Neuroevolution

Emil Lundt Larsen (s153214)

December 6, 2020

Emil Lundt Larsen (s153214)
6/12-20

Abstract

The aim of this study is to explore problems and state-of-the-art solutions in the field of neuroevolution and apply relevant techniques for simple reinforcement learning problems in practice. An overview of the main ideas from the literature is given and two influential algorithms for training neural networks using evolutionary algorithms are described as well as a short account on some of the classical reinforcement learning techniques including the the Deep Q-learning algorithm in order to draw a comparison. The algorithms are benchmarked on a cartpole balancing problem and pacman environment, based on which the algorithms are compared and the pros and cons of the different approaches discussed. An attempt is also made to analyse the runtime of the (1+1) ES evolutionary algorithm on a simple neuroevolution optimisation problem, which is not something that has been extensively studied in the literature before. The study concludes that NEAT and GA achieve good results on the reinforcement learning problems considered and are competitive alternatives to traditional reinforcement learning techniques such as Q-learning for training neural networks. GA is remarkably simple and scalable, but is still able to find good solutions very fast, while NEAT by virtue of evolving an initially minimal structure shows that small simple networks with the right topology can be very efficient at reinforcement learning tasks.

Contents

Emil Lundt Larsen (s153214)
6/12-20

1 Introduction

With the massive increase in computing power and amount of data available in recent decades, machine learning and deep learning, in particular, has gained a very high amount of popularity among researchers, scientists and engineers. Much of the research in deep learning has focused on training neural networks using stochastic gradient descent and backpropagation, which has lead to many great results. However, there is also an alternative approach for training neural networks, which is to use evolutionary algorithms instead of backpropagation. This idea gives rise to the term neuroevolution that offers many advantages such as greater potential for exploration, parallelisation and diversity maintaining techniques also relevant in other machine learning areas. Recent research show impressive results in difficult domains in reinforcement learning motivates further study and application of these techniques in future research. [22, 18, 24]

The aim of this project is to explore problems and state-of-the-art solutions in the field of neuroevolution and apply relevant techniques for a simple reinforcement learning problem in practice. A characterisation of neuroevolution will be given as well as an account on the main ideas and how they relate to classical reinforcement learning, which is notoriously difficult paradigm. The NEAT algorithm [21] and a simple genetic algorithm (GA) [24] for reinforcement learning will be implemented and benchmarked on the classic reinforcement learning problem cartpole as well as the harder domain of pacman. A comparison will be made with the classic reinforcement learning approach Deep Q-learning (DQN) [4] and the advantages and disadvantages of the approaches be discussed. Furthermore, the runtime analysis of a simple theoretical benchmark problem for neuroevoluition will be presented and analysed and later benchmarked by running an implementation of the (1+1) ES algorithm on instances of the problem.

The report is structured as follows: Initially in section 2 the background and main ideas behind neuroevolution are described as well as an account on the classical terms of the reinforcement learning paradigm. Section 4 analyses the cartpole and pacman problems and what will become relevant in the design of solutions on those problems. We also analyse a runtime analysis problem and make propositions on the runtime of some evolutionary algorithms on the problem. Section 5 presents the design and structure of the implementation of NEAT, the GA and DQN as well as the runtime analysis problem and the algorithms used in that context. Followed by this 6 touches upon the details of the implementation of the various algorithms and problems. The results and benchmarks are carried out and presented in 7 followed by a discussion of the results. Further discussion on a more general level of the project is given in section 8. The report is concluded with ideas for future work and a conclusion in section 8 and 9, respectively.

References are listed at the end of the report. The source code can be found as a .zip folder alongside this report.

2 Background

2.1 Neuroevolution

The main idea behind neuroevolution is to evolve and train artificial neural networks using evolutionary algorithms. Inspired by nature, neuroevolution encompasses methods and techniques, where typically a population of neural networks is maintained and over generations adapted and improved with respect to some fitness measure using mutation, crossover and selection operations on the individuals of the population. The main difference between neuroevolution and classical reinforcement learning techniques therefore lies in neuroevolution algorithms using multiple neural networks and training them using evolutionary principles instead of using changing the weights of the networks by backpropagation computed with respect to an error function.

There are different ways this evolving process can occur. Both in terms of the evolutionary hyper-parameters and workings of the mutation, crossover and selection operators, but also in terms of what aspects of the neural networks are evolved. There are three main ways to evolve the weights of the network: Either the network topology i.e. the number of hidden neurons and connections are fixed and only the weights of the connections are changed during training as in [24] described in 2.4. Alternatively the weights as well as the topology of the network are evolved like in the Neuroevolution for Augmented Topologies (NEAT) approach [21] described in section 2.3. Lastly, an algorithm can keep the weights of the network fixed and evolve only the topology of the network as in the weight agnostic network paper by Gaier et al. [8]. Other ways exist as well such as changing activations and evolutionary hyper-parameters during training, but do not seem as common. Neural architecture search is reminiscent of the idea of searching for an optimal network topology, but it differs from neuroevolution in that it typically optimises the network weights themselves as a part of this search process through backpropagation.

Another aspect of nature that neuroevolution draws inspiration from is that of novelty and diversity. An important aspect of natural evolution is that it not just attempts to find one optimal species; nature features a wide range of different species and niches that do not directly compete against one another and are able to co-exist. The idea of creating diversity can be greatly beneficial in solving computational problems as well, as purely gearing search towards the optimal solution based on how good the solution is in terms of fitness with respect to some fitness function can lead to the problem of deception. Lehman and Stanley [14] argued based a maze problem that searching for the solution can be worse that not looking for the solution at all and reward diverse solutions instead of solutions that come closer to the goal in the maze. Diversity can help create stepping stones towards an optimal solution and by serendipity some valuable individuals may be found this way, which can lead to a breakthrough in the search process. Hence the name novelty search as the authors call their search technique. Based on this a number of derivative search methods have come to light including novelty search with local competition, curiosity search and quality diversity like NSLC and MAP-Elites. [7, 23, 17] Sometimes goal switching may also be beneficial if the search stagnates or if other types of solutions may be interesting by changing the measure of fitness.

