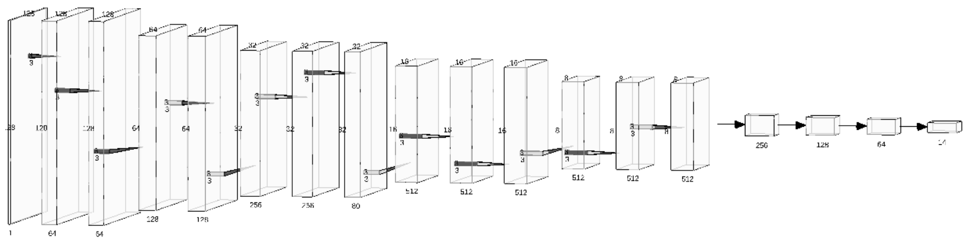
$$\cos(a, b) = \frac{ab}{\|a\| \|b\|} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} (a_i)^2} \sqrt{\sum_{i=1}^{n} (b_i)^2}} \tag{3.10}$$

Architecture

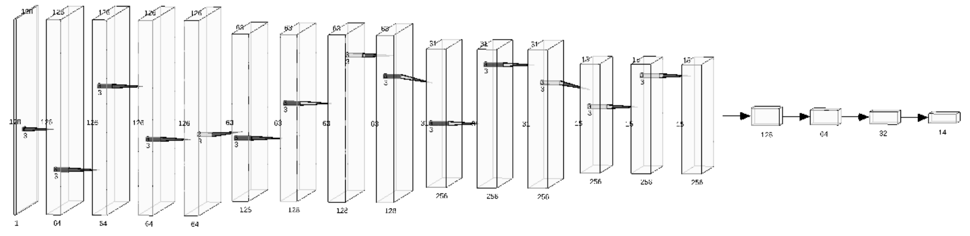
One of the most challenging parameters to choose is the number of layers, nodes, weights, and biases, so the simplest way is to look at earlier work done in machine learning with images. Looking at competitions and reputation in the field of image classification and processing over the last decade, two models stand out: (1) AlexNet[31], showed in figure 3.5a, won ImageNet Large Scale Visual Recognition Challenge on September 30, 2012, by a significant margin, (2) VggNet[50], showed in figure 3.5b, won the ILSVRC (ImageNet Large Scale Visual Recognition Competition) 2014 in the classification task and is the first deep learning classifier that got an error rate below 10%. The author created a third model based on the VGGNet model, with extra layers, shown in figure 3.5c.



(a) Alexnet



(b) VGG16



(c) VGG Altered

Figure 3.5: 3 Different deep learning architectures

3.3 Hardware

This section provides an overview of the hardware employed in this dissertation, divided into CPU and GPU. The number of CPU hours is also estimated to indicate the amount of work that has been done in this thesis.

- CPU:
  - NREC (Norwegian Research and Education Cloud)
    - 16 core, 32 GB RAM virtual computers
    - 4 Nodes October to January, upgraded to 10 in January-May
    - November-Present, rarely not running experiments, so upwords to 100% percent active usage
  - Rudolph (Threadripper)
    - 2 x AMD Ryzen Threadripper 1950X 16-Core Processor, 3xGeForce GTX 1080 Ti
    - 7 days
  - ROBIN NREC Node
    - 1 x 120 core, 460GB RAM
    - 18 December to 21.january
  - Personal Laptop
    - Used throughout the entirety of the thesis, sums up to 20-30 full days of active usage
    - Laptop: Intel Core i7-7500U CPU @ 2.70 GHz 4 core, 8 GB RAM
- GPU:
  - Dunder (GPU Computer at Department of Informatics at the University of Oslo)
    - 1 x Intel i7 8700K (6-core), Nvidia RTX3090
    - 20 days of continous usage
  - Dancer (GPU Computer at Department of Informatics at the University of Oslo)
    - 1 x Intel i7 8700K (6-core), Nvidia RTX3090
    - 20 days of continuous usage
  - Personal Computer
  - Intel Core i5-7600K CPU @ 3.80GHz, 4 core, 8 GB RAM, GeForce GTX 1060 6 GB
  - 3 months of continous usage

Approximating the exact number of CPU-hours spent in this thesis is hard, but a rough estimate is 550 000 to 600 000 CPU hours (62 years), over all hardware through the entirety of the thesis, including all preliminary experiments and main experiments ++.

Chapter 4

Implementation

This chapter aims to describe the changes and customization to the tools, algorithms, and environments in the thesis. This chapter also describes the experimental setup and challenges that were faced during the experiments.

4.1 Environment

The environment itself can be optimized to save time and reduce the dimensionality of the data. These strategies quickly became necessary to stay within the computational budget of a master project. Some of these challenges are further explained in section 4.5.

Time-steps

The maximum number of time-steps in an episode of bipedal-walker is 1600. However, initial experiments showed that, on average, the agent only uses between 850-1050 time-steps to complete the environment, depending on the walking pattern and algorithm. The maximum number of time-steps was set to 1100 to save time and increase learning speed.

Sensor values

The original environment provides 24 sensor inputs, as described in the last chapter, which is the agents' state and terrain information through the lidar values. However, in the normal version of the environment, the last ten sensor values should not be needed to complete the environment. The last ten sensor values are lidar values that give information about different lengths and the road in front of the agent. However, without inclines, holes, and obstacles in the normal environment, these values are not needed. So when the algorithms are used together with a DL model that predicts only the 14 first (state) values, the lidar values are not a part of the experiment.

Image

The environment provides a method for extracting screenshots directly from the environment, and with some small alterations in the code, this was used as the high-dimensional observations in this master thesis. The screenshot is an RGB NumPy array that provides an image of the agent and its immediate surroundings. However, some optimizations can be done to simplify the problem further and decrease learning time.

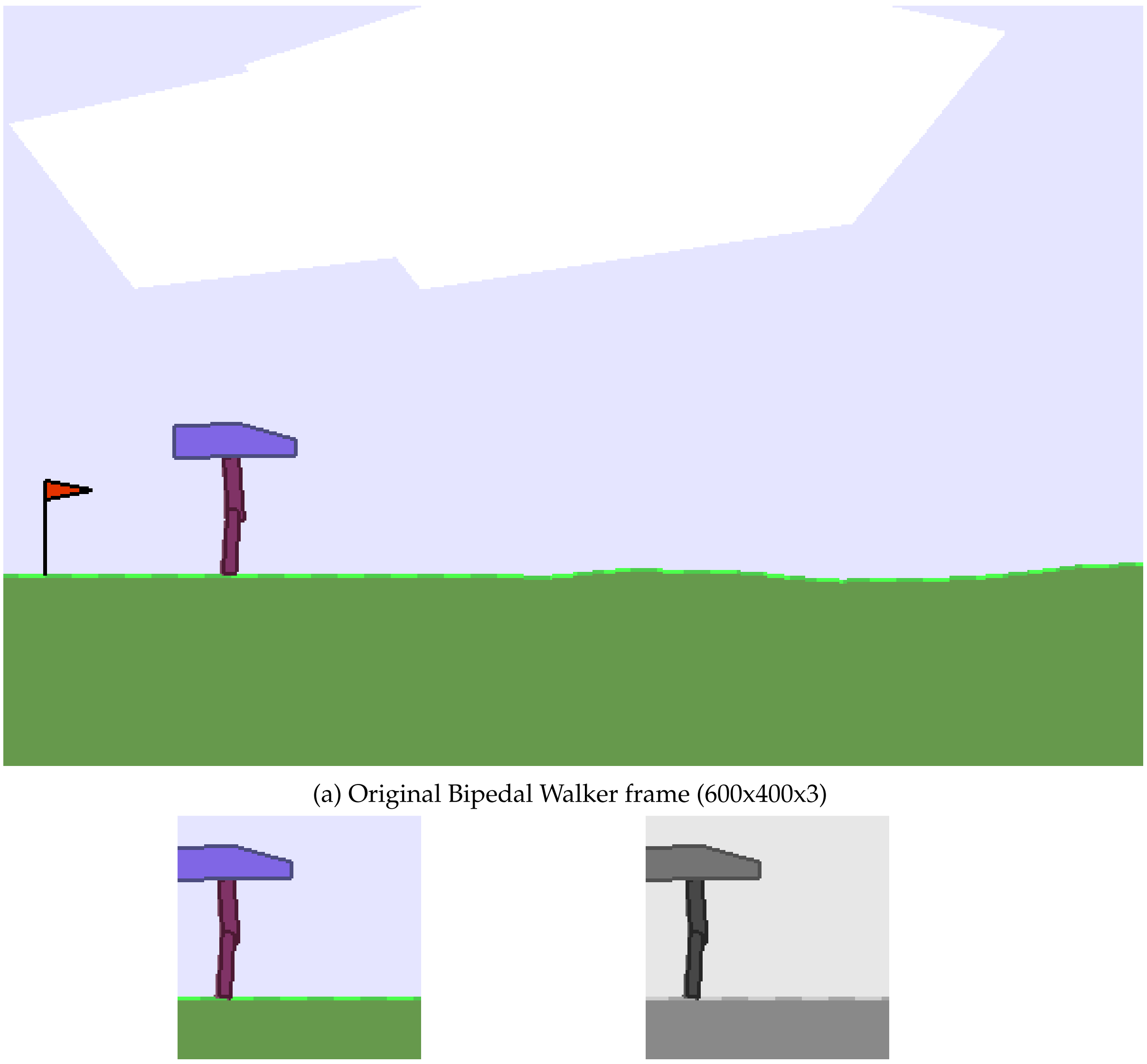
Size of environment The original bipedal-walker frame is 600x400x3. However, this image size provides much information about the environment that is unnecessary for this task. More precisely, a lot of information about the road ahead of the agent. This extra information can be helpful in a hardcore version of the environment with obstacles, holes, uphills, and downhills, but not in the normal mode, which is just flat ground. So by cropping the frame to be 128x128x3, a high-dimensional environment is still present, but with only necessary information.

Grey The image is an RGB NumPy array, meaning that it is a colored image. The colors do not provide any necessary or beneficial information. The images have been converted into a greyscale image to lower the dimensionality from 128x128x3 to 128x128x1.

Other image processing A few other image processing steps have been added to the algorithms, such as normalizing the pixel values between 0 and 1 and removing zero-valued pixels to avoid zero division which might arise in some of the algorithms. Dimensionality reduction has also been added as support to the algorithms struggling with high-dimensionality to make them more competitive with the other algorithms. NEAT, for example, is known for struggling with high-dimensionality. Other image processing, like flattening and such, is added in response to what the different algorithms expect as input.

Results By removing unnecessary information from the environment, the dimensionality is reduced significantly. The actual change to the environment image can be seen visually in figure 4.1 and the calculations to the reduced dimensionality are showed in equation 4.1.

$$\begin{aligned} 600x400x3 &= 720000 \\ 128x128x1 &= 16384 \\ 720000 - 16384 &= \underline{703616} \end{aligned} \tag{4.1}$$


(a) Original Bipedal Walker frame (600x400x3)

(b) Cropped frame (128x128x3)

(c) Greyscale (128x128x1)

Figure 4.1: This figure displays the changes done to the original frame in the environment, by cropping and removing color channels. Screenshots are extracted from bipedalwalker environment [3].

4.1.1 Sparse rewards

One of the central goals of this thesis is investigating challenges with sparse rewards. Therefore, a change in the environment's reward function is needed. There are several ways of making the reward sparse; the main focus of this thesis will be the agents' position. The reward function is modified so that only the x-position contributes to the accumulated reward, implying that the agent needs to move forward to receive rewards. However, one important aspect to note is that the starting x-position of the bipedalwalker is around 4.66.

The reward function is simple, using a start threshold, threshold increase size, and a fitness value. The start threshold value is how far the agent has to move in the forward direction before receiving its first reward. The threshold increase size is the distance the agent has to walk after reaching the start threshold to receive its next reward. In between those threshold step sizes, the agent receives 0 in reward or -100 if it should fall over. Figure 4.2 displays this fitness function graphically.

A constant threshold step size of 5 is used in our experiments, meaning that when the agent reached the starting threshold, it has to walk another 5 to receive the next reward. The fitness value received at each step is 1.0. Different starting thresholds are tested and experimented with, described in the experiment and result chapter.

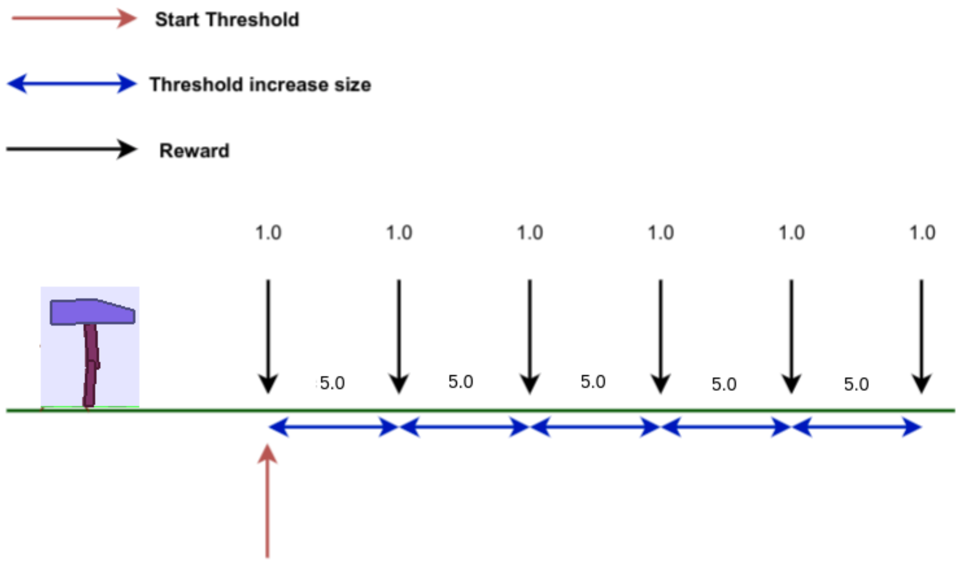


Figure 4.2: Sparse Reward Function

Maximum Reward: The max position in the environment is approximately 90 meters, and this means that when the reward function is changed, so is the maximum achievable reward. The maximum achievable fitness is naturally related to the starting threshold and is depicted in table 4.1.

Start Threshold	Max Fitness
5	17
10	16
15	15
20	14
25	13
30	12

Table 4.1: Maximum Fitness values according to the start sparse threshold

4.2 Dataset

When training the Deep Learning model used for feature extraction to the NEAT controller, a dataset is required to train the DL model. Screenshots extracted from the bipedal walker environment make up the dataset. The initial idea was to create a dataset using only random actions. However, to get a more diverse dataset, the dataset's creation was divided into three different parts, the first part is random actions, the second part is an original environment solution from the NEAT algorithm, and a solution from the SAC algorithm. Using all three should lead to a more diverse dataset and a better, more robust, and accurate DL model.

To greatly simplify the prediction for the deep learning model, only the 14 first (agent state) of the 24 original sensors, displayed in table 3.1, values from the environment are predicted. As explained earlier, the last ten digits are the lidar values, and these values are not useful in the normal version of the environment. The first 14 values include the velocity in both x and y directions and the speed of the joints. These values were extracted simultaneously as the images in the dataset and are used as the correct labels that the deep learning model will try to predict.

4.3 Algorithms

The experiment code used in this thesis is open source and is available at [1].

4.3.1 SAC and NEAT

Keeping these algorithms close to the original implementation has been important. Since optimizing one or the other can make the comparison unfair, it also increases the applicability of our work to others using the same algorithms.

NEAT

The NEAT package was found by looking at the papers that have done similar work. This package includes a config file with around 40+ hyperparameters that can be tuned. The first couple of months, NEAT was being run with different activation functions, starting architectures, speciation thresholds, survivor amount, the number of individuals in each species and, more. These experiments were necessary for finding what works well and what does not work well for this problem. High-dimensionality data can be a problem, especially NEAT, so options for downsampling the input have been added in the experiments where this occurs.