There are many applications of neuroevolution as neural networks are a very flexible model for a wide range of optimisation problems. A very broad class of problems where neuroevolution in particular in recent years have shown great potential is that of problems in the reinforcement learning paradigm. The reason for this lies in the typically very complex and non-convex nature of the problems and the computational demands for finding a good approximation for a solution. Moreover the integral and important aspect of exploration in order to find good solutions. Recent literature has shown that genetic algorithms are a competitive alternative for training deep neural networks for difficult reinforcement learning problems [24] and how evolutionary strategies can be used to make training highly scalable across many CPU cores and enable achieving state-of-the-art results very fast compared to previous research [18].

2.2 Reinforcement Learning

Reinforcement learning is about how to map situations to actions in order to maximise a reward signal. An agent has to learn by trial-and-error what actions in a given state in the environment leads to the greatest reward. In this way reinforcement learning is reminiscent of unsupervised learning, but learning a model and structural representation of the environment is only a means for maximising the reward. The true goal is learn an optimal policy, which maps states to actions (or probabilities of choosing an action if the policy is stochastic). The following subsections briefly describe the fundamental terms of reinforcement learning based on the book by Sutton and Barto and course notes of David Silver. [4, 19]

2.2.1 Markov Decision Process

The environment the agents are set in is usually formalised as a Markov Decision Process (MDP), which characterises the sequential decision making process of the agent and transition from state to state. Formally an MDP is a 4-tuple (S, A, P_a, R_a) where:

- S is the state space and is a set of states
- A is the action space and is a set of actions
- $P_a(s, s')$ is the probability that action a in state s at timestep t will lead to state s' in timestep t+1
- $R_a(s, s')$ is the reward received after transitioning from state s to state s' by taking action a

P_a and R_a form the dynamics of the MDP and may be known to the agent a priori or not. For dynamic problems these may also change with time. Note that we only consider MDPs over finite state-action pairs here. In an MDP the states should satisfy the Markov property, which is that the future is independent of the past given the present state, that is, $p(s_{t+1}|s_t, a_t) = p(s_{t+1}|h_t, a_t)$, where s_t is the state at timestep t and h_t is the history of states at timestep t; a_t is the action chosen by the agent at timestep t. Theoretically this can always be satisfied by setting the current state equal to the history of states, but in practice it is often assumed that the most recent observation is a sufficient statistic of the history and that the relationship holds.

2.2.2 Fundamental Terms

Some fundamental terms in reinforcement learning that will become relevant when discussing the algorithms of this project is that of episodes, return and horizon. An episode denotes the sequence of states and actions that an agent has been in and taken until the terminal state is reached. The term trajectory is sometimes used interchangeably . The return is defined as the sum of rewards until the end of the episode. The time horizon is how many future steps are considered in the calculation of the return. The return is mathematically defined as:

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{1}$$

where $0 \leq \gamma \leq 1$ and γ denotes a discount factor or discount rate. It is the return that is central in how well an agent is performing at any state while interacting with the environment, since it talks about the total reward and not just the immediate reward from reaching one state. The total reward at the end of an episode is what the result ultimately is.

Next we define the so-called value function as the expected return starting in state s and following policy π:

$$\begin{aligned} v_\pi(s) &= E[G_t|S_t = s] && (2) \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] && (3) \end{aligned}$$

Equation (3) is called the Bellman equation for v_π.

Similarly we define the action-value function for policy pi:

$$q_\pi(s, a) = E_\pi[G_t|S_t = s, A_t = a] \tag{4}$$

which includes the state and an action. The action-value function is essential in reinforcement learning control problems as it directly tells which action in the current state would yield the greatest expected return under the current policy and hence the presumably best action.

Solving a reinforcement learning task amounts to finding an optimal policy, that is, a policy that maximises the expected return for any given state. It can formally be defined wrt. to the value function by the policy for which $v_\pi(s)$ is the greatest for all states and policies. An optimal policy π^* is thereby given as:

$$\pi^* = E[\sum \gamma^t R_a(s, s_{t+1})] \tag{5}$$

Equivalently one can define the optimal policy to be the greatest element in the set of policies partially ordered by a value function v such that two policies are ordered $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_\pi(s)$ for all states $s \in S$. We define the optimal state-value function as:

$$v_*(s) = \max_\pi v_\pi(s) \tag{6}$$

and similarly the optimal action-value function as:

$$q_*(s,a) = \max_\pi q_\pi(s,a) \tag{7}$$

Note that assuming the agent has full knowledge of the dynamics of the MDP, it is straight forward to derive the action-value function based on the value function, as the agent can evaluate the value-function for the possible actions in the current state, so there is a close link between the two functions.

Lastly we mention in this subsection the two types of reinforcement learning problems, namely the prediction and control problems, which can either be model-based or model-free. Prediction problems are about predicting which states are better than other states. Control problems are concerned with choosing which action is the best in a given state. Model-based means the dynamics of the MDP problem are given to the agent so it knows the probabilities of the transition from a state by taking an action leading to other states. In model-free problems the dynamics are not all given; the environment may be partially observable but not fully observable.

2.2.3 Classical Reinforcement Learning Techniques

Before describing the DQN algorithm we briefly describe some of the classical reinforcement learning techniques that make up the basis for more advanced reinforcement learning algorithms such as DQN, AC3 and similar. This foundation includes the methods dynamic programming, monte carlo prediction/control and temporal difference learning.

To understand dynamic program we must first state the Bellman optimality equations:

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{8}$$

$$q_*(s,a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{9}$$

The main idea of DP is to use the Bellman optimality equations for updating our knowledge about the value function. Once we have the optimal value function it is straightforward to find an optimal policy. There are a number of DP algorithms, one of which is value iteration, which sweeps through the state space and updates the value function for each state by equation (8). Finally after this loop converges then the policy is set to $\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')]$. Thus, we iteratively improve the value function by taking one step, looking at the reward and then backing up the previous state with the reward plus our estimate of the value from the successor state. Hence, the dynamic programming aspect as we utilise the optimal substructure property of the problem, that is, we utilise that the value of a state is equal to the immediate reward of one step plus the expected return from the successor state. This can visually be represented in the form of backup diagrams (see figure 1)

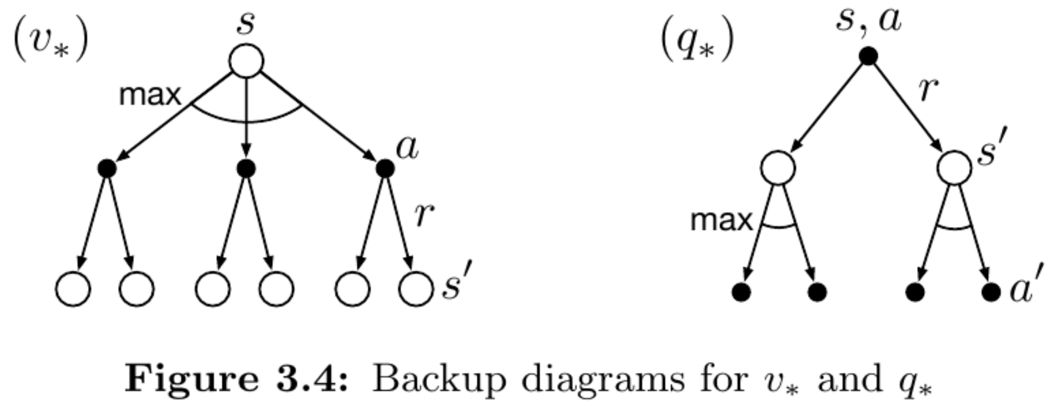


Figure 1

DP is a model-based technique and requires full knowledge of the problem formulated as an MDP, which is an assumption that often is not satisfied in practice. Two major methods that do not need full knowledge of the dynamics for prediction (model-free prediction) are Monte-Carlo (MC) and Temporal-Difference-learning (TD). The difference between them lies in MC learning from complete episodes without bootstraping and TD learning from incomplete episodes by bootstrapping. Bootstrap in this context refers to the act of substituting the remainder of the trajectory with the estimate of what will happen from that point onward. This is expressed in the following update rules for the value of a state. Equation (10) is for monte carlo and (11) is for TD-learning:

$$v(s_t) = v(s_t) + \alpha(G_t - v(s_t)) \tag{10}$$

$$v(s_t) = v(S_t) + \alpha(r_{t+1} + \gamma v(s_{t+1}) - v(s_t)) \tag{11}$$

where α is step size parameter and t the timestep in the trajectory. Observe that TD utilises the Markov property of the MDP and updates for each step whereas MC ignores the Markov property and just updates based on the entire trajectory. In a non-Markov environment updating based on the entire trajectory can be an advantageous, however.

A number of derivatives and in-between methods exists based on these methods. For instance TD(λ), which can allow more than one step before updating the value for the state S. For control problems the so-called SARSA technique is the method that is based on TD-learning and the Monte Carlo approach is dubbed Monte-Carlo control.

2.2.4 Double Q-Learning

A major problem with DP is that it requires full knowledge of the dynamics of the environment which is often not what we want from a method for solving problems. TD-learning and Monte-Carlo do not have this assumption and learn just by experience sampled from interaction with the environments. However, both DP, TD and MC are tabular methods and a direct implementation scale very poorly with the number of states. Implementing them would amount to storing and updating a look up table for the value function V and/or action-value function Q for each state in the state space, which would simply require too much memory in practice. The training time would also be intractable for all but small problems.

This motivates the use of neural networks as we can use these for value function approximation. The input will be the state and the output the value of the state. One can then use Mone-Carlo or TD/SARSA updates as before. A drawback is that some of the nice theoretical properties and guarantees of convergence are given up at expense of this, but in practice this is necessary.

Deep Q-Networks or Double Q-Learning (DQN) improves on the Q-learning method by two tricks: It uses an experience replay buffer to store past experience and samples from this instead of always choosing the most recent episode for making updates. This decorrelates trajectories and makes better use of the data. DQN also uses two networks by boostrapping towards the Q values of a fixed target network, but updating another network. Every n number of iterations the target network is set equal to the current network and the training continues. This also helps deal with correlation between the target and Q values in the update moment. Technically the DQN algorithm when using these two tricks originally used for non-tabular settings by authors at Deep Mind is called Double Q-Learning [15] as Deep Q-Learning refers to the original application of Q-learning using neural networks on the Atari domain in reinforcement learning [16], but we take the liberty of using the Deep Q-Networks and Double Q-Learning interchangably in this project as Double Q-learning since the original non double Q-learning variant is much less used.

2.3 NEAT

The idea for Neuroevolution for augmenting topologies dates back to 2002 by Stanley and Miikkulainen, [21] but has been used and cited extensively since then in further research and is still considered a good algorithm for some problems.