[1]https://github.com/matstveter/master_thesis

SAC

The RL algorithms were chosen based on the performance in the naive implementation of the bipedal environment. A few different algorithms were tested, and different implementations of the same algorithms. Finding an implementation/package that could easily be modified and easily understood has helped in the later stages of the thesis. The different RL algorithms tested were: PPO, DQN, SAC, A2C, DDPG, and more.

From the initial experiments, SAC was chosen due to its fast time to solve the environment and part of a python package that was easy to change to fit the requirements of this thesis. This package was based on the original paper on SAC. According to the original SAC paper, hyperparameter tuning is not necessary other than the fitness function itself [18].

4.3.2 DL

Activation functions

The original architectures below use the "relu" activation function on all layers. However, in the authors' take on the three architectures below, a "relu" activation function was used on the convolutional layers, and a "tanh" activation function was used on the fully connected dense layers. The reason for this is that with the 14 sensor values depicted in 3.1, most range from -1 to 1, with "relu", all negative values would be set to 0. Therefore changing this activation function to "tanh" allows for keeping the output between -1 and 1, which should fit perfectly for the environment.

Architectures

Figure: 3.5a,3.5b,3.5c, shows the different architectures experimented with during the initial DL model training. The aim was to find the model that best fit the data in the dataset. These figures represent the convolutional layers and the dense layers, not the max-pooling layers. The max-pooling layers can be spotted by looking at the difference in image size in between layers. If there is a large decrease in the image size, it is due to a max-pooling operation. The main difference between the architecture used in this thesis and the original architectures is the sizes of the dense layer. For example, the original alexnet architecture has these dimensions in the final three dense layers, 4096–4096–1000, but only 14 values are needed in the output for our experiments. So this resulted in changing the final three-layer dimensions to 128-64-14 to better fit the problem. The same change has also been done to the VGG16 model.

Alexnet Alexnet uses a Dropout layer between the third and second last layer, with a probability of 0.5, meaning that there is a 50% likelihood for each neuron to get dropped out.

VGGNet and VGG These architectures are quite similar, but the VGG in figure 3.5c was created from the VGGNet version by just adjusting some of the dimensions of the convolutional layers and adding an extra layer.

Choice of architecture, optimizer and loss function

Extensive cross-testing was performed by varying the four different optimizers, three model architectures, and four loss functions. All of the different cross-tested parts resulted in 48 models trained and evaluated. Alexnet with a mean squared error function and the Adam optimizer performed slightly better than the rest, architecture displayed in figure 3.5a. A few of the state predicted values were velocities that are not possible to compute from images. Some hyperparameter tuning was tested, adding more layers, adding dropout layers, and so on, without achieving a substantial increase in performance. These experiments resulted in choosing an Alexnet architecture, using an Adam optimizer with a Mean-Squared-Error function.

4.4 Experimental Setup

The environment utilized in the experiments is the Bipedalwalker environment described in section 3.1.1. The types of data from the environment are either sensor values from the original environment or images extracted from the environment. The reward function is the normal original fitness function or the authors' custom sparse fitness function. The algorithms are trying to accumulate the highest possible fitness, which is done by walking at the highest possible speed in the environment without falling.

Goal of experiments

The goal or threshold for solving the original environment is accumulating a fitness of 300+ over 100 generations. However, in this thesis, the goal has been changed to reach a threshold of 300 or the equivalent in the sparse reward domain which varies greatly with different starting threshold. Although averaging above 300 in fitness over 100 generations is ultimately the goal, the first steps are to see if the algorithms are able to learn and improve from the supplied inputs and combinations, which is the main focus of this thesis. The problems used in this thesis are more complex than the original environment, so solving the environment might require additional training time and may not be feasible with the time and resources available in a master project.

Termination Criteria

The termination criteria are either maximum number of generations, which is set to be 2500 generations or 500000 episodes. This maximum number of generation was set, by looking at similar work in the related papers. Finishing the environment based on a threshold fitness value (reaching

300 fitness for neat) or a non-improvement threshold was set to be 100 generations or 20000 episodes. The non-improvement threshold was sat mainly to finish the results in time and obtain the results needed. This can bee seen in table 4.2

	Fitness Naive	**Fitness Sparse**	**Non-Improve**	**Max**
NEAT	300	See Fig: 4.1	100 Gens	2500 Gens
SAC	300	See Fig: 4.1	20000 Ep	500 000 Ep

Table 4.2: Maximum fitness thresholds, and termination criterias

Comparison

Generations vs. Episodes NEAT is a population-based algorithm that is run in generations. One generation equals to one run in the environment for all individuals in the population. SAC is not population-based, so it is measured in episodes. This difference makes the comparison somewhat difficult, but to attempt to make the comparison as fair as possible, the relation between episodes and generations is $episodes = generations * neat\_population$. This is used when setting the different termination criteria for the different algorithms in the experiments.

Plots All experiments have been run ten times, except for DL+SAC, which is only run five times due to the time constraint of the master thesis, and there was simply no time to complete ten runs. To visualize the thesis results, a 95% confidence interval and the mean over ten runs are also added to each of the plots in addition to the maximum over all runs. NEATs plot is created using the maximum fitness at each generation over ten runs, and the confidence interval is created from the ten best-performing individuals at each generation. The maximum values are used in these plots because and not the population average, in a complex problem, the new individuals chosen in the selection process of NEAT have no guarantee of performing as well as their parents. This process will lead to a large portion of the new individuals not producing interesting and good-performing solutions.

Gaussian filter

In the experiments involving reinforcement learning, there are a lot of fluctuations in the data, along with a vast amount of data points, which results in plots that are extremely hard to analyze visually. In these cases, a Gaussian filter has been applied to the data; this is a smoothing or filtering process that involves averaging neighboring data-point. Doing this makes it easier to see trends and analyze the results, but it can also result in losing some information. Figure 4.3 displays one run with SAC, where the normal data is quite chaotic, but by applying Gaussian filters with different sigma(standard deviation for the Gaussian kernel used) values,

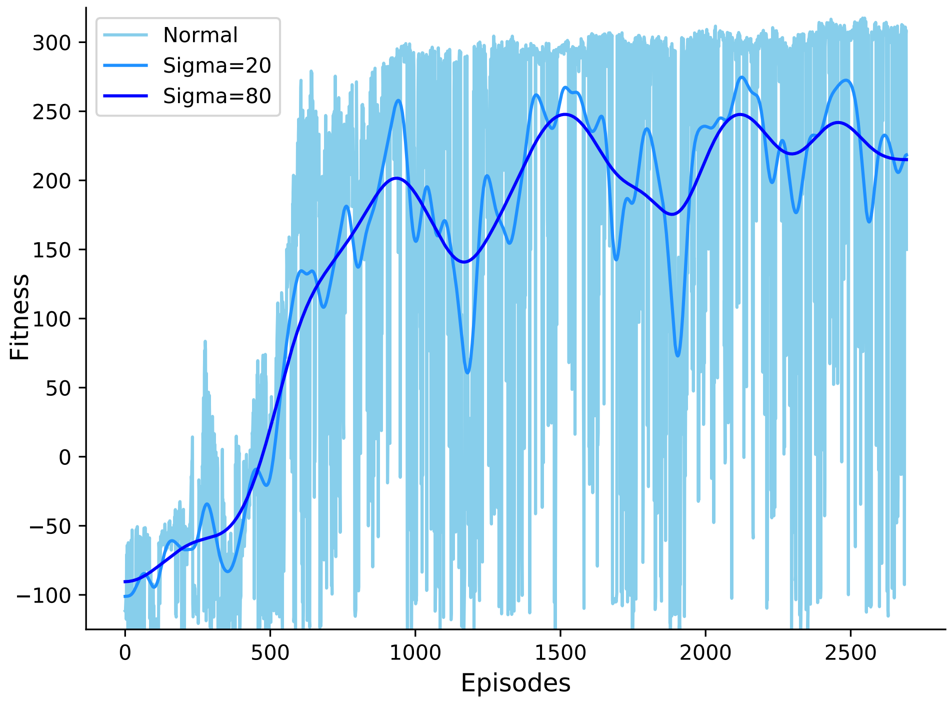


Figure 4.3: Gaussian Filter example

the performance and trends of the algorithm are easier to see. However, some of the higher values are lost; for example, at 900 episodes in figure 4.3, the achieved fitness is 200 for sigma=80 line, 250 for sigma=20 line, and 300 for the normal data.

In all experiments involving RL, two for NEAT, this Gaussian filter has been applied. The sigma values are reported in the plots, and the raw version showing the original data is supplied in the appendix C.

4.5 Experimental Challenges

A few challenges quickly appeared during the preliminary experiments and initial testing of the main experiments. These challenges were mostly concerning the time it took to run a single experiment, primarily with images. In order to produce some results in time for the delivery of this thesis, these challenges needed solutions or at least optimizing parts of the problem to work more efficiently. This section focuses on the problems and solutions necessary for completing the result of the thesis.

Problems and solutions

Rendering The rendering function in the bipedal walker environment draws the environment at every time step, and this is important so that the algorithm can get information about the effects of the last action. Extracting these images, as done with the image experiments, took a lot of time and required a lot of computing power. This drawing and extraction method is part of the bipedal walker environment, and changing this would mean changing a big portion of the environment.

Different approaches were tried to solve this problem, using various methods and python packages, but the results were not as significant as hoped. Two things were optimized, (1) Stopping the creation of a new window at every time-step and (2) using a virtual-frame buffer instead of the computer screen.

Keras Using the pre-trained Keras model together with NEAT turned out to be challenging. Keras models do not support parallelization and being shared among several workers. Initially, all the workers in the algorithm were put in a wait queue and halted. To use the parallelization of NEAT, the DL model had to be loaded separately for each individual in each generation. This resulted in a considerable amount of time being spent solely on initializing the models and algorithms at each generation (200 seconds per generation). Another problem with Keras is that the very slow prediction method, 0.035 seconds per predict multiplied with the number of time-steps, maximum 1600, resulted in worst-case 40+ seconds per individual per generation.

However, there was no straightforward solution to this problem, but this led to discovering other ways of optimizing the usage of the DL model. At each time step, a screenshot is captured and sent into the DL model, and Keras supplies a "predict" method for this purpose. But this turned out to be quite slow, 0.035 per predict and multiplied with the number of workers, population size, and generations result in a huge amount of time. Another method is simply sending the screenshots through the model, skipping the prediction method. This resulted in 0.025 seconds per prediction, which, when multiplied for an entire run, leads to a significant speedup.

EA vs. RL Comparing EA and RL can be challenging, with the core algorithmic differences, NEATs population-based search versus SACs singular global optimum search, parallelization versus not parallelized. With NEAT being one of the more important elements of this thesis, it was natural to start experiments using NEAT. However, when multiple experiments were done with NEAT, transferring the same criteria, generations, and hyperparameter to SAC resulted in weeks and months of run time for SAC. With 100 generations translating to 20000 episodes, and the maximum number of generations 2500 equals 500 000 episodes.

Allround Solution

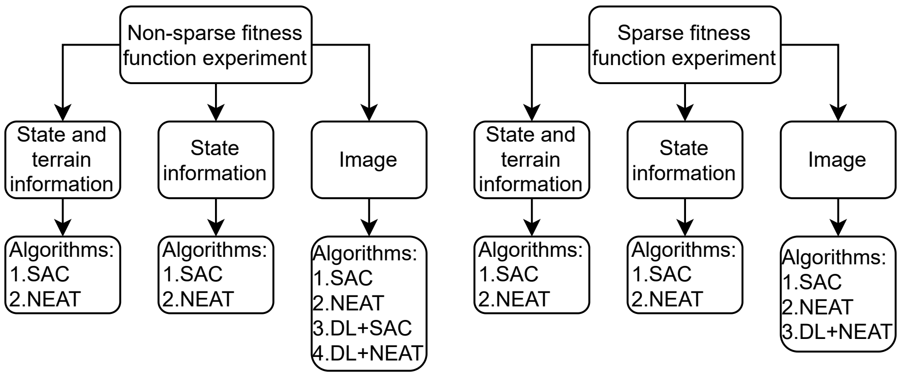
With a large number of complex experiments and no simple, not time-consuming solutions, the termination criteria were revisited, and techniques for optimizing the agent in the environment were examined. This resulted in adding a no-improvement threshold of 100 generations or 20000 episodes, in addition to the already present criteria, threshold, and a maximum number of generation. The maximum number of generations was originally set to be 7500 but was lowered to 2500, both concerning the above challenges and by looking at similar research.

The modifications done to the agent/environment have already been described in section 4.1, but the most important element was scaling the time-steps down from 1600 to 1100 per run. And finally, adding more computational resources helped a lot, starting with only the personal laptop and ending with the resource described in section 3.3

Chapter 5

Experiments and results

In this chapter, the experiments and the results will be presented. Each section has a motivation, a plot visualizing the results, and analysis or short description of the result, the full overview of all experiments is shown in figure 5.1. As mentioned in the previous chapter, the main goal of the experiments is not necessarily to solve the environment but rather to see if the algorithms can learn and improve from the given inputs and different fitness functions. The code used in all these experiments is open-source and supplied on github.



Non-sparse fitness function experiment
State and terrain information
State information
Image
Algorithms: 1.SAC 2.NEAT
Algorithms: 1.SAC 2.NEAT
Algorithms: 1.SAC 2.NEAT 3.DL+SAC 4.DL+NEAT
Sparse fitness function experiment
State and terrain information
State information
Image
Algorithms: 1.SAC 2.NEAT
Algorithms: 1.SAC 2.NEAT
Algorithms: 1.SAC 2.NEAT 3.DL+NEAT

Figure 5.1: Overview of the experiments presented in this chapter

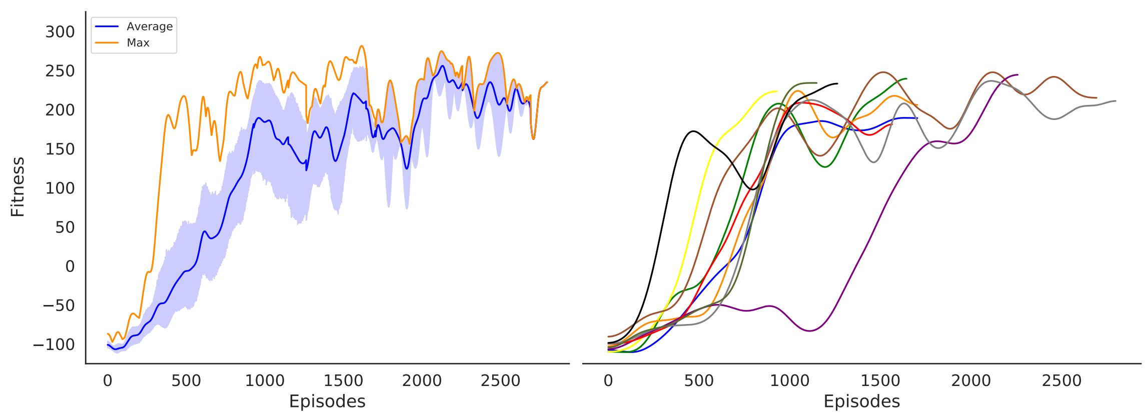
5.1 Non-sparse Fitness Function

This section aims to investigate whether the algorithms can solve the original environment with the non-sparse fitness function. The focus here will mainly be to get an overview of how the algorithms perform with various input types, starting with normal state and terrain sensor values, clipped sensor values with only state information, and then finally, images. These results will indicate if the termination criteria are balanced, if the algorithms are able to solve this environment and the general robustness and limitations of the algorithms.

5.1.1 Sensor Values

5.1.1.1 State and Terrain Information

This experiment provides the algorithms with the original state and terrain sensor outputs from the environment, displayed in 3.1. These values give the agent information about its velocity, state of joints, and through the lidar values, terrain information. This experiment aims to see how well the different algorithms perform in the most straightforward version of the environment, using the non-sparse fitness function. The results can indicate if extensive hyperparameter tuning is needed and if the different termination criteria are adequately balanced according to the algorithms and the environment.

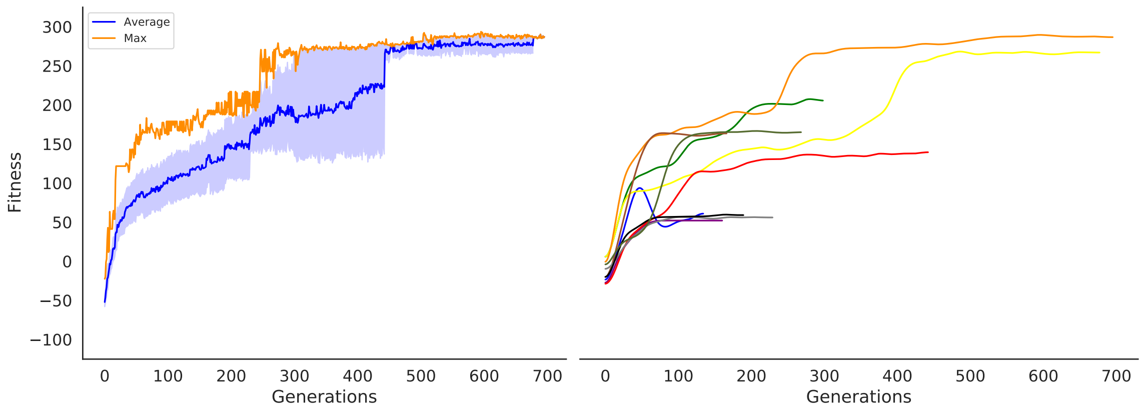


(a) Mean and confidence Interval (b) Single Run

Figure 5.2: **RL - State and terrain information - Non-sparse**
A Gaussian filter smooths out the graph due to the high fluctuations in the data to more accurately get a sense of the actual improvement of the algorithm, with a sigma value of 80 in (a) and 20 in (b). Plot (a) displays the confidence interval over ten runs, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. The ten single runs can be seen in plot (b), showing that all runs are able to solve the environment fast and reaches the threshold termination criteria.

Result

Using state and terrain sensor data and the original non-sparse fitness function, SAC is able to solve the environment quickly and consistently by reaching a fitness value of 300+. Figure 5.2 displays the accumulated fitness over the episodes and is divided into plot, mean, confidence interval and max in figure 5.2a and the ten separate runs in plot 5.2b. SAC solves the environment in 1000-2500 episodes or 5-12.5 generations and is able to solve it in every run. Although it might not look like it achieves 300 in fitness, it is due to the Gaussian filter, explained in section 4.4, which smoothes out the graph. (more details and raw plots can be found in the appendix C)



(a) Mean and confidence Interval (b) Single Run

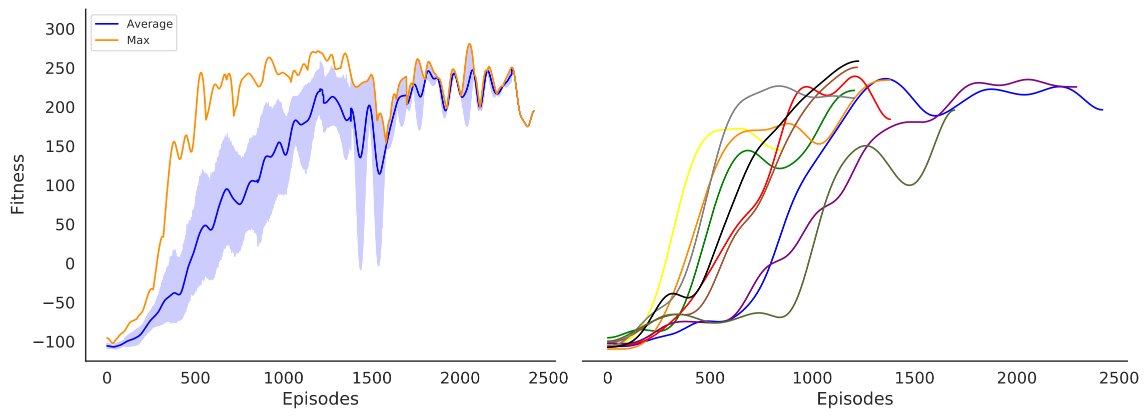
Figure 5.3: **NEAT - State and terrain information - Non-sparse Reward**
Plot (a) displays the confidence interval over ten runs, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. The maximum plot is the max of the max at each generation. From plot b, it is apparent that most of the runs got stopped with the first termination criteria, which is a non-improvement criterion over 100 generations. Plot (b) displaying the ten single runs has used a Gaussian filter, with a sigma of 10.

Result

Using state and terrain sensor information and the non-sparse fitness function, NEAT is able to perform well and accumulate a high fitness value, although not reaching the "solve" threshold fitness value of 300+. The figure 5.3 displays the accumulated fitness over the generations and is further divided into plot 5.3a with the average, confidence interval and maximum of the best performing individuals at each generation. Plot 5.3b displays the ten runs independently with the best performing individuals at each generation. From these plots, it is apparent that NEAT tends to get stuck at certain fitness levels, 50 and around 150 in particular, and with this short non-termination criterion of 100 generations, it is not always able to improve beyond this.

5.1.1.2 Only State Information

In this experiment, the last ten sensor values are not supplied to the algorithm, only the agent's state. This experiment is conducted to examine whether NEAT and SAC are still able to perform well in the environment without the lidar values providing information about the terrain using the non-sparse fitness function. This experiment is essential because these 14 values that provide agent-state information used in this experiment are the same values predicted by the DL model used in later experiments. The results from these experiments may indicate if a combination with DL is possible.



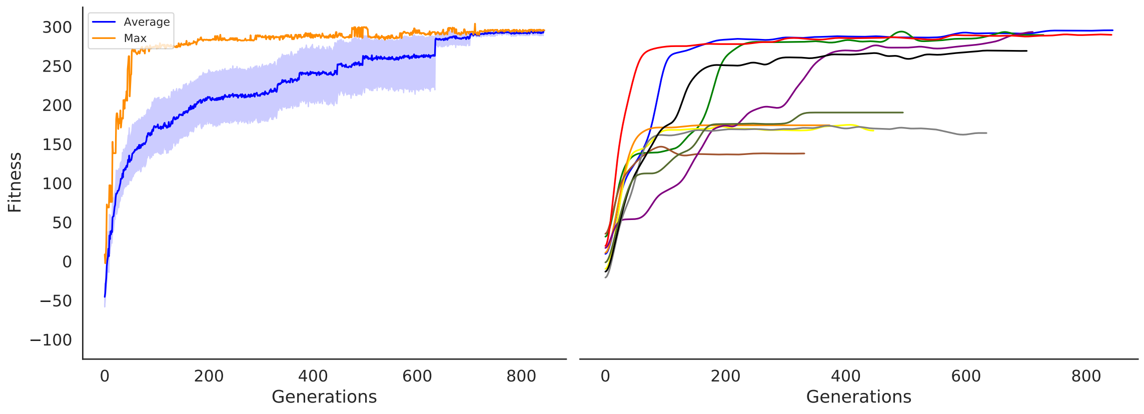
(a) Mean and confidence Interval

(b) Single Run

Figure 5.4: **RL - Only state information - Non-sparse Reward**
Gaussian filter with a sigma value of 80 in (a) and 20 in (b). This is the reasons why it is difficult to see that SAC reached a threshold of 300. Plot (a) displays the confidence interval over ten runs, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. Plot (b) displays the ten single runs.

Result

In this experiment, SAC was provided with agent state sensor values, and figure 5.4 displays the accumulated fitness over the episodes in two separate plots: (1) Plot 5.4a displaying average, confidence interval and maximum fitness over ten runs and (2) plot 5.4b displaying the ten runs separately. The performance in this experiment is similar to the previous state and terrain sensor experiment in that SAC is still able to solve the environment quickly and consistently at each run. However, the average solutions to the environment are found slightly faster with these sensor values, with seven runs completed in 1250 episodes (6.25 generations) vs three runs completed using state and terrain information. (more details and raw plots without Gaussian filter can be found in the appendix section C).



(a) Mean and confidence Interval (b) Single Run

Figure 5.5: **NEAT - Only state information - Non-sparse Reward**
The single plot shows that a couple of runs get stuck around 150 in fitness, but red manages to reach the threshold value of 300+. Plot (b) has used a Gaussian filter, with a sigma of 10. Plot (a) displays the confidence interval over ten runs (shaded area), the mean (blue line), and the maximum (dark orange line). Plot (b) displays the ten single runs.

Result

In this experiment, NEAT is provided with only sensor values with the agents' state and a non-sparse fitness function. The performance is displayed in figure 5.5 consisting of plot 5.5a displaying the average, confidence interval and maximum of the best performing individuals in the population at each generation. Plot 5.5b displays the performance at the ten single runs. From these plots, it is apparent that NEAT continues to perform well in this environment and is able to accumulate high fitness values. Comparing the performance of this experiment with the previous state and terrain experiment, it seems that NEAT performs better with fewer sensor values, and NEAT managed to solve the environment with a fitness of 304 in one of the runs. Furthermore, NEAT does not seem to get stuck at the lower 50 fitness step as it did in the previous NEAT experiment.

5.1.2 Image Values

In this experiment, the values supplied from the environment are no longer sensor values but extracted images from the environment. The reasons why this experiment is conducted are twofold; (1) Can the algorithms solve this problem directly from pixel-values, and (2) How well do they work in combination with DL?

The underlying goal is that NEAT and SAC will attempt to solve the problem with higher-dimensionality and directly from pixel-values. A down-sample (from 128x128x1 to 32x32x1) option was supplemented to help to solve this problem for NEAT at first because it is known to struggle to work effectively with high-dimensional data and later to SAC after running the with the original full-size image. The second test, when combined with DL, can indicate how well the DL model actually predicts the sensor values and how well the algorithms can perform even if the predictions are not perfect.

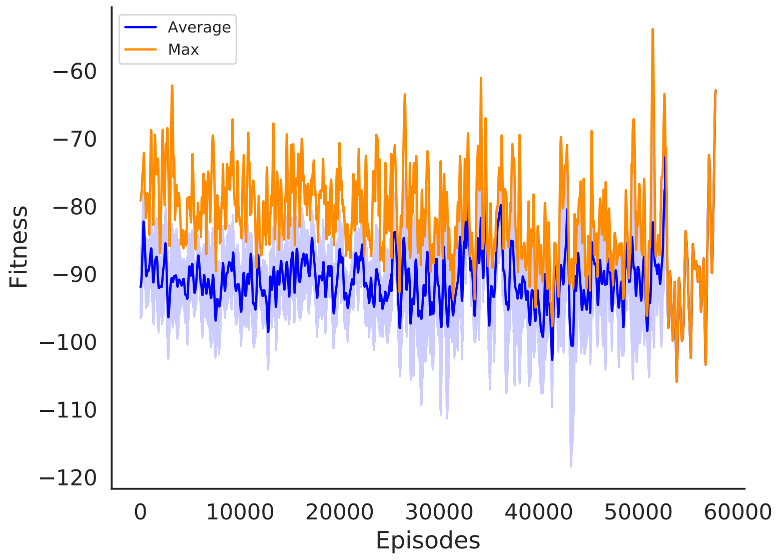
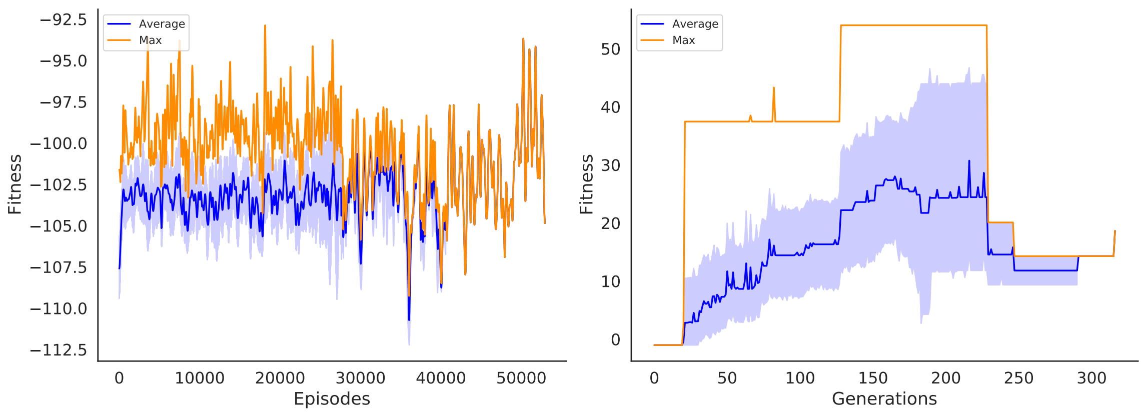


Figure 5.6: **RL - Image values - Normal reward - Full size**
The plot displays the confidence interval over ten runs, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. This graph shows that the algorithm struggles to improve with this input, averaging its accumulated fitness around -100. The best performing values are the yellow line, showing that even the best run did not achieve high results. As with all RL experiments, a Gaussian filter was applied here to smooth out the graph so that it is easier to see the actual performance of the algorithm (sigma=80).

Result

SAC appears to have a difficult time learning and improving using the full-size image with the pixels as input values and with a non-sparse fitness function. Figure 5.6 shows that the accumulated fitness is averaging around -95 and that the non-improvement threshold of 20000 episodes ultimately stops SAC. (raw plot, without Gaussian filter, see appendix C). From the plot, it is apparent that there are no tendencies of learning using this input, as seen in previous experiments.



(a) SAC (b) NEAT

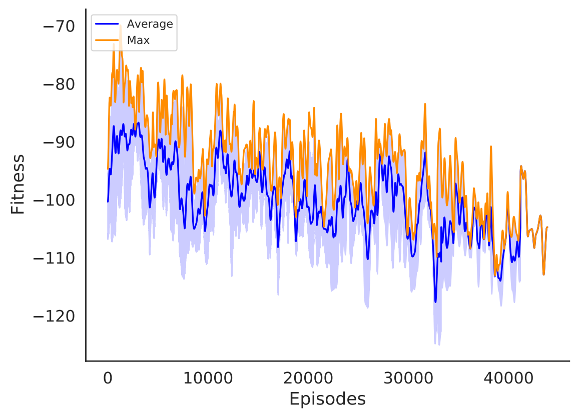
Figure 5.7: **NEAT and RL - Image values - Non-sparse Reward**
The plots displays the confidence interval over ten runs, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. In this experiment the input image was down-sampled using SciPy [59], from 128x128x1 to 32x32x1. The figure shows both SAC and NEAT using this down-sampled image as input, and this shows that SAC performs badly with this input, while NEAT is able to accumulate a positive fitness. In plot (a) a sigma=80 was used.

Result

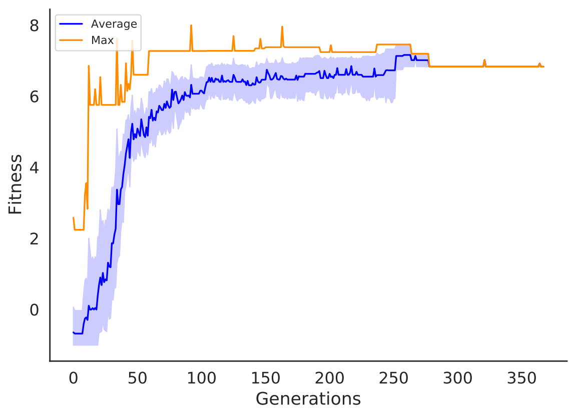
In figure 5.7 a down-sampled image was used as input, with NEAT in figure 5.7b and SAC in figure 5.7a. The down-sampling of the image should help both algorithms to improve more efficiently with the decreased dimensionality. However, SAC (a) struggles with this experiment as well as with the full-size image and is not able to improve successfully. NEAT (b) manages to reach positive fitness levels and proves that it can accumulate some fitness and improve using this input. From NEATs plot, it also appears that the confidence plot is a lot wider in these experiments than with the two previous experiments, and also that it seems to improve steadily before the no-improvement threshold stops it.

Result

From figure 5.8 showing the results from a DL+SAC combination in figure 5.8a and a NEAT+DL combination in figure 5.8b, it is apparent that both algorithms struggles to improve effectively with this combination. However, there is a difference between the performance, with SAC struggling to improve at all, NEAT manages to accumulate a positive fitness value, which relates to it learns how to move slightly in the right direction in the environment, without falling. NEAT seems to have a steady increase in fitness over the generation. Although it is slow, it has tendencies to keep increasing.