The main idea is to maintain a population feed-forward neural networks that are evolved over a number of generations by selection, mutation and crossover. Each individual neural network is represented by a genotype consisting of a number of node genes and connection genes. Each node gene has a number and is either of type input, output or hidden. A connection gene has two numbers for in and out node genes, a weight and is either enabled or not and also has an innovation number which is used in the historical marking process (see below). The genotype uniquely encodes a neural network, which is called the phenotype, a term from genetic programming. The encoding scheme is direct encoding since the genotype directly translates to the phenotype neural network contrary to indirect encoding schemes where a genotype encodes the means to grow the corresponding phenotype. Indirect encoding is more advanced, but also has the capability to encode more advanced structures in less space. In a paper by the same authors later released, they adapt NEAT to use indirect encoding, where they call the algorithm Hyper-NEAT [20].

Pseudo-code for the NEAT algorithm is given below, where some details have been omitted.

Algorithm 1 NEAT

```
1: Initialise population $P_0$ of size $\mu$
2: $t \leftarrow 0$
3: **while** stopping criterion not fulfilled **do**
4:     $P_{t+1} := \emptyset$
5:     Purge stagnant species
6:     Adjust fitness
7:     **for** species in Species **do**
8:         $P_{t+1} \leftarrow P_{t+1} \cup$ {best genome in species}
9:         **for** $i$ in range(size) **do**
10:            Select $p_1, p_2$ randomly from members of species
11:            Cross-over p1 and p2 to get c
12:            Mutate c to get $c'$ $P_{t+1} \leftarrow P_{t+1} \cup \{c'\}$
13:        **end for**
14:    **end for**
15:    Speciate
16:    $t \leftarrow t + 1$
17: **end while**
```

The mutation operator is either add node mutation or add connection mutation. The add node mutation splits a connection up into two with an intermediate node in-between where the newly created node will always be a hidden node. The connection mutation adds a new connection gene between two nodes.

In order to apply crossover the authors introduce the notion of matching genes and non-matching genes. The matching genes are present in both genomes and are easily identified based on the innovation number of the gene. The non-matching genes are the genes that do not match in the middle. The excess the genes are the genes of the longest parent genome that have higher innovation number than the highest innovation number connection gene of the shortest parent. The matching genes are inherited randomly from either parent, which amounts to setting the weight of the corresponding connection to the weight setting of one of the parents. The distjoint genes and excess genes are inherited from the fittest of the two parents. Figure 2 shows an example of how the crossover operation is applied.

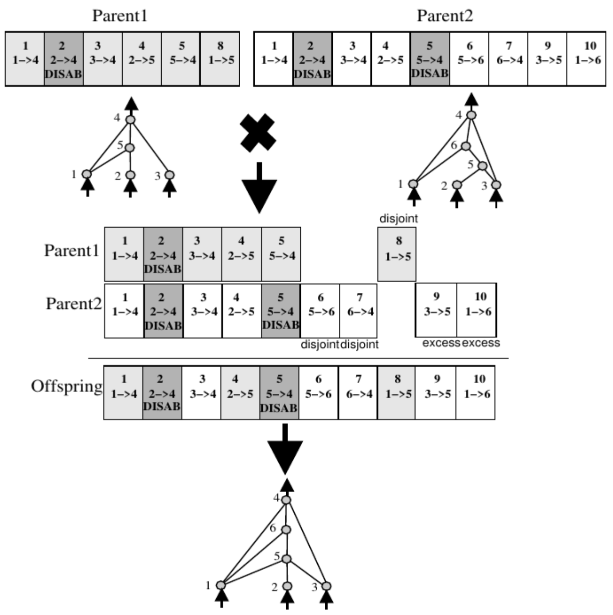


Figure 2: Example of NEAT crossover operation.

NEAT uses a speciation technique with the aim of protecting innovations and creating diversity in the population such that networks with a new kind of structure can be stepping stones on the way to the optimal network structure. The population is divided into a number of specifies and the individuals of the population only compete within each specifies (each niche). The authors define the compatibility distance between two individuals as:

$$\delta = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W} \tag{12}$$

where c_1, c_2, c_3 are some constants, E are the number of excess genes, D the number of disjoint genes and W the average weight differences of matching genes and N the number of genes in the largest of the two genomes. NEAT maintains a list of species ordered in ascending fitness wrt. the most fit genome in the species. In each generation the genomes are all placed into species by going through the ordered list of species and placing the current genome in the first species where the compatibility distance to the representative (most fit individual) of the species is below the compatibility threshold δ_t (a hyper-parameter). The number of species in total is a hyper-parameter of the algorithm and in each generation the individuals are placed in their closest species. If the compatibility distance is above the threshold for all current species a new species is created. In each generation species where the best fitness of the genomes has not improved in 10 generations is deemed stagnant and purged unless the total number of species would drop below a limit (hyper-parameter).

In order to determine the number of reproductions of each species in the population in a way that gives rise to a new population of the same size and also takes into account how well each species is performing relative to its size, Stanley and Miikkulainen use explicit fitness sharing. The adjusted fitness f_i' for a genome is computed as follows:

$$f_i' = \frac{f_i}{\sum_{j=1}^{n} sh(\delta(i,j))} \tag{13}$$

where

$$sh = \begin{cases} 0 & \text{if } \delta(i,j) > \delta_t \\ 1 & \text{otherwise} \end{cases} \tag{14}$$