(a) Dl+SAC



(b) DL+NEAT

Figure 5.8: **Image values - Non-sparse Reward - NEAT and DL**
Plot (a) was run five times and plot (b) was run ten times. These plots displays the confidence interval, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. These plots display a comparison of both SAC and NEAT combined with DL, with the DL model receiving the extracted image from the environment and predicting agent state values, which should be the same 14 agent state values provided by the environment. The plot shows that both algorithms struggle to improve effectively together with this DL model, but from plot (b), it can be seen that NEAT successfully accumulates a positive fitness. In plot (a), a sigma of 80 were used.

5.1.3 Summary - Non-Sparse Reward

The three previous experiments have used the original non-sparse functions, and the agent has received different types of information from the environment. From the first experiment with the original state and terrain information, both algorithms performed very well, with SAC being able to solve the environment quickly and consistently in every run and NEAT reaching high fitness values. In the second experiment, the terrain information or lidar values were not given to the agent, only agent state information. The results showed that SAC solved the environment slightly quicker, and NEAT saw a significant increase in performance and on average fitness over all runs. The last experiment gave the algorithms extracted images from the environment, either directly, down-sampled, or run through a DL model that attempted to extract the agents' state information from images. With this experiment, the complexity was significantly increased. This could be seen in the results and ended in lower performance overall. SAC did not perform well with a full-image, down-sampled image, or in combination with DL. It appeared that it struggled to learn anything at all. NEAT was not run with the full-size images due to the time it would take to complete such an experiment and that it is known to struggle to improve with large input sizes. NEAT performed well with the down-sampled image as input and accumulated small positive fitness when combined with DL.

5.2 Sparse Fitness Function

This section focuses on the performance of the algorithms when switching the fitness function from the normal non-sparse fitness function to a sparse fitness function, which is described in detail in section 4.1.1. The experiments are similar to the previous section, but the goals are now different. The focus of this section will be to see how the different algorithms perform on a sparse problem with the same input values provided as in the previous experiments.

The potential positive fitness values are small with the new sparse fitness function, while the negative values are still -100. Negative values are therefore normalized to -1 to simplify visualization. In this section, the fitness values of the y-axis are changed from the original scale to the fitness reward scale; please see section 4.2 for details on the sparse fitness function.

5.2.1 Sensor Values

5.2.1.1 State and Terrain Information

This experiment returns to the state and terrain sensor value experiment but with the new fitness function. The goal of this experiment is twofold, (1) Test whether SAC and NEAT can learn and improve with the new fitness function, (2) test different levels of sparseness. The latter is an experiment mainly to set a sparse threshold for later experiments, but also interesting to see where the algorithms start to struggle.

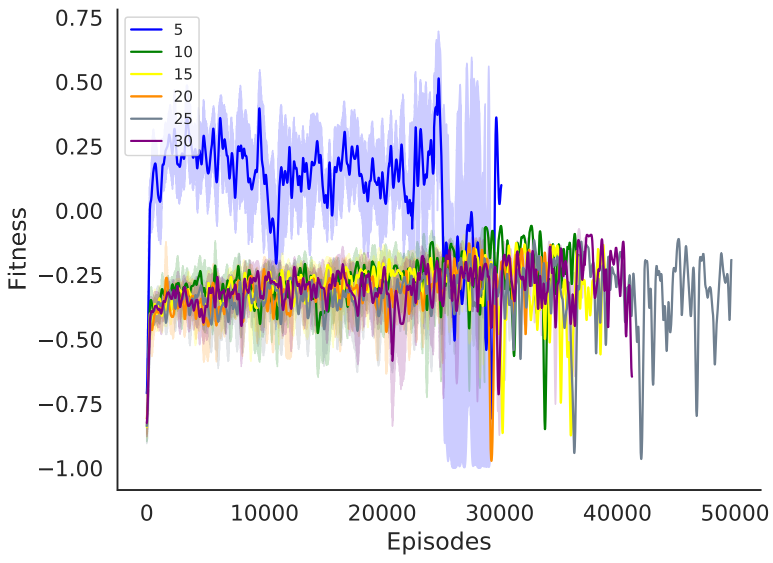


Figure 5.9: **RL - State and terrain information - Sparse Reward**
This plot displays the confidence interval (shaded area) over ten runs at different sparse fitness levels and the average as a line, over ten runs, using SAC. Gaussian filter was applied, with a sigma of 80. Due to a lot of fluctuations, it is hard to see what is going on here, but the most important thins is that the average lies around -0.25 for all runs, except for sparse-level 5 which has a average of 0.25.

Result

In this experiment, state and terrain sensor information is provided to SAC, and the environment has a sparse fitness function. Several different sparse start threshold values are tested to see the potential of SAC with sparse rewards. From figure 5.9 it is apparent that SAC struggles, in general, to improve and learn using a sparse reward function, regardless of the starting threshold. The Gaussian filter removes some information here, indicating that SAC is able on the best run to accumulate some fitness at the lower sparse levels (raw plots in appendix section C), but on average seen in this plot, it struggles. The plot also shows that with five as the sparse threshold, it accumulates positive fitness at the beginning of the run but slowly decreases throughout the runs. It is also important to note that the initial starting x-position of the agent is 4.66 on average, meaning that the agent only have to extend a foot to receive the first reward when using 5 as the sparse threshold.

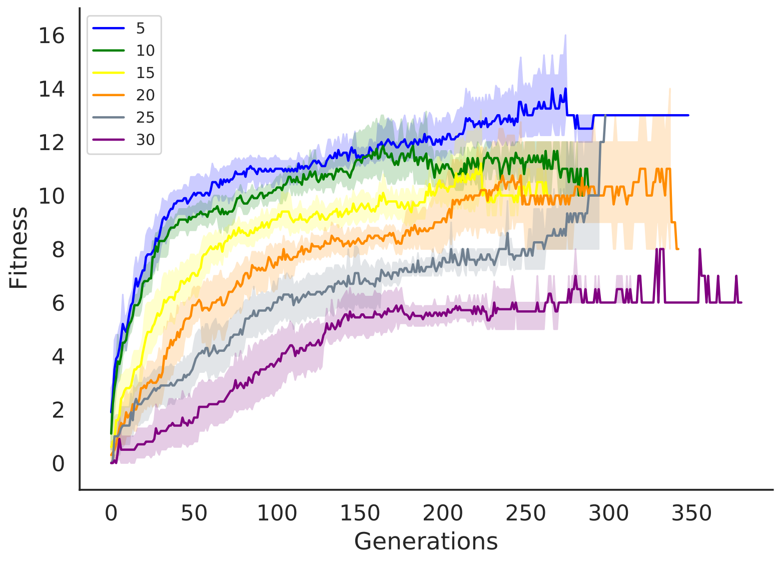


Figure 5.10: **NEAT - State and terrain information - Sparse Reward**
This plot displays the confidence interval (shaded area) over ten runs at different sparse fitness levels and the average as a line, using NEAT. It is able to learn and improve using all different levels of sparseness that was tested.

Result

In this experiment, NEAT is provided with state and terrain sensor information but with a change in the fitness function from non-sparse to sparse. NEAT is tested with six different start thresholds, correlating to how far it must walk to receive its first reward. The performance is plotted in figure 5.10 and shows that it is able to learn and improve with all different sparse levels. It performs better and learns faster with lower sparse levels, and with a quite narrow confidence interval, it is also apparent that it is pretty consistent in how it performs with the different values. The maximum achievable fitness is naturally correlated with the start threshold and can be seen in table 4.1. (more details and singular plots for all runs, with non-sparse fitness plots, see appendix section B.1)

5.2.1.2 Only State Information

In this experiment, the agent is given only state information and the sparse fitness function. The sparse level is set to 10 to let both algorithms have a genuine possibility of succeeding in this environment, based on the results from the previous experiment. The sparse level 10 is also chosen based on an the initial starting position of the bipedalwalker explained in section 4.1.1, that is around 4.66, so by choosing 5 as sparse level, would only require the bipedal-walker to extend a foot to receive a reward.

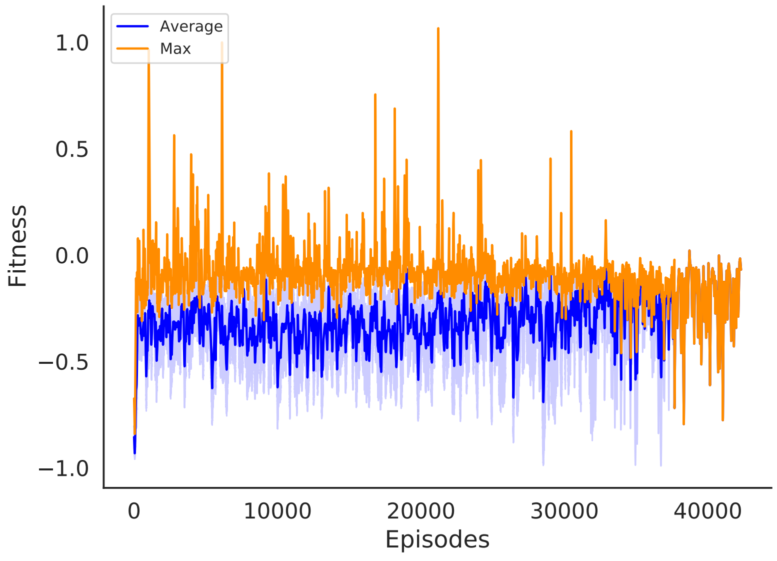


Figure 5.11: **RL - Only state information - Sparse Reward**
The plot displays the confidence interval over ten runs, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. As with the state and terrain sensory inputs, SAC continues to struggle with sparse rewards. It is, however, able to reach the first threshold at the beginning of the run, achieving a reward of 1 but only in the maximum performing run over ten runs. The average is below 0. A gaussin filter was applied here, with a sigma of 80.

Result

Using the agent's state information, a sparse fitness function, and a sparse start threshold of ten, SAC continues to struggle in general with sparse rewards. From figure 5.11 it can be seen that the fitness is averaging between -1 and 0, meaning that it is altering between falling and not doing anything. In other words, it is not able to learn and improve effectively using this fitness function. From the maximum values in this plot, it appears to be able to accumulate a high fitness level initially, yet cannot effectively utilize the provided information and reward function and ends up it losing the fitness towards the end, this effect can be seen better in the raw versions supplied in the appendix section C.

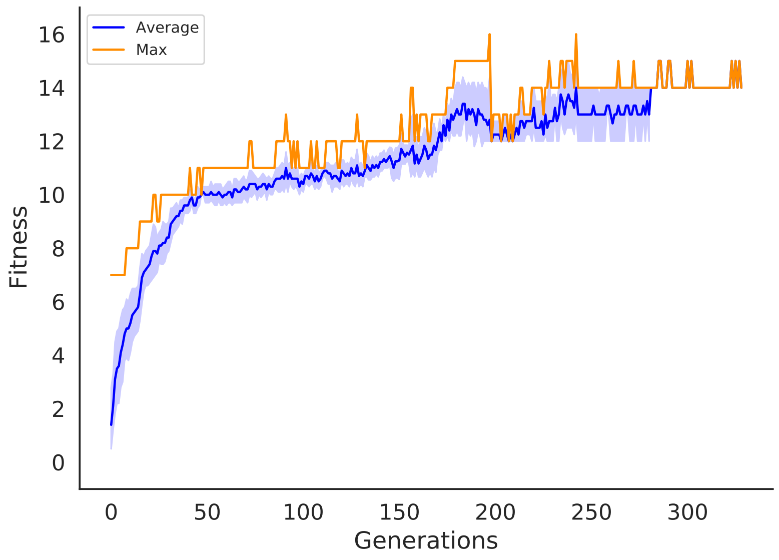


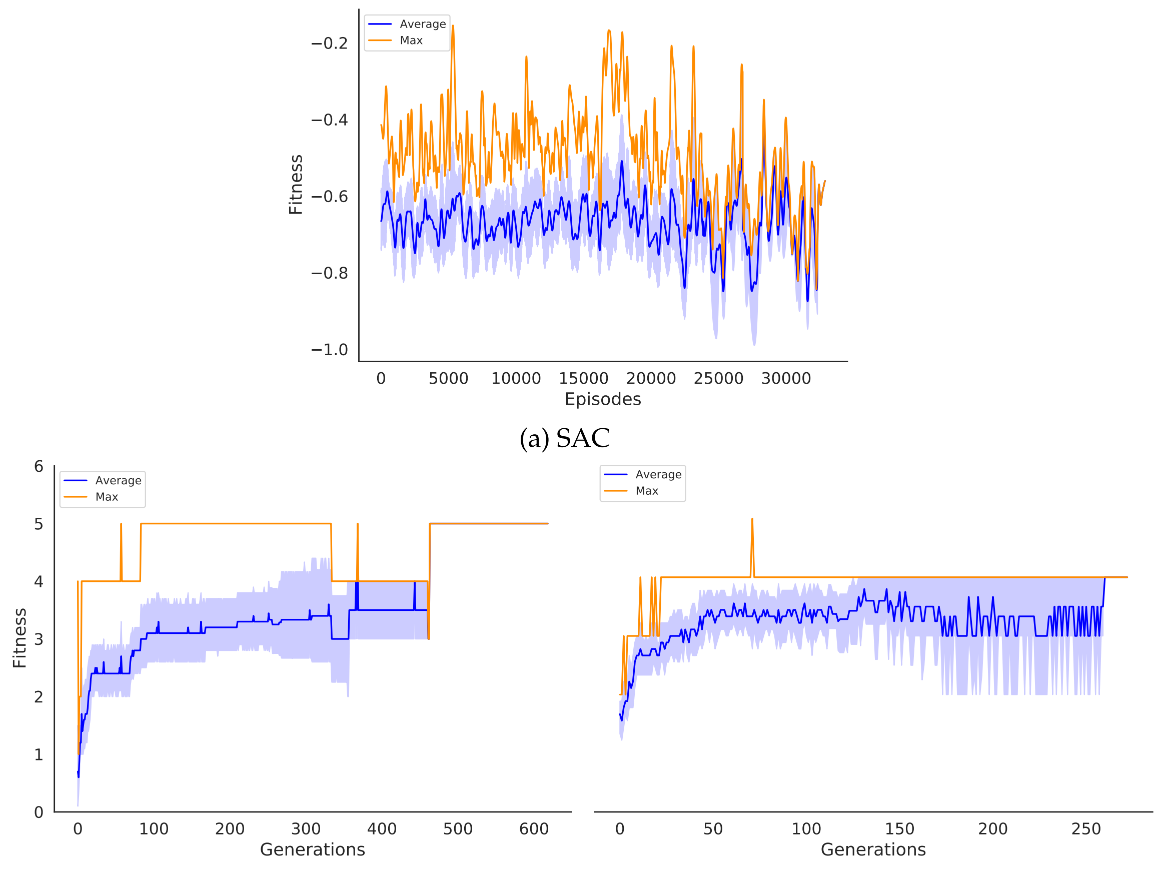
Figure 5.12: **NEAT - Only state information - Sparse Reward**
The plot displays the confidence interval over ten runs, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. NEAT is still able to perform well with the agent state sensor input, even reaching the maximum fitness of 16. Additionally, this sparse fitness function has a much narrower confidence interval than normal fitness functions.

Result

Figure 5.12 shows that NEAT is still able to perform well when only given agent state information and using the sparse reward function. When comparing this performance with the previous experiment in figure 5.10 with sparse level 10, it seems like it performs slightly better in this experiment, even solving the environment by reaching the threshold value of 16. The same difference was apparent in the non-sparse reward experiments with the same two values in the first two experiments in this chapter. With this step size approach, one can also see that the confidence interval is a lot smaller than it was with non-sparse fitness function.

5.2.2 Image Values

The final experiment combines all previous experiments, a sparse reward function, with images extracted from the environment. This experiment results in a problem with high-dimensional data with sparse rewards. The experiment's goal is twofold, (1) Can the algorithms solve this problem? Furthermore, (2) How do they compare to the other algorithms. In these experiments, a sparse level of 10 was used, a sparse level that requires some walking before it receives a reward but is not too hard. Hopefully, choosing a lower sparse level can help the algorithms perform better with the more complex problems.



(a) SAC

(b) NEAT

(c) DL+NEAT

Figure 5.13: **Image values - Sparse Reward - NEAT, DL and SAC**
These plots displays the confidence interval over ten runs, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. NEAT with and without a DL component is performing quite similarly from this plot. SAC is struggling with this problem, averaging between 0 and -1. NEAT was using a down-sampled version of the image and was allowed to run for 250 generations without improvement as opposed to only running 100 on all other experiments. In figure 5.13a a sigma of 80 were used.

Result

Figure 5.13a, shows that as with the other sparse reward problems, SAC on average stays between 0 and -1, meaning that it varies between falling and not creating a walking pattern that can reach the first threshold. However, these results are not surprising by looking to the previous experiments, with images and sparse rewards, where SAC also struggled. From figure 5.13b, NEAT is using a down-sampled image as input, and through this is able to reach similar fitness values as with DL+NEAT, which is displayed in figure 5.13c. Both were performing well with images and sparse rewards. Using this down-sample option makes NEATs performances comparable with DL+NEAT, but DL can handle a vast increase in dimensionality, resulting in a considerable decrease in NEATs performance if it were to use the same input-dimensionality as DL. This increase would most likely result in NEATs performance decreasing and would require a vast amount of computing power to compare to a DL+NEAT combination on the same problem. NEAT alone was allowed to run for longer by extending the non-improvement threshold from 100 to 250 generations.

5.2.3 Summary - Sparse Rewards

The three previous experiments have been run with the same information given to the agent as the first three experiments, state and terrain, only state, and images. The difference in these experiments is that the fitness function was switched to a sparse fitness function. In the first experiment, involving state and terrain information, the goal was to test different levels of sparseness and see how the algorithms performed. SAC struggled with most of the experiments involving sparse rewards but managed to accumulate rewards with the lowest level of sparseness. On the other hand, NEAT performed well on all levels of sparseness, even completing the environment in some cases. From the second experiment involving only state information, and a sparse level of 10, SAC continued to struggle. NEAT performed slightly better, a steeper learning curve with only state information than state and terrain information. In the last and final experiment, the sensor values were exchanged with images from the environment. SAC continued to struggle with using images as input and a sparse reward function. NEAT performed well using a down-sampled image and reached the same fitness levels as the DL+NEAT combination. NEAT alone were allowed to run for 150 additional generations to see the potential and ended up performing equally to DL+NEAT.

Chapter 6

Discussion

This chapter aims to discuss the results presented in the previous chapter in light of the goals of the dissertation. This chapter's format is linked directly to the proposed milestones presented in the introduction to this thesis. The final two sections address the limitations and a discovery made during the experiments.

6.1 General discussion

In the background section, a thorough survey of the related field of NE+DL was conducted. This research is highly promising and has achieved excellent results in various complex environments. Utilizing different algorithms and approaches, but with the core motivation being the same, DL handles the high-dimensionality, and a controller is evolved using neuroevolution for choosing actions. This completes the first goal G.1 of the thesis.

The results presented in this thesis show that a DL and NE combination performs better on high-dimensional, sparse reward problems and consequently completes the second goal G.2. All milestones were reached and will be described in the subsequent sections.

6.2 Comparing the results of the algorithms with a non-sparse fitness function

The results show that SAC is superior to NEAT in the non-sparse environment, given information about the agents' state and terrain information. SAC is able to complete the environment in 1000-2500 episodes, or 5-12.5 generations, by reaching the fitness threshold in all conducted runs. While NEAT is performing well, it does not reach the same levels and has the tendency to get stuck at certain levels, and is stopped by the non-improvement termination criteria in 200-700 generation. This leads to two interesting questions that need to be discussed further: (1) Why is SAC that much faster than NEAT, and (2) Why does NEAT get stuck?

In order to answer the first questions, it is helpful to take a closer look at one of the differences between RL and EA in an RL problem setting and the use of a value function. The value function estimates how good it is for the agent to be in a state, which many RL algorithms uses and estimates. Value function methods such as SAC allow for individual states and actions to be evaluated, thus increasing the effectiveness of an optimal policy search. However, this is not strictly necessary to solve RL problem, where EAs is a prime example. EAs ignores much of the useful structure of an RL problem, and this includes: "they do not use the fact that the policy they are searching for is a function from states to actions; they do not notice which states an individual passes through during its lifetime, or which actions it selects." [56]. Taking NEAT as an example in the bipedal-walker environment, a population of 200 candidate solutions is kept, all individuals are evaluated by performing actions and receiving rewards and feedback from the environment. The best individuals are chosen as parents, thus pushing the population towards higher performing individuals. The problem with this evaluation method is that it only considers the final accumulated reward, not the individual actions or states that led to that solution. This, in practice, means all actions taken by a well-performing individual receive high credit, although these actions might be non-optimal or unnecessary. In other words, SAC is learning while interacting with the environment, which NEAT does not.

The second question is why NEAT tends to get stuck at some levels, and an interesting observation can be seen when looking at the visual solutions of these algorithms. Solutions produced by SAC have a more vertical, running pattern, while NEAT often produces a very stable walking pattern, dragging one leg behind as support and using the other to move forward. With speed being one of the main factors for solving the environment, this is an important observation, with NEAT often choosing more stable, secure solutions, while SAC moves faster and more optimally, but with a higher risk of falling. In order to accumulate higher rewards, NEAT has to produce a more upright walking pattern, thus minimizing friction and increasing the walking speed, similarly to SAC, but this also requires more balance. Altering this behavior might, for a time, lead to a substantial decrease in fitness through more falling, which is significantly penalized, and at least in the short termination criteria in this experiment, it struggles to optimize towards this behavior. SAC does not seem to have the same problem with getting stuck and the reason for this is the same theory described in the previous paragraph, with SAC interacting more with the environment with continuous feedback and analyzing each action. This results in greater agent control in the environment, knowing what to do and not to do at every state, and can choose optimal actions at each time step.

The results, which complete the first milestone M.1, show that both algorithms are able to perform well in the environment with the non-sparse fitness function. SAC and NEAT are both able to learn and improve effectively, but with SAC being superior in this experiment.

Running Time: The actual time required to run these experiments is quite similar. With SAC being faster at solving the environment when looking at episodes and generations, but NEAT can use parallelization to speed up the actual running time greatly. Therefore, the comparison of real-time usage is not included, as both algorithms can use different hardware to speed up the algorithm, cores for NEAT, and GPU for SAC, and a definite comparison would not be fair.

6.3 Investigating possibilities of a DL combination

It is apparent that removing the lidar values from the standard version of the environment did not result in a decrease in performance, but the opposite. With SAC performing slightly better than it did in the previous experiment, still being able to solve and reach the threshold value of the environment quickly and consistently. NEAT performs better with only state sensor inputs, achieving higher overall fitness over the separate runs, with a narrower confidence interval and a steeper learning curve. One of the reasons why this occurs is that NEAT thrives and performs better with low-dimensionality, simplifying the optimization considerably with fewer nodes and connections. In this experiment, NEAT also manages to solve the environment in one of the runs, reaching a threshold of 304. NEAT also does not get as easily stuck at lower fitness values as it did with the previous experiment.

With the experiment being quite similar to the one discussed in the previous sections, a lot of the same theory on the difference in performance between SAC and NEAT applies to this experiment as well. SAC being superior to NEAT, but NEAT performing better than in the previous experiment.

The results in this experiment accomplished two goals; one, NEAT and SAC proceeds to perform well in this environment, confirming that in the standard version of the environment, as expected, the lidar values are not necessarily needed, although it can provide the agent some extra information. Second and most importantly, solving the second milestone M.2, both algorithms should perform well when combined with a deep learning model. The reason for this is that the values used in this experiment are the same as the DL model aims to predict, so if the DL model can predict values that are close to this, both algorithms should perform well.

6.4 Investigating the performance of the algorithms with high-dimensional input

Increasing the dimensionality of the problem, and switching to pixel values instead of sensor values, changed the performance of both algorithms. Three types of experiments were run with SAC, full-size image, down-sampled, and features extracted from the DL model, while only two experiments were run with NEAT, down-sampled, and DL with NEAT. Using a full-size image as input to NEAT was not feasible in the time frame of this thesis due to the high complexity and huge time requirement to run the experiment.

The DL model was responsible for receiving the image as input and predicting the agent state, and with a perfect model, these values should be the same as the state sensor values supplied by the environment. With a perfect model, the results with DL+SAC and DL+NEAT should be comparable to the experiments in the previous section with the state sensor inputs. However, this was not the case, and it seems that the values predicted by the DL model are too inaccurate for the algorithms to reach the same levels. SAC struggled greatly with the features extracted from images, while NEAT was at least able to reach a positive fitness of average 8, which is not walking but balancing and maybe falling forward on its front foot. This is further discussed in section 6.7.

SAC continued to struggle with both the down-sampled image and the full-size image. The reason it struggles with the full-size image was briefly mentioned in the background section, with DRL algorithms being known to struggle to explore and improve efficiently in high-dimensional environments. However, it may also indicate that SAC struggles with pixel-to-action mapping with it struggling with the down-sampled image as well.

NEAT performed well with the down-sampled image as input and achieved a positive fitness of over 50 with the best individual; this corresponds to the agent being able to walk a reasonable distance without falling. Through this experiment, it seems that NEAT is less dependent on the information from the environment, and this fits well with the claim that "Evolutionary methods have advantages on problems in which the learning agent cannot sense the complete state of its environment" [56].

The results, which completes the third milestone M.3, show that NEAT and NEAT+DL can accumulate positive fitness values using these high-dimensional image input values, while SAC seems to struggle with high-dimensionality, pixel-to-action mapping, and in combination with a DL model.

6.5 Investigating performance of the algorithms with a sparse reward function

6.5.1 State and terrain sensor input

The first experiment using a sparse fitness function led to some interesting discoveries and is promising in leading up to the final experiment. The results from the experiments using state and terrain information sensor values with a sparse fitness function show that NEAT manages to improve with different sparseness levels effectively and that SAC struggles to improve and learn effectively with these rewards.

NEAT performs very well with all the different levels, although the performance drops with an increase in sparseness. This is not surprising as the maximum achievable fitness also decreases in correlation with the sparseness. These results also indicate how NEAT manages through random mutations to solve different levels of sparse rewards.

The results show that SAC struggles with this fitness function by looking at the different sparse levels' confidence intervals. Although not present in the plot, in the best max performing runs of SAC actually at the lower sparse levels, SAC manages to accumulate a quite high-fitness initially, but it struggles significantly to improve this solution further effectively and ends up with 0 in fitness in the end. This is further described in the next section, with state information only, where the best run is included in addition to the confidence intervals.

These results support the claim that NEAT is good at solving problems with sparse rewards and that RL, in general, struggles with sparse rewards. These results and discussion and the next section also complete milestone M.4.

6.5.2 Only state sensor input

The format of this experiment was similar to an earlier experiment where the lidar values were removed, and the agent is only presented with sensory inputs about its current state but with a sparse reward function. From this experiment, the results show that NEAT still manages to improve and perform in this sparse reward setting, while SAC continues to struggle with sparse rewards, although some interesting behaviors emerged. In these experiments and the final experiments, the sparse level was set to 10, allowing for both algorithms to have a chance of performing well in this setting. This sparseness could have easily be set higher for NEAT to further increase the difference between the results of the algorithms.

The same results can be seen here: with the difference in state and terrain inputs and state inputs with the non-sparse fitness function, that NEAT performs better with fewer input dimensions, with a narrower confidence interval, steeper learning curve, and overall higher fitness levels. With only state sensor information and a sparse level of 10, NEAT is even able to solve this environment a couple of times successfully. These results show that NEAT is able to perform well with sparse rewards, using the state sensor information, which in turn means that in combination with DL predicting the same state values, it should yield a good combination. With NEAT choosing the actions in a sparse reward problem, while DL is responsible for extracting the features from high-dimensional images.

The results show that SAC continues to have problems with sparse rewards and is not able to perform on similar levels as NEAT. As mentioned in the last chapter, some information from the plots is naturally lost due to the use of Gaussian filters, with the raw plots supplemented in the appendix section C. The raw version of this experiment showing the best of the ten runs shows that SAC manages to quickly reach a fitness value of 10, which is relatively high with this sparse reward, but from there, this fitness declines steadily down to 0 after 40000 episodes. This implies that SAC struggles to improve with and properly exploit the reward function, and with the confidence interval being relatively low, it might be a random good performing individual that caused these high values. These results also indicate that an experiment with DL and SAC in a sparse, high-dimensional setting is not strictly necessary, with SAC struggling with this problem overall.

6.6 High-dimensional, sparse reward problem

In the final experiment, the focus was to achieve the second main goal of the thesis, which is solving a high-dimensional, sparse reward problem and investigating whether this combination is better than the separate algorithms on their own. Four important aspects need to be remembered when looking at these results, (1) DL can handle a vast increase in dimensionality, referring to the input image. This point will result in the differences being even greater when comparing with NEAT and SAC. (2) NEAT alone here was down-sampled with a factor of 8 in this experiment and was allowed to run for 150 extra generations (3) The DL model was not as accurate as initially hoped, so with better predictions, both SAC and NEAT would be able to solve the environment with normal rewards. (4) The sparse level was set to 10 to allow SAC to be competitive based on previous experiments but could be sat even higher in respect to NEAT.

A combination of DL and SAC were tested with sparse reward but not included. Due to the already large amount of plots and that the performance was poor, which is also supported by earlier experiments. It is not surprising that SAC continues to struggle with the experiment being both sparse and high-dimensional. However, SAC was able to reach a fitness level of 2 (seen in the raw plots without Gaussian filters, in the appendix 12c), in a few runs, but it struggles to improve the fitness beyond this, and often ends up at fitness of 0, at the end of the run. So it seems like this is not necessarily a learned behavior but can be a product of especially two things: (1) Random actions that turn out to give some good behaviors, and (2) the sparse level was perhaps a little bit too low and made it to easy to accumulate fitness.

NEAT performed well alone and in combination with DL and reached a fitness level of five in both cases. With the maximum fitness possible being 16, five equals walking some distance without falling. These results were achieved with a pretty low non-improvement of 100, resulting in the maximum number of generations that DL+NEAT were allowed to run to 250 generations.

This experiment was conducted to allow all algorithms to be competitive, which resulted in quite similar performances over all experiments., which was mentioned in the intro: (1) Low dimensionality, which DL could handle a massive increase of, (2) down-sample and longer non-improvement threshold for NEAT, and (4) low sparseness threshold. These three points are essential when discussing the results of this main experiment and whether a DL combo is better than the other approaches alone. The deep learning model can handle a huge increase in dimensionality, which for DL would not be very noticeable but would increase both the time, complexity, and results of the other experiments, and they would not be competitive with a DL combo just by looking at real-time training. As mentioned, the DL model was not as accurate as hoped, and with a more accurate model, the results would have been far greater, reaching similar levels as with the sensor values. NEAT was allowed to run an extra 150 generations and use a down-sampled image, which contributed to NEAT being competitive with DL in these experiments regarding results and training time. Without these optimizations, NEAT would have struggled, at least with a full-size image. Finally, the sparse threshold level was set to 10 to give SAC a chance to learn something in this experiment, but a far larger threshold (3 times) could be used, and DL+NEAT would still be able to perform well.

With all the different points and results discussed in this section, the second main goal G.2 was achieved.

6.7 DL Combination

With the environmental challenges explained and discussed in section 4.5, and section 6.8, some of the results involving the DL model did not achieve the same fitness levels as with the state sensory input experiment. With a perfect DL model, these fitness levels should have been equal to the state sensor input experiments in the same amount of generations/episodes, but with the results presented in both section 6.3 and 6.5.2 it showed that this was not the case. These results show that the values predicted by the DL model are too inaccurate for some of the algorithms to improve. However, this led to an exciting discovery when comparing the performance of SAC and NEAT using the same DL model and its inaccurate predictions.