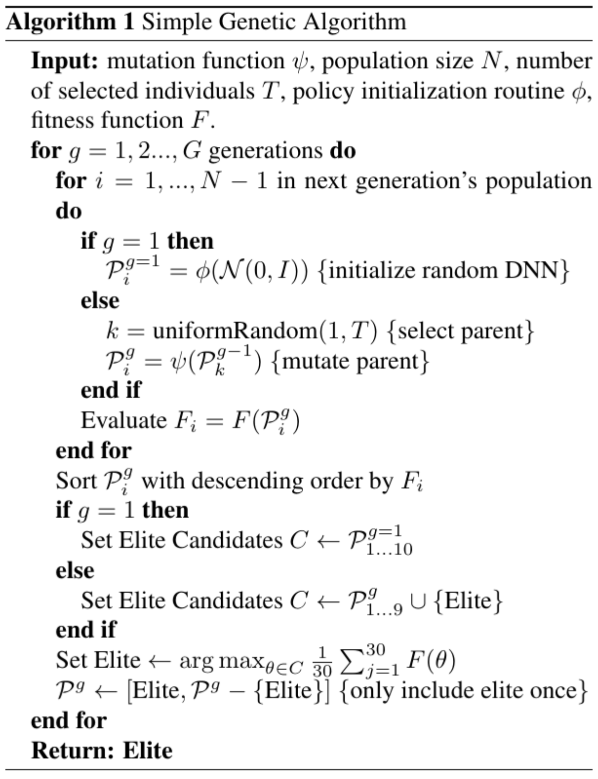
The number of reproductions of a species is then proportional to the sum of the adjusted fitness of its members. The lowest performing members in the species are purged and child genomes generated by randomly selecting among the remaining genomes.

An important aspect is that the population starts out minimally with a number of individuals in the population consisting of the genotype that encodes networks with only the input and output neurons. The hidden neurons will have to be added through the add node mutation operators. By starting out minimally the idea is to end up with as small networks that can accomplish the given task as possible. Simpler models are generally preferred over more complex ones if they have equal performance (law of parsimony) so this is a well-argued idea.

2.4 Simple Genetic Algorithm for RL

The algorithm the authors use in [24] on the Atari domain is a simple genetic algorithm (which we shall refer to as GA in the rest of this report) that maintains a population of genomes and over a number of generations adapts the population by selection and mutation. Cross-over is not used in this algorithm for simplicity reasons. Each genome is a parameter vector of the weights of all connections of the fully connected neural networks that the genomes represent. The selection scheme is truncation selection, where the top T individuals become the parents of the next generation. The population size is constant by producing a new population by selecting a parent at random and mutating it by additive Gaussian noise to the parameter vector. The weights for the networks in the initial population are initialised using Xavier initialisation for added stability.

The pseudocode for the algorithm is shown in figure 3



```
Algorithm 1 Simple Genetic Algorithm
  Input: mutation function $\psi$, population size $N$, number
  of selected individuals $T$, policy initialization routine $\phi$,
  fitness function $F$.
  for $g = 1, 2..., G$ generations do
    for $i = 1, ..., N-1$ in next generation's population
    do
      if $g = 1$ then
        $\mathcal{P}_i^{g=1} = \phi(\mathcal{N}(0, I))$ {initialize random DNN}
      else
        $k = \text{uniformRandom}(1, T)$ {select parent}
        $\mathcal{P}_i^g = \psi(\mathcal{P}_k^{g-1})$ {mutate parent}
      end if
      Evaluate $F_i = F(\mathcal{P}_i^g)$
    end for
    Sort $\mathcal{P}_i^g$ with descending order by $F_i$
    if $g = 1$ then
      Set Elite Candidates $C \leftarrow \mathcal{P}_{1...10}^{g=1}$
    else
      Set Elite Candidates $C \leftarrow \mathcal{P}_{1...9}^{g} \cup \{\text{Elite}\}$
    end if
    Set Elite $\leftarrow \arg\max_{\theta \in C} \frac{1}{30} \sum_{j=1}^{30} F(\theta)$
    $\mathcal{P}^g \leftarrow [\text{Elite}, \mathcal{P}^g - \{\text{Elite}\}]$ {only include elite once}
  end for
  Return: Elite
```

Figure 3: Pseudo-code for the simple genetic algorithm (GA)

The authors use a compression technique to encode the genomes in the population when they apply the algorithm on Atari domain problems for efficiency reasons. Instead of storing each individual as a parameter vector directly, they store it as an initialisation seed followed by a list of random seeds that produce the series of mutations a given genome; a technique which resembles the idea of indirect encoding. From this list of random seeds each network in the population can be reconstructed when necessary. This approach works well for genetic algorithms encoding neural networks as in the context of this paper. This is also a difference between NEAT and the GA, which will allow the GA to be applied to a much larger problem than NEAT is readily able to do.

2.5 Runtime Analysis of Randomised Search Heuristics

Runtime analysis is the study of the runtime of evolutionary algorithms. The aim is to derive bounds on the running time for algorithms on problems, which will characterise their efficiency as well as improve our understanding of their working principles despite their black-box optimisation nature. Runtime analysis can both be theoretical and empirical, where theoretical bounds are typically proved for certain classes of problems and empirical benchmarks either used to get a sense of the runtime during the process of working towards deriving a runtime bound or to solidify and verify theoretical claims.