From figure 5.8 from normal fitness function and the figure 5.13 using the sparse fitness function, it can be seen that with an "inaccurate" view of the environment, which is a problem in most complex RL environments, NEAT still manages to learn and accumulate a satisfactory fitness level, though not nearly as high as hoped. SAC and other RL algorithms are far more dependant on information and interactions with the environment and are therefore more sensitive to inaccuracy from the environment than EAs. Nevertheless, the most important is that NEAT manages to learn and overcome the challenges of an imperfect view of the environment, and if given more time, probably achieving even higher fitness values. SAC, on the other hand, is struggling with these imperfect predictions.

6.8 Limitations

A few of the limitations concerning the experiments conducted will be discussed in this section. Some of the challenges faced in this thesis were briefly explained in chapter 4, under Experimental Challenges, and how all the different factors that played in, especially with the image experiments and the usage of DL, resulted in very time-consuming experiments. With time being one of the biggest challenges, getting the results needed to finish on time resulted in some decisions that might have impacted the results of this thesis.

The non-improvement threshold was an essential factor in finishing the results. However, this might have also led to the algorithms not getting enough time to optimize for the more complex experiments. By looking at all the experiments, it is quite apparent that this non-improvement threshold was the termination criteria that stopped the algorithms most often for some of the complex experiments involving images and sparse rewards.

The deep learning model did not predict as accurately as hoped, resulting in not achieving similar fitness levels as with sensor values. There are several reasons for this; firstly, as mentioned, predicting velocity values from images is not possible, and therefore, a majority of the predicted sensor values were inaccurate. Secondly, as mentioned in the previous paragraph, the algorithms, or at least NEAT, did not get enough time to learn to work with inaccurate environment predictions. However, these limitations led to some exciting findings comparing NEAT and RL; when given inaccurate information about the environment, NEAT successfully overcame and learned to walk despite this.

With many complex, time-consuming experiments, other interesting aspects that could help on the performance of the algorithms or explore this NEAT+DL more thoroughly were left out due to the short time frame of a master thesis. Parameter-search was mainly left out, although a few parameters were experimented with during the preliminary results, which helped choose some parameters. NEAT having around 40++

parameters that could be experimented with and possibly helped improve the performance. In the original SAC paper, the authors claimed that the only hyperparameter that needed tuning was the reward signal, which would also have been interesting to experiment more with different sparse and normal reward functions and compare the performance across different reward functions. Lastly, exploring more environments with NEAT+DL to get an even more understanding of the strengths and limitations of this combination.

Chapter 7

Conclusion

The main goal of this thesis was to explore ways to combine deep learning and neuroevolution to get the benefits of both, without their weaknesses. This goal was split into two, with the first being to survey the existing field of DL+NE with a thorough comparison of the algorithms, approaches, problems, and representations. The second goal of the thesis was to investigate the performance of a deep learning and neuroevolution combination vs either of them alone, on high-dimensional, sparse reward problems. This was achieved by deploying a deep convolutional neural network as a feature extractor and an evolved controller using neuroevolution. The deep convolutional network functioned as a feature extractor, converting the high-dimensional data to a compact feature representation. The neuroevolution controller was responsible for choosing actions based on this compact representation in a sparse reward environment. The experiments indicate that a deep learning and neuroevolution combination outperforms either alone, NEAT or deep reinforcement learning (SAC) on the same problem. This supports: (1) NE+DL is good at solving high-dimensional sparse reward problems and managed to outperform each of them alone. (2) NEAT through population-based search is able to solve problems with sparse reward, which was supported by various experiments that thoroughly compared RL and NE performance with incremental sparseness. (3) Deep learning is good at extracting features from high-dimensional data, although with lower than expected performance in some experiments.

The work presented in this thesis is both exploratory and supplementary in the relatively new field of NE+DL. Exploratory in that new types of problems have been solved and analyzed, with a new format of the environment, newly created fitness function, and discoveries along the way. More explicitly, contributing to a comparison between three approaches for solving a problem, DRL, NEAT, and NEAT+DL, along with a thorough survey of the field. Supplementary in that it supports the earlier claim in the related field that a NE+DL combination can solve high-dimensional, sparse reward problems.

We demonstrated that deep learning in combination with neuroevolution can be superior in solving high-dimensional, sparse reward problems, suggesting that they can play a vital role in taking us from simplified RL problems to complex real world environments.

Chapter 8

Future Work

In this chapter, possible avenues for future work will be presented, which could help to better understand the implications of the results in this thesis and improve the quality of solutions.

Limitations

Addressing some of the limitations described in the discussion chapter could contribute to achieving higher fitness values for the DL+NE combination primarily and support the results and conclusion in this dissertation. This involves three elements: parameter-search, allowing the algorithms to run for longer to give it more time to optimize, and discovering new ways of applying the DL model.

The DL model had problems with accurately predicting the sensor values, and the reason for this has been described in earlier sections. With a big portion of the sensor values being velocities and speed, which is impossible to predict from still images, additional training and further optimization of the DL model will most likely not significantly improve performance. Consequently, locating new ways of utilizing the DL model would be the logical next step. Employing a VAE, for instance, could be an excellent place to start. The difference between these approaches would be that the input supplied to the NE component will be a compact/encoded representation of the images from the VAE instead of predicted sensor values from the DL model. The training process and usage of the VAE would not involve the sensor values, therefore skipping the problem of the velocity sensor values. The image is used as input and is encoded into a compact representation of the input, which the decoder can decode to recreate the input image. The loss would then be the difference between the recreated output and the input. Hopefully, this can create new representations of the environment that focuses on different aspects than the current DL model, which can hopefully yield a high fitness when used in combination with the NE component. Based on the results, NEAT has performed well with different representations of the environment, from sensor values, inaccurate predictions of the sensor values, and a down-sampled version of the image. This is promising in that NEAT is highly robust to the input it receives, so it should perform well with the new representation given by the VAE, given that this representation is useful.

Algorithms

The focus of future work regarding algorithms is quite broad but can be divided into two sub-focuses: (1) improving performance or (2) comparing or trying new algorithms.

The experiments conducted in this dissertation have been diverse and complex, and choosing an algorithm that performs well in all experiments is not necessarily possible. NEAT struggled with the high-dimensionality and used a down-sample option to help it overcome this challenge. Another approach could be to employ HyperNEAT [52] instead of NEAT to help tackle the high-dimensionality problems. SAC showed to struggle with pixel-to-action mapping, so applying the method described in the background section that focuses on data augmentation could help improve SACs performance on these experiments [28].

The second focus could be to change the algorithms to different related algorithms and examine the difference in performance. For the NE part, CMA-ES and CoSyNe were used in the related papers and could be interesting to investigate this environment further. For the DL model, VAE is particularly interesting to examine in this problem.

Environments

The simulated environments are created as a more simple, controllable version of a real-world environment and are used to test promising algorithms without getting a lot of noise, uncertainty, and other factors, which are ever-present in the real world. The combination with deep learning and neuroevolution showed some promising results in the environment used in this dissertation.

Three distinct directions present themselves when looking at future work regarding environments, (1) environment with the same complexity but with different goals and variations, (2) applying sparse rewards functions to other existing benchmarks, or (3) increasing the complexity, nearing realistic real-world experiments. These directions can help explore the NE+DL combination further to get a better overview of its capabilities. The first direction can help explore the robustness of the approach by changing the environment, and all that comes with this, change in input, output, and goals. This can further relate to the capabilities of the DL model with different types of input and NE capabilities in environments with different goals. The second direction can help explore the strengths, weaknesses of the DL+NE combination with other environments and sparse rewards. The last direction is a step towards the ultimate goal for this field: to perform well in a real-world environment and overcome all the challenges that come with this. Changing the environment by gradually increasing the complexity, adding noise, introduce uncertainty about the perceived data, and the environment in general. Then overcome these challenges by finding algorithms that can thrive in these settings, overcome uncertainty, handle multiple high dimensional data inputs and choosing the correct actions to achieve the goal.

Fitness Function

The sparse fitness function applied in this dissertation was a simplistic, sparse function, which solely focused on the agent's x-position, supplied reward in steps, 0 in-between steps, and -100 if the agent fell. There are, however, countless ways of defining reward functions that can focus on different aspects of the environment, the agents' state, time, and so forth. Evaluating the performance over many different reward functions can give a better understanding of how to design reward functions that have a greater probability of leading to better overall solutions when applying this DL+NE combination to a problem.

Bibliography

[1] S. Alvernaz and J. Togelius. 'Autoencoder-augmented neuroevolution for visual doom playing'. In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. 2017, pp. 1–8. DOI: 10.1109/CIG.2017.8080408.

[2] P. J. Angeline, G. M. Saunders and J. B. Pollack. 'An evolutionary algorithm that constructs recurrent neural networks'. In: *IEEE Transactions on Neural Networks* 5.1 (1994), pp. 54–65. DOI: 10.1109/72.265960.

[3] Greg Brockman et al. *OpenAI Gym*. cite arxiv:1606.01540. 2016. URL: http://arxiv.org/abs/1606.01540.

[4] N. K. Chauhan and K. Singh. 'A Review on Conventional Machine Learning vs Deep Learning'. In: *2018 International Conference on Computing, Power and Communication Technologies (GUCON)*. 2018, pp. 347–352. DOI: 10.1109/GUCON.2018.8675097.

[5] Jie Chen et al. 'Optimal Contraction Theorem for Exploration–Exploitation Tradeoff in Search and Optimization'. In: *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans* 39.3 (2009), pp. 680–691. DOI: 10.1109/TSMCA.2009.2012436.

[6] François Chollet et al. *Keras*. https://keras.io. 2015.

[7] Gabriel V. de la Cruz, Yunshu Du and Matthew E. Taylor. 'Pretraining Neural Networks with Human Demonstrations for Deep Reinforcement Learning'. In: *The Knowledge Engineering Review* 34 (2019), e10. DOI: 10.1017/S0269888919000055.

[8] Alexey Dosovitskiy and Vladlen Koltun. *Learning to Act by Predicting the Future*. 2017. arXiv: 1611.01779 [cs.LG].

[9] Adrien Ecoffet et al. *First return then explore*. 2020. arXiv: 2004.12919 [cs.AI].

[10] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. 2nd. Springer Publishing Company, Incorporated, 2015. ISBN: 3662448734. URL: https://link.springer.com/book/10.1007/978-3-662-44874-8.

[11] Kai Olav Ellefsen and Jim Torresen. 'Self-adapting Goals Allow Transfer of Predictive Models to New Tasks'. In: Nov. 2019, pp. 28–39. ISBN: 978-3-030-35663-7. DOI: 10.1007/978-3-030-35664-4_3.

[12] Clement Farabet et al. 'Learning Hierarchical Features for Scene Labeling'. In: *IEEE Trans. Pattern Anal. Mach. Intell.* 35.8 (Aug. 2013), pp. 1915–1929. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2012.231. URL: https://doi.org/10.1109/TPAMI.2012.231.

[13] Dario Floreano and Claudio Mattiussi. 'Neuroevolution: From architectures to learning'. In: *Evol Intell* 1 (Mar. 2008). DOI: 10.1007/s12065-007-0002-4.

[14] Nicolás García-Pedrajas, Cesar Martínez and Domingo Ortiz-Boyer. 'CIXL2: A Crossover Operator for Evolutionary Algorithms Based on Population Features'. In: *Journal of Global Optimization* 24 (Sept. 2011). DOI: 10.1613/jair.1660.

[15] Faustino J. Gomez and Risto Miikkulainen. 'Solving Non-Markovian Control Tasks with Neuroevolution'. In: *Proceedings of the 16th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'99. Stockholm, Sweden: Morgan Kaufmann Publishers Inc., 1999, pp. 1356–1361. DOI: 10.5555/646307.687435.

[16] Faustino Gomez, Jürgen Schmidhuber and Risto Miikkulainen. 'Accelerated Neural Evolution through Cooperatively Coevolved Synapses'. In: *J. Mach. Learn. Res.* 9 (June 2008), pp. 937–965. ISSN: 1532-4435. DOI: 10.5555/1390681.1390712.

[17] David Ha and Jürgen Schmidhuber. *Recurrent World Models Facilitate Policy Evolution*. 2018. arXiv: 1809.01999 [cs.LG].

[18] Tuomas Haarnoja et al. 'Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor'. In: *CoRR* abs/1801.01290 (2018). arXiv: 1801.01290. URL: http://arxiv.org/abs/1801.01290.

[19] Nikolaus Hansen. 'The CMA Evolution Strategy: A Comparing Review'. In: *Towards a New Evolutionary Computation: Advances in the Estimation of Distribution Algorithms*. Ed. by Jose A. Lozano et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 75–102. ISBN: 978-3-540-32494-2. DOI: 10.1007/3-540-32494-1_4.

[20] Hado V. Hasselt. 'Double Q-learning'. In: *Advances in Neural Information Processing Systems 23*. Ed. by J. D. Lafferty et al. Curran Associates, Inc., 2010, pp. 2613–2621. URL: http://papers.nips.cc/paper/3964-double-q-learning.pdf.

[21] Hado van Hasselt, Arthur Guez and David Silver. *Deep Reinforcement Learning with Double Q-learning*. 2015. arXiv: 1509.06461 [cs.LG].

[22] Matteo Hessel et al. 'Rainbow: Combining Improvements in Deep Reinforcement Learning'. In: *CoRR* abs/1710.02298 (2017). arXiv: 1710.02298. URL: http://arxiv.org/abs/1710.02298.

[23] Todd Hester et al. 'Learning from Demonstrations for Real World Reinforcement Learning'. In: *CoRR* abs/1704.03732 (2017). arXiv: 1704.03732. URL: http://arxiv.org/abs/1704.03732.

[24] Ashley Hill et al. *Stable Baselines*. https://github.com/hill-a/stable-baselines. 2018.

[25] G. Hinton et al. 'Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups'. In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 82–97. DOI: 10.1109/MSP.2012.2205597.

[26] *IN5400 - Machine Learning for Image Processing, Week 5*. https://www.uio.no/studier/emner/matnat/ifi/IN5400/v20/material/week5/in5400_2020_week5_convolutional_nerual_networks.pdf. Accessed: 2021-04-26. 2020.

[27] Shauharda Khadka and Kagan Tumer. *Evolution-Guided Policy Gradient in Reinforcement Learning*. 2018. arXiv: 1805.07917 [cs.LG].

[28] Ilya Kostrikov, Denis Yarats and Rob Fergus. 'Image Augmentation Is All You Need: Regularizing Deep Reinforcement Learning from Pixels'. In: *CoRR* abs/2004.13649 (2020). arXiv: 2004.13649. URL: https://arxiv.org/abs/2004.13649.

[29] Jan Koutnik, Juergen Schmidhuber and Faustino Gomez. 'Evolving Deep Unsupervised Convolutional Networks for Vision-Based Reinforcement Learning'. In: *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. GECCO '14. Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 541–548. ISBN: 9781450326629. DOI: 10.1145/2576768.2598358. URL: https://doi.org/10.1145/2576768.2598358.

[30] John R. Koza. 'Human-Competitive Results Produced by Genetic Programming'. In: *Genetic Programming and Evolvable Machines* 11.3–4 (Sept. 2010), pp. 251–284. ISSN: 1389-2576. DOI: 10.1007/s10710-010-9112-3. URL: https://doi.org/10.1007/s10710-010-9112-3.

[31] Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton. 'ImageNet Classification with Deep Convolutional Neural Networks'. In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386.

[32] Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton. 'ImageNet Classification with Deep Convolutional Neural Networks'. In: *Communications of the ACM* 60.6 (June 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386.

[33] Yann Lecun, Yoshua Bengio and Geoffrey Hinton. 'Deep learning'. English (US). In: *Nature Cell Biology* 521.7553 (May 2015), pp. 436–444. ISSN: 1465-7392. DOI: 10.1038/nature14539.

[34] Sambit Mahapatra. *Why Deep Learning over Traditional Machine Learning?* Jan. 2019. URL: https://towardsdatascience.com/why-deep-learning-is-needed-over-traditional-machine-learning-1b6a99177063.

[35] Gary Marcus. *Deep Learning: A Critical Appraisal*. 2018. arXiv: 1801.00631 [cs.AI].

[36] Alan McIntyre et al. *neat-python*. https://github.com/CodeReclaimers/neat-python.

[37] T. Mikolov et al. 'Strategies for training large scale neural network language models'. In: *2011 IEEE Workshop on Automatic Speech Recognition Understanding*. 2011, pp. 196–201. DOI: 10.1109/ASRU.2011.6163930.

[38] Volodymyr Mnih et al. 'Human-level control through deep reinforcement learning'. In: *Nature* 518.7540 (Feb. 2015), pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

[39] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[40] David Moriarty et al. 'Symbiotic Evolution of Neural Networks in Sequential Decision Tasks'. In: (May 1997). URL: https://www.researchgate.net/profile/David_Moriarty/publication/2672614_Symbiotic_Evolution_of_Neural_Networks_in_Sequential_Decision_Tasks/links/0a85e530df1c0ca19a000000/Symbiotic-Evolution-of-Neural-Networks-in-Sequential-Decision-Tasks.pdf.

[41] Anh Nguyen et al. *Synthesizing the preferred inputs for neurons in neural networks via deep generator networks*. 2016. arXiv: 1605.09304 [cs.NE].

[42] Matthias Plappert et al. *Parameter Space Noise for Exploration*. 2018. arXiv: 1706.01905 [cs.LG].

[43] Andreas Precht Poulsen et al. 'DLNE: A hybridization of deep learning and neuroevolution for visual control'. In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. 2017, pp. 256–263. DOI: 10.1109/CIG.2017.8080444.

[44] Edmund Ronald and Marc Schoenauer. 'Genetic lander: An experiment in accurate neuro-genetic control'. In: *Parallel Problem Solving from Nature — PPSN III*. Ed. by Yuval Davidor, Hans-Paul Schwefel and Reinhard Männer. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 452–461. ISBN: 978-3-540-49001-2. DOI: 10.1007/3-540-58484-6_288.

[45] Tara N. Sainath et al. 'Deep Convolutional Neural Networks for Large-scale Speech Tasks'. In: *Neural Networks* 64 (2015). Special Issue on "Deep Learning of Representations", pp. 39–48. ISSN: 0893-6080. DOI: https://doi.org/10.1016/j.neunet.2014.08.005. URL: http://www.sciencedirect.com/science/article/pii/S0893608014002007.

[46] Tim Salimans et al. *Evolution Strategies as a Scalable Alternative to Reinforcement Learning*. 2017. arXiv: 1703.03864 [stat.ML].

[47] David Silver. *Interested in learning more about reinforcement learning? Follow along in this video series as DeepMind Principal Scientist, creator of AlphaZero and 2019 ACM Computing Prize Winner David Silver, gives a comprehensive explanation of everything RL.* URL: https://deepmind.com/learning-resources/-introduction-reinforcement-learning-david-silver.

[48] David Silver et al. 'Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm'. In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815.

[49] David Silver et al. 'Mastering the Game of Go with Deep Neural Networks and Tree Search'. In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. DOI: 10.1038/nature16961.

[50] Karen Simonyan and Andrew Zisserman. 'Very Deep Convolutional Networks for Large-Scale Image Recognition'. In: *arXiv 1409.1556* (Sept. 2014).

[51] Jost Tobias Springenberg et al. *Striving for Simplicity: The All Convolutional Net*. 2014. arXiv: 1412.6806 [cs.LG].

[52] Kenneth O. Stanley, David B. D'Ambrosio and Jason Gauci. 'A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks'. In: *Artificial Life* 15.2 (2009). PMID: 19199382, pp. 185–212. DOI: 10.1162/artl.2009.15.2.15202. eprint: https://doi.org/10.1162/artl.2009.15.2.15202. URL: https://doi.org/10.1162/artl.2009.15.2.15202.

[53] Kenneth O. Stanley and Risto Miikkulainen. 'Evolving Neural Networks through Augmenting Topologies'. In: *Evolutionary Computation* 10.2 (2002), pp. 99–127. DOI: 10.1162/106365602320169811. eprint: https://doi.org/10.1162/106365602320169811. URL: https://doi.org/10.1162/106365602320169811.

[54] Kenneth Stanley et al. 'Designing neural networks through neuroevolution'. In: *Nature Machine Intelligence* 1 (Jan. 2019). DOI: 10.1038/s42256-018-0006-z.

[55] Felipe Petroski Such et al. 'Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning'. In: *CoRR* abs/1712.06567 (2017). arXiv: 1712.06567. URL: http://arxiv.org/abs/1712.06567.

[56] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[57] C. Szegedy et al. 'Going deeper with convolutions'. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 1–9. DOI: doi:10.1109/CVPR.2015.7298594.

[58] Jenine Turner and Eugene Charniak. 'Supervised and unsupervised learning for sentence compression'. In: *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*. 2005, pp. 290–297.

[59] Pauli Virtanen et al. 'SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python'. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[60] Haoran Wang, Thaleia Zariphopoulou and Xunyu Zhou. *Exploration versus exploitation in reinforcement learning: a stochastic control approach*. 2018. arXiv: 1812.01552 [math.OC].

[61] Wikipedia contributors. *Biological neuron model — Wikipedia, The Free Encyclopedia*. [Online; accessed 28-April-2021]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Biological_neuron_model&oldid=1020222941.

[62] YY Wong et al. 'A novel approach in parameter adaptation and diversity maintenance for genetic algorithms'. In: *Soft Computing - A Fusion of Foundations, Methodologies and Applications* 7 (Jan. 2003), pp. 506–515. DOI: 10.1007/s00500-002-0235-1.

[63] Xin Yao. 'Evolving artificial neural networks'. In: *Proceedings of the IEEE* 87.9 (1999), pp. 1423–1447. DOI: 10.1109/5.784219.

[64] Matthew D Zeiler and Rob Fergus. *Visualizing and Understanding Convolutional Networks*. 2013. arXiv: 1311.2901 [cs.CV].

[65] Xiaojin Zhu and Andrew B Goldberg. 'Introduction to semi-supervised learning'. In: *Synthesis lectures on artificial intelligence and machine learning* 3.1 (2009), pp. 1–130.

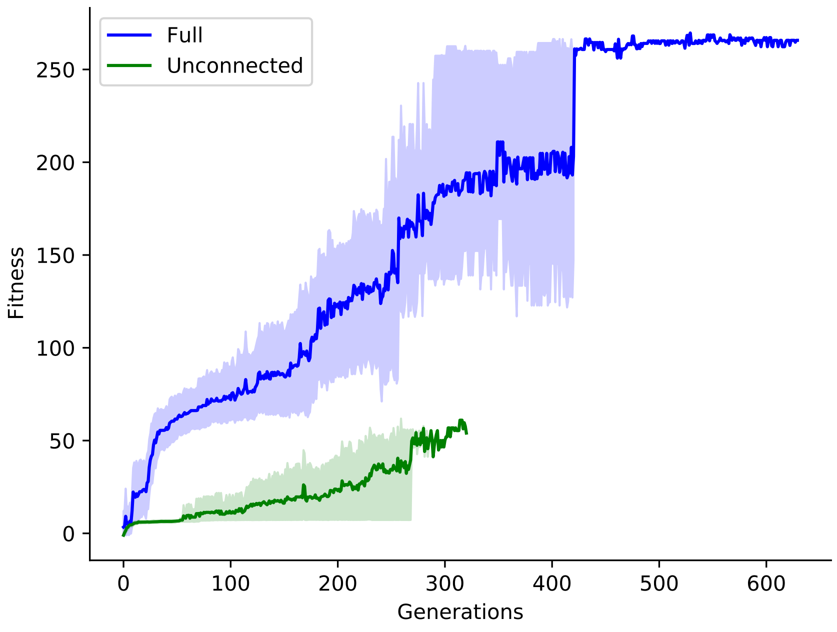
Appendix

A Additional Experiments

In this section, a few extra experiments are presented that was not included in the experiment and result section.

A.1 NEAT - Fully Connected vs. Unconnected

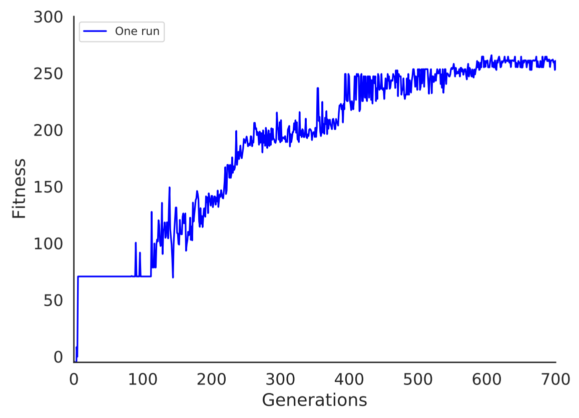
An experiment was done with altering the starting architecture, full connect, meaning that all input nodes are connected to all output nodes, and unconnected meaning no connections.



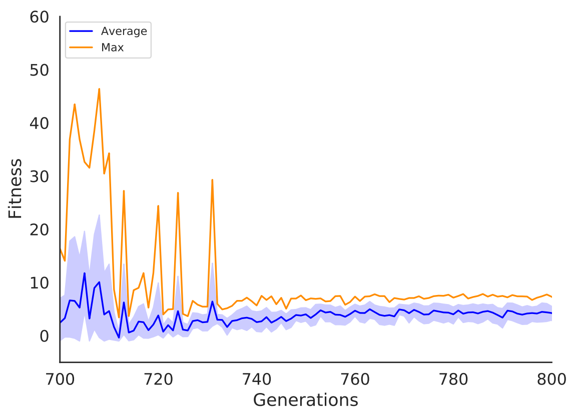
Starting architecture full connect vs. unconnected

A.2 NEAT - Pretrained NEAT

In this experiment, NEAT was first trained using the sensor values from the environment, then when stopped by the non-improvement threshold, switch to sensor values from the DL model.



(a) Environment Values



(b) DL values

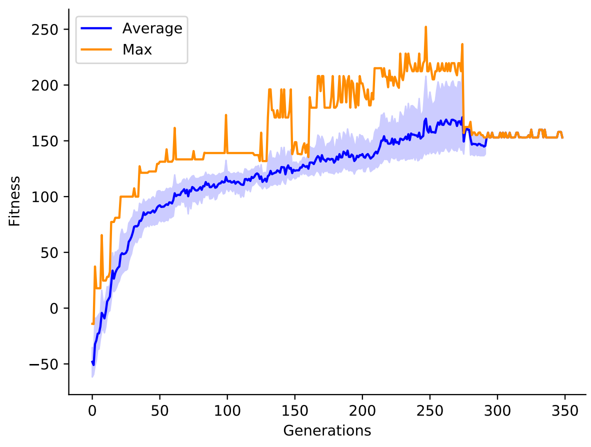
NEAT was first trained, using the original state sensor values provided by the environment, then after 700 generations, the sensor values were switched to using the predicted state sensor values provided by the DL model. From these plots, it is apparent that the DL values are quite inaccurate as the fitness drops significantly and remains quite low until the non-improvement threshold stops it. Plot (a) displays one run of neat, plot (b) displays confidence interval (shaded area) over 8 runs, average (blue line) and maximum (dark orange line)

B Supplementary Plots

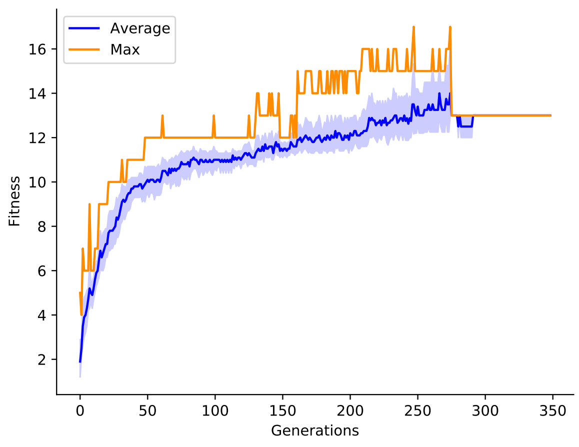
This section displays supplementary plots to some experiments in the Experiments and results chapter.

B.1 Sparse Fitness Sensor Levels

This section extends from the section with the 24-sensor sparse fitness function in chapter 5.Experiments and Results. These plots are with NEAT only because RL did not perform well with this fitness function. In these plots, both the max accumulated sparse reward and max accumulated normal reward show how the agent performed with the sparse fitness function with the normal fitness function.

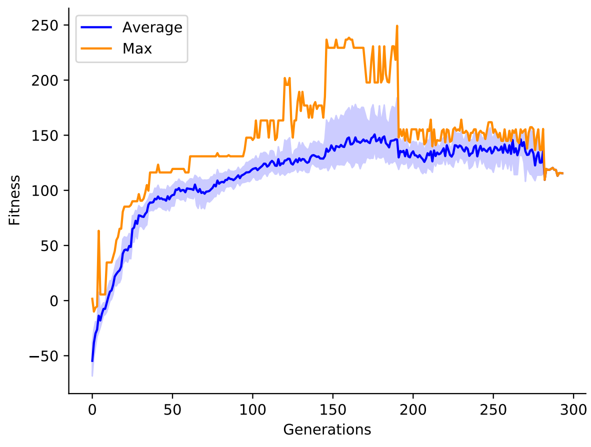


(a) Normal Reward

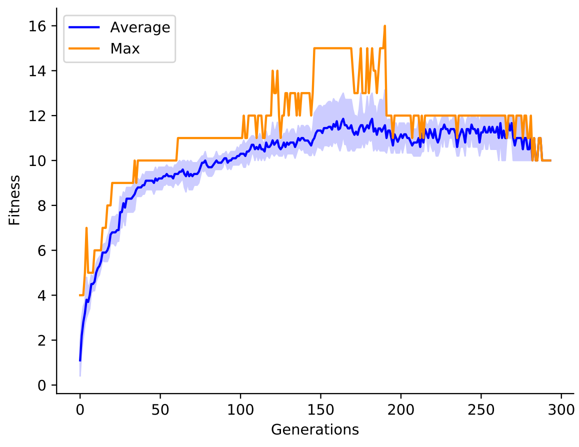


(b) Sparse Reward

These plots displays the confidence interval over ten runs, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. Neat with sparse level 5, using 24 state and terrain sensor values - plot (a) and (b) displays the performance by looking at the different accumulative fitness, with either the normal fitness function or sparse fitness function

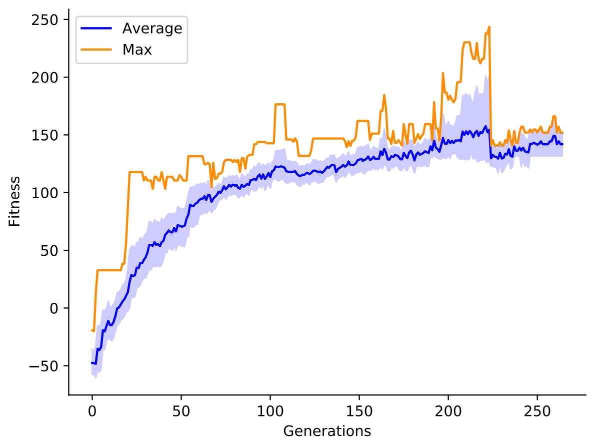


(a) Normal Reward

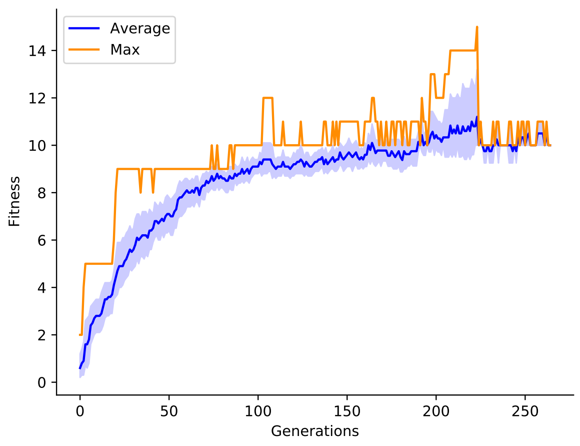


(b) Sparse Reward

These plots displays the confidence interval over ten runs, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. Neat with sparse level 10, using 24 state and terrain sensor values - plot (a) and (b) displays the performance by looking at the different accumulative fitness, with either the normal fitness function or sparse fitness function