Runtime analysis includes many techniques and uses tools from probability theory. These include among others, elementary probability theory results such as linearity of expectation and the waiting time argument; it also includes Chernoff bounds, the Markov inequality and the Chebyshev inequality as well as the coupon collectors theorem. Often the analysis of randomised search heuristics consists of finding a clever way to apply a small collection of simple tools much more often than having to prove deep theorems in probability theory. [3] Sometimes probabilistic theoretical results are also developed and new results shown as part of the analysis of a randomised search heuristic. For instance the theory of multiplicative drift has been used to simplify the derivation of the expected runtime bound $O(n \log n)$ of the evolutionary algorithm (1+1) EA on linear functions and since then developed further and used to refine the bound to reveal the constants hidden in the O-notation. [27]

2.6 (1+1) ES and CMA-ES

In the runtime analysis of the problems presented in 4.1 two algorithms will be relevant, namely the (1+1) ES as presented in [12] and Covariance matrix adaptation evolution strategy (CMA-ES) [11], which are both so-called evolution strategies and can be applied for continuous optimisation problems. Contrary to regular genetic algorithms the evolutionary strategy-based algorithms sample new candidate solutions according to some distributions, typically a multivariate normal distribution in $\mathbb{R}^n$. The parameters of the distribution like the mean and standard deviation of the normal distribution are updated as the algorithm adapts to the fitness environment. Mutation amounts to pertubating the offspring with typically a Gaussian mutation vector and multiplying with some mutation strength scalar. Both the (1+1) ES and CMA-ES use multivariate normal distributions, but CMA-ES self-adjusts the covariance matrix along the way. The (1+1) ES self-adjusts the mutation strength only. Listing 2 shows the (1+1) ES algorithm.

Algorithm 2 $(1+1)$ ES

```
1: Initialise c at random from standard normal distribution of $n$ dimensions
2: $t \leftarrow 0$
3: while stopping criterion not fulfilled do
4:     Choose a random mutation vector $m \in \mathbb{R}^n$ , where the distribution of m may depend on the course of the optimization process
5:     Generate the mutant $x \in \mathbb{R}^n$ by $x := c + m$
6:     If $f(x) \leq f(c)$ then x becomes the current individual ($c := x$), otherwise x is discarded (c unchanged)
7:     $t \leftarrow t + 1$
8: end while
```

The name of the algorithm (1+1) ES follows the convention as it generates one individual in each generation (the second number one in the parenthesis) and selects one individual to be the new current individual among the current and the one mutated (the first one in the parenthesis).

There exist different ways of adapting the mutation strength. A common way is to use the 1/5-rule, which works by observing the optimisation process for some number of steps without changing s and if more than one fifth of the steps in this observation phase have been successful, s is doubled, otherwise, s is halved. [12]

CMA-ES is much more advanced than (1+1) ES and achieves state of the art result on many kinds of non-convex optimisation problems.

3 Related Work

The work of this project draws heavy inspiration from the research conducted at Uber Labs by Ken Stanley, Joel Lehmann, Jeff Clune among others, who are among the leading researchers within the neuroevolution research field.

The main paper that lays the foundation for this study is [22], which gives an overview of the neuroevolution research field. It describes both the main idea, the motivation, the recent development and advancements as well as current research problems and future work.

As stated earlier one of the paradigms in which neuroevolution in recent years has achieved significant break throughs is the application of evolutionary algorithms for training neural networks on difficult reinforcement learning problems such as the Atari domain. The paper by Such et al. [24] with the simple genetic algorithm described in section 2.4 has been inspiration and motivation for implementing a simple genetic algorithm for reinforcement learning as well as studied as part of this project. The paper by [18] which motivated said paper has been an interesting alternative approach based on evolutionary strategies and strengthens the motivation for evolutionary algorithms and concepts applied in the reinforcement learning paradigm.

Novelty search originally introduced by Joel Lehman et al. in 2011 [14] and the variations of the ideas based on that concept has provide great inspiration for interesting experiments with search not directly aimed at optimising fitness as well as deeper insight to techniques and ideas that can be used in general in other areas of machine learning as well. Due to time restrictions novelty search has not been implemented as part of this project, however.

The article by Stanley and Miikkulainen on Neuroevolution for augmenting topologies has also been one of the important papers of this project and the NEAT algorithm therefore implemented and benchmarked in the project. Although it dates back to 2002 the ideas in the paper and the performance of the algorithm has made it a strong attribution to the neuroevolution repertoire of algorithms and the basis for later algorithms.

The pacman environment has been studied as part of a course of artificial intelligence at the university of Berkley [25] where it was used both for symbolic AI methods such as

A* search and reinforcement learning methods. This environment has not been analysed extensively in the literature in itself, but it is possible to find some open access references and work with the environment. In one report by a group of students who have taken the course at Berkley they apply Q-learning, approximate Q-learning by using feature engineering and deep Q-learning based on a pixel-based board representation. The students have experimented with different kinds of hand-crafted features and through an ablation analysis showed how the features helped performance of the overall. The ideas behind these features could be used in this project as well. The students conclude that the Q-learning approach does not scale well with the size of the level and the deep Q-learning algorithm has only been applied to smaller levels, due to extensive training time. Both of these results are as expected. [1]

Looking at the literature there does not seem to be much existing research on runtime analysis of neuroevolution. Currently as of writing Carsten Witt and Paul Fischer are researching the possibility of defining simple classification problems based on equidistributed points on a hyper-sphere and analyse the runtime of evolutionary algorithms applied to optimising the classification accuracy of a simple artificial neural network with one hidden neuron and possibly more advanced settings. The problems and results of such research could potentially lay a foundation for runtime analysis on other problems and neural network architectures evolved using evolutionary algorithms. Similar to the OneMax pseudo-boolean optimisation problem providing a common problem in runtime analysis of metaheursistics, the problems introduced could play a similar role within runtime analysis of neuroevolution. An attempt to contribute to this research is made as part of this project in section 4.1.

4 Analysis

4.1 Runtime Analysis

In this project we will consider two simple neuroevolution problems and analyse the runtime for a select few evolutionary algorithms on these problems. We first introduce the problems and then make an attempt to analyse them.

4.2 Problem Without Local Optima

Following the problem description in [28] we consider a set C_n of n points in a D-dimensional space that are (more or less) equidistantly placed on a hypersphere of radius 1 around the origin. For example, if D = 2 the points have polar coordinates $(1, 2\pi i/n)$ for $i = 0, \dots, n-1$, i. e., they are equidistantly placed on a circle around the origin. We divide C_n into two classes of roughly equal size by distinguishing according to their first dimension x_1 in Cartesian coordinates and calling points where $x_1 \geq 0$ white and black otherwise. The classification problem is simply to learn whether a given input $x_1, \dots, x_D$ is black or white, using C_n as a training set.