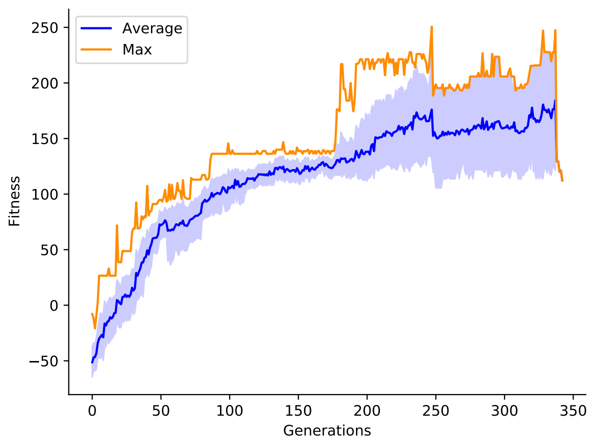


(a) Normal Reward

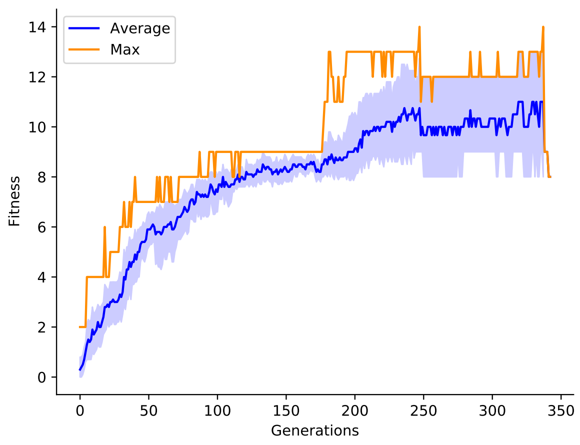


(b) Sparse Reward

These plots displays the confidence interval over ten runs, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. Neat with sparse level 15, using 24 state and terrain sensor values - plot (a) and (b) displays the performance by looking at the different accumulative fitness, with either the normal fitness function or sparse fitness function

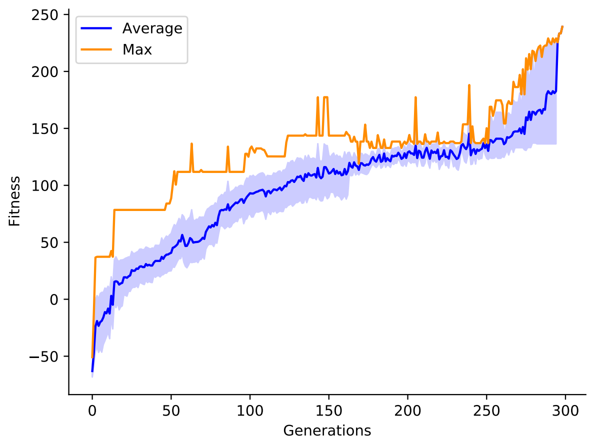


(a) Normal Reward

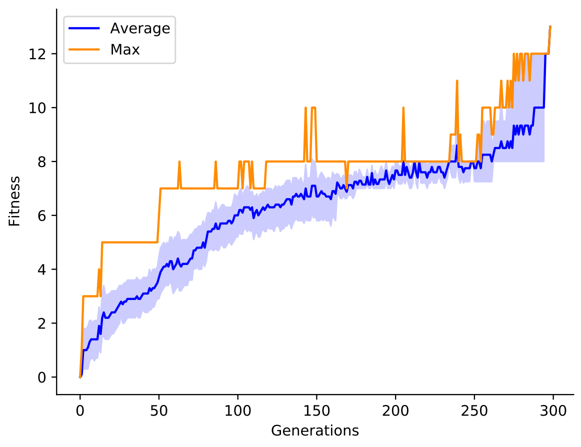


(b) Sparse Reward

These plots displays the confidence interval over ten runs, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. Neat with sparse level 20, using 24 state and terrain sensor values - plot (a) and (b) displays the performance by looking at the different accumulative fitness, with either the normal fitness function or sparse fitness function

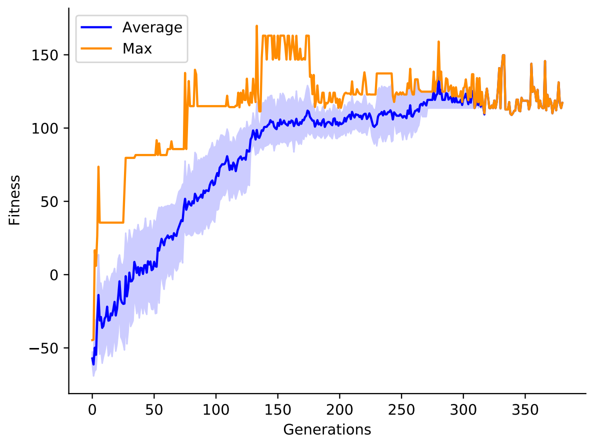


(a) Normal Reward

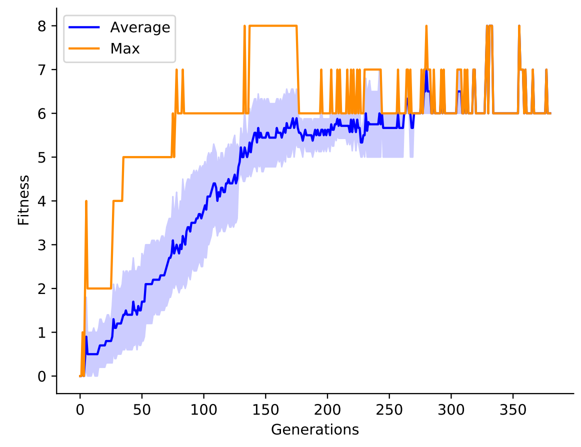


(b) Sparse Reward

These plots displays the confidence interval over ten runs, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. Neat with sparse level 25, using 24 state and terrain sensor values - plot (a) and (b) displays the performance by looking at the different accumulative fitness, with either the normal fitness function or sparse fitness function



(a) Normal Reward

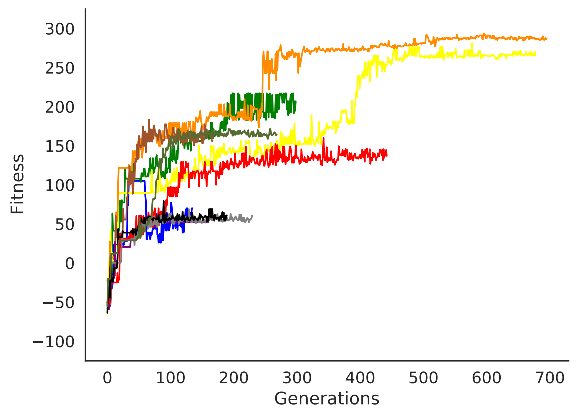


(b) Sparse Reward

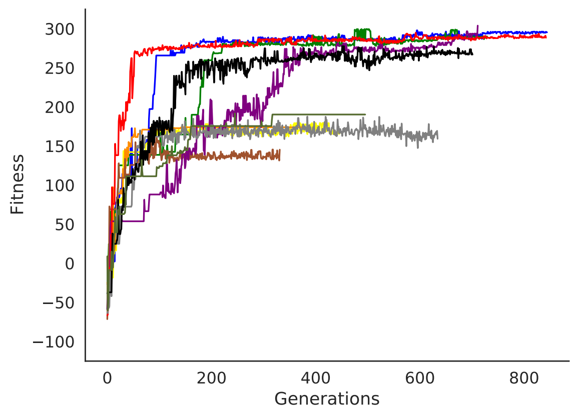
These plots displays the confidence interval over ten runs, which is the shaded area, the mean is the blue line, and the maximum is the dark orange line. Neat with sparse level 30, using 24 state and terrain sensor values - plot (a) and (b) displays the performance by looking at the different accumulative fitness, with either the normal fitness function or sparse fitness function

C Raw plots

This section displays the raw versions of the plots from experiments and results, without the Gaussian filters applied.

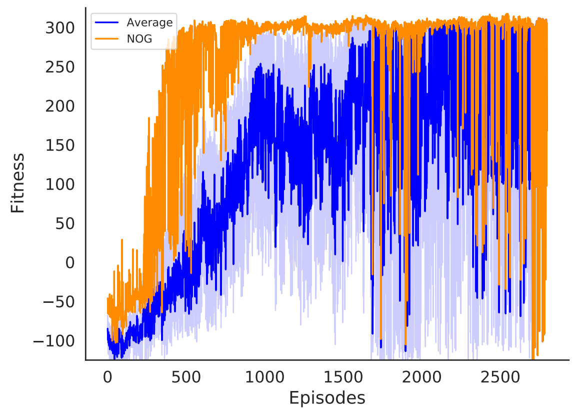


(a) State and Terrain

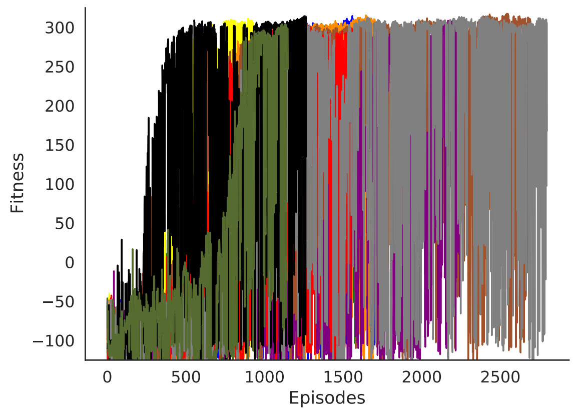


(b) State

NEAT - Normal Rewards - Single run - Raw Plots

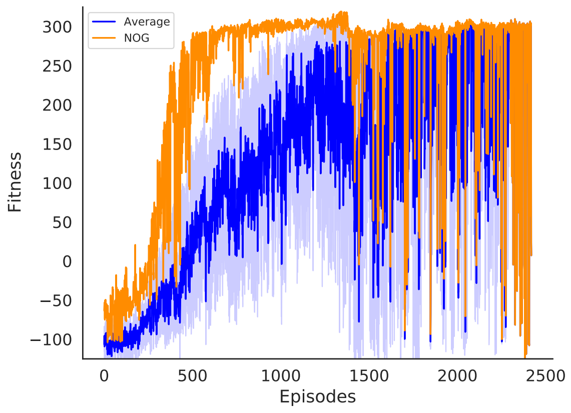


(a) Confidence interval

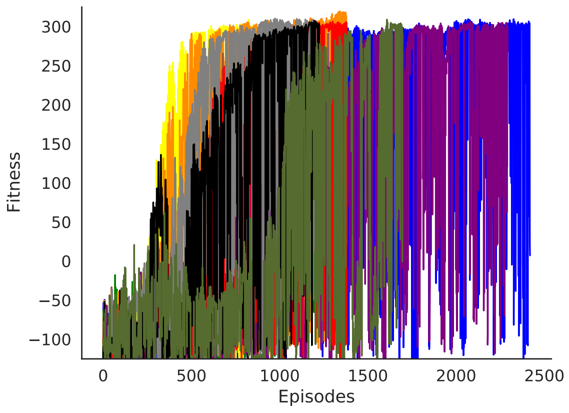


(b) Single Runs

SAC - Normal Rewards - State and Terrain sensor - Raw Plots

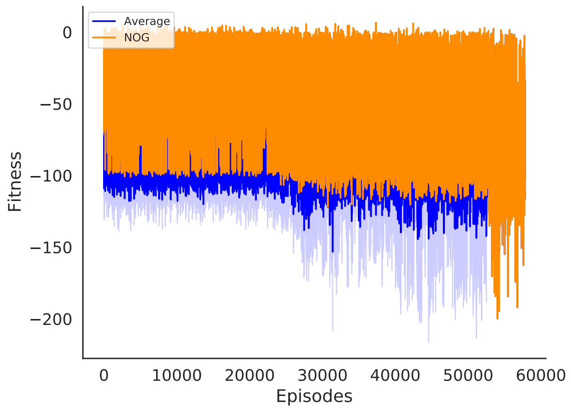


(a) Confidence interval

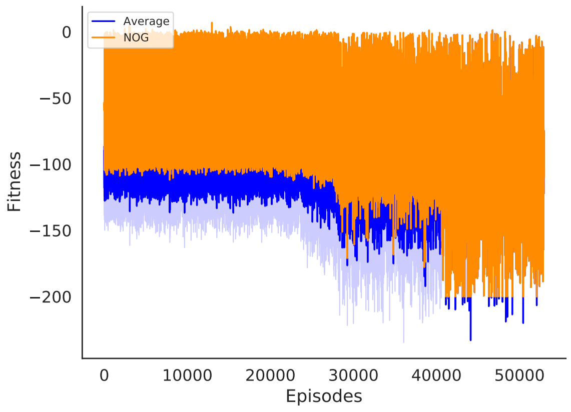


(b) Single Runs

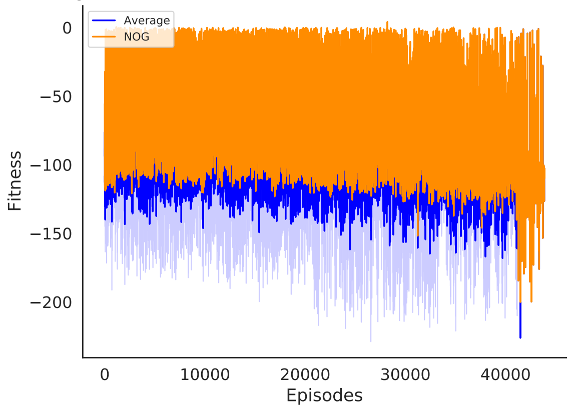
SAC - Normal Rewards - State sensor - Raw Plots



(a) Full size image

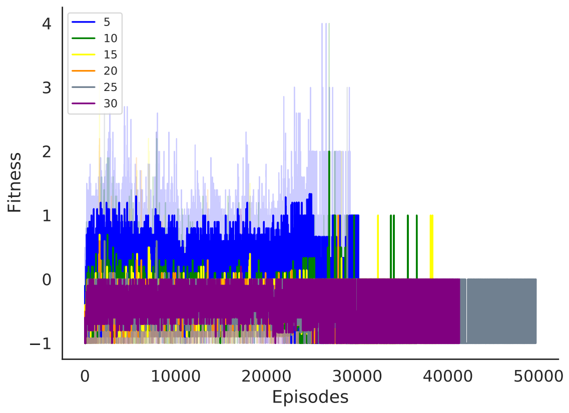


(b) Down-sampled image

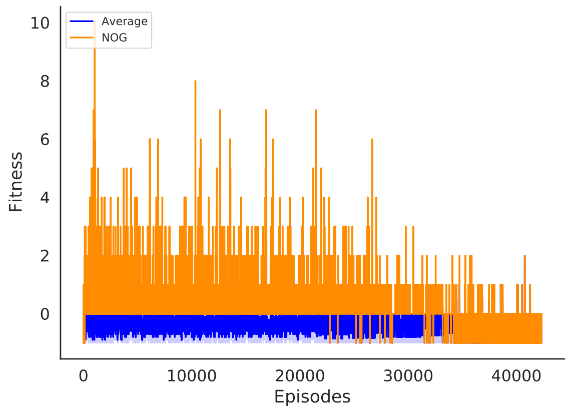


(c) DL + SAC

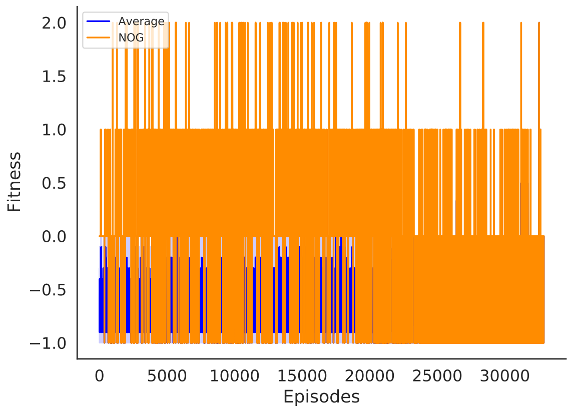
SAC - Normal Rewards - Image values - Raw Plots



(a) State and Terrain



(b) State



(c) image

SAC - Sparse Rewards - State and Terrain/State/Image sensor - Raw Plots