We simply consider a single neuron N with D inputs, weights $w_1, \dots, w_D$ and bias b along with a binary activation function, i. e., that fires and classifies an input $(x_1, \dots, x_D)$ as white if $i = w_i x_i \geq b$. Implicitly assuming that the neuron is evaluated on the whole training set C_n, we aim at minimizing the following error function:

$$f_n := |\{x \in C_n | \, N^* \text{classifies x wrongly}\}| \tag{15}$$

The function takes a minimum of 0 at, e.g. the weight vector $(1, 0, \dots, 0)$ and bias 0.

4.2.1 Problem Considerations

In the following we consider a simplified problem setting, where we let $b = 0$. A search point then corresponds to a vector of weights of the neural network. And the input consists of a vector of coordinates - one for each dimension. Since the weights are real numbers, the problem is continuous. Since the first input coordinate x_1 uniquely determines the class of the point, the search point $w^* = (1, 0, 0, \dots, 0)$ of length D will give an optimal solution and gives rise to a hyperplane of dimension $D - 1$ through the origin.

Concerning the property of points being equidistributed in an interval or surface, we know that a sequence $(s_1, s_2, s_3, \dots)$ of real numbers is said to be equidistributed, or uniformly distributed, if the proportion of terms falling in a subinterval is proportional to the length of that subinterval [26]. The points are therefore not necessarily spaced exactly evenly apart. For $D = 2$, however we can always achieve that they are exactly evenly separated by using polar coordinates (c.f. [28]). For $D = 3$ we can use one of the two strategies presented in [5]. For $D > 3$ we will use uniform random sampling as suggested in [28], which is also what [5] use in the first strategy for $D = 3$.

Related to this observation we also note that the number of points in class "white" is not uniquely determined by a specific setting of D and n, since there are multiple ways the points can be placed while still being equidistributed. Furthermore, a search point not exactly equal to w^* may still make the neural net classify all search points correctly and therefore be an optimal solution.

We note that the problem is a minimisation problem as we seek to minimise the error function. So lower fitness is better here. To convert the problem to a maximisation problem we could just look at the number of correctly classified points instead or multiplying the fitness function with -1 though.

In order to make the analysis easier for higher dimensions we will now let $n \to \infty$ and consider the fitness function:

$$f = \frac{vol(R_d \cap W_D)}{vol(W_D)} \tag{16}$$

where R_D and W_D are defined as follows: Let $v = (v_1, v_2, \dots, v_D)$ be the normal vector of the hyperplane H_D (where H_D is the hyperplane that we are optimising and that separates our prediction of the class frontier).

$$W_D = \{x_1, ..., x_n \mid x_1 >= 0 \, \wedge \, x_1^2 + \dots + x_n^2 = 1\}$$

$$R_D = \{x_1 + \dots + x_D \mid x_1 v_1 + \dots + x_D v_D \geq 0\}$$

One consequence of this approach, however, is that the problem then cannot really be discretised anymore, since either the neighbourhood would consist of infinitely many points or mutations would not yield significant progress towards an optimum.

We can map a search point, which is a weight setting, $w = (w_1, w_2, \dots, w_D)$ to the hyperplane H_D as the hyperplane is uniquely defined by w as normal vector.

4.2.2 Simplified Discrete Problem Adaptation

We first consider a discrete version of the problem and analyse the runtime of an evolutionary algorithm inspired by the (1+1) EA. There are a number of ways to make the problem discrete, which we describe in each subsection.

4.2.2.1 D = 2

We use the suggestion in [28] and let the search space be $S_n = \{0, 1, \dots, n-1\}$ and interpret $i \in S_n$ as polar coordinate $(1, 2\pi i/n)$ and map a polar coordinate to weight pair (w_1, w_2) corresponding to the line through $(1, 2\pi i/n)$ and $(1, \pi + 2\pi i/n)$. The EA then works on S_n by mutating the current point by uniformly going to a neighbour in the ± 1-neighbourhood (modulo n) and selecting it if its f-value is the same or lower than that of the parent. Initially the current search point $i \in S_n$ is chosen at random.

Given this simplification we see that in each iteration the probability of the EA choosing a neighbouring search point that brings the hyperplane closer to the x_1-axis corresponding to w^* is 1/2 since there are only two neighbouring points. For $D = 2$ it is easy to see that the fitness will not increase if that point is chosen as the child of the iteration. Choosing the other point p' cannot lower the fitness; it may stay the same, so the algorithm might go in the wrong direction one step, but in the next step it will then not be able to go further in the wrong direction, since p' will be classified wrongly then, making the fitness worse. So in the worst case the algorithm generates the mutant p' and accepts it as the next current search point and it then needs to get back to p and advance one or two more points in the right direction to not be able to get back to the current search point p again. With probability at least 1/8 the algorithm will do this, so we can use a waiting time argument to show the algorithm makes progress. The number of iterations is thereby $O(1)$. Since there are at most n such steps until the algorithm cannot do any better, the running time is $O(n)$.

4.2.2.2 D = 3

The discrete setting for $D = 3$ is more complex and we cannot readily use polar coordinates and consider neighbourhoods in the same manner as we did for the case $D = 2$. To investigate the runtime here it would be advantageous to compare with some empirical benchmarks to get an idea of whether or not it is worth pursuing the idea of a linear time complexity as was the case for $D = 3$. For fixed D it seems like it could be reasonable still to achieve a linear bound, but it seems very likely that that complexity increases with D. For $D = 3$ we may still be able to carry out a derivation in a way where we look at the neighbourhoods as for $D = 2$. We give no rigorous argument here for the runtime, but attempt to give a sketch with some ideas.

In order to proceed we define the search space slightly different to be the indices of the points and map a search point to a weight triple (w_1, w_2, w_3) by defining the hyperplane of dimension $D - 1 = 2$ to be given by the normal vector from the origin to the point (the position vector) and let the hyperplane pass through the origin. This defines a unique hyperplane of two dimensions.

As noted before we cannot assume that the points are exactly separated by the same distance on the hyper-sphere between all pairs of closest points and just use the general formulation for the equidistributed placement of the points on the hypersphere. We therefore must now define what the set of neighbouring points are for a point p. An idea is to look at the angle between the position vector of p and the position vector of the other points. Then choose the 5-6 closest points to be the neighbouring points as follows: Consider the hyperplane by disregarding one dimension (w.l.o.g. say x_1) and looking at all the points projected onto this 2-dimensional space. Choose the point among all points that has higher value of x_2 than p (if possible) where the angle is the smallest in the original space between the position vectors. Then do the same with the point with lower value of x_2. Do the same for dimension x_3 instead of x_2. Then let the next dimension be fixed and repeat. This way a total of up to six points can be chosen. If n is high enough at least five will be chosen in expectation (the sixth point may not be possible to find if e.g. p had $x_1 = 1$).

We conjecture the algorithm will thereby be able to make progress by tilting the hyperplane by choosing one of these neighbouring points or possibly achieve the same fitness but cannot go more than a constant number of steps astray. Since the number of neighbours is $O(1)$ a waiting time argument would again give $O(1)$ no. iterations before making progress and take some steps to neighbours to get to a point where it cannot go back to p since the fitness would become worse. It then follows that the total no. iterations is $O(n)$ as there are n points the algorithm needs to visit in the worst case. As mentioned above this argument is not rigorous enough to guarentee the correctness of the linear runtime bound. Note that the complexity of computing the neighbourhood points should also be properly taking into consideration in a real proof.

4.2.3 Continuous Problem Analysis

We now consider the original problem formulation where the problem is no longer discretised. Consider the (1+1) ES as defined in [12] in section 1.1 with the 1/5-rule. In particular we use a Gaussian mutation vector for $m \in \mathbb{R}^D$, which will be added to the current search point c when mutating to get the mutant/child x. According to proposition 3 in [12] a scaled Gaussian mutation vector is isotropically distributed meaning all directions are equiprobable. We know the optimal hyperplane is the one defined by the normal vector $(1, 0, \dots, 0)$. We make the observation that no matter the current rotation of the hyperplane of the current search point c we can turn it by pertubating the weights slightly to get a new search point x that will achieve better fitness than c w.r.t. the fitness function (16). In particular, the fitness can be improved by just changing one of the weights towards 0 for all other weights than w_1, which should be changed towards 1.

In order to derive a bound on the runtime an approach would be to use the results of [12] and give the same upper and lower bounds on the runtime, which is possible if we can prove that the theorems are applicable in our case. This then amounts to showing that the preconditions are satisfied. The paper considers the following function scneario:

Definition 1. *A function $f : \mathbb{R}^n \rightarrow R$ belongs to this class if*

- *a minimizer $x^* \in \mathbb{R}^n$ exists, i.e. $f(x^*)$ is minimal*
- *$d(x^*, x) < d(x^*, y) \Rightarrow f(x) < f(y)$ for any two points $x, y \in \mathbb{R}^n$*

Informally: unimodal functions that are monotone with respect to the distance from the minimum.

We conjecture that our fitness function belongs to this class with the following arguments: Starting with the first condition, we know that $w^* = (1, 0, \dots, 0)$ is an optimal weight setting that will make all points be classified correctly by the neural network, so a minimiser x^* exists such that $f(x^*)$ is optimal (minimal when considering the minimisation version of the problem). One important observation to make, however, is that there are an infinite amount of minimsers/optimal solutions as any $w' = (i, 0, 0, \dots, 0)$ for $i \geq 1$ would lead to a perfect classification of the all the points. Also, note that in practice a weight setting where w_1 is sufficiently high may allow other weights to be different from 0 and all points will still be classified correctly by virtue of the linear combination between the inputs and the weights. However, we may perhaps still argue that (16) belongs to the class above as long as the fitness landscape satisfies the second condition, so it is not necessarily problematic that there are many solutions, though this is something important to be aware of. Theoretically the hyperplane given by $x_1 = 1$ is the only optimal solution, so when thinking in terms of hyper planes there is also only a single solution.

Now for the second condition we need to choose a distance metric. Both the L1 and L2 norm might work here and generally we would claim that the problem is convex. In a rigorous proof this should be argued for though, but for now we conjecture that this is the case. For now it will also be assumed that we have a distance metric and can say that our fitness function (16) satisfies the definition. To make of for this we shall test some this empirically in section 7.

Finally we must argue that this holds regardless of the value of D. We may neglect n as we let $n \to \infty$ and look at fitness in terms of volume rather than the exact number of points classified correctly. For D=2 and D=3 one could argue that it can be seen without too much effort that we can turn the hypersphere and improve on f, which in turn, means we can pertubate the weights and get closer to the optimal solution using the (1+1) ES. For higher dimensions $D > 4$ there also does not at first glance seem to be a problem with drawing the same conclusion, though some arguments are needed.

4.2.3.1 Lower Bound

From [12] theorem 14 states:

Theorem 1. *Let the (1+1) ES minimize a function $f : \mathbb{R}^n \rightarrow R$ in the considered function scenario using isotropic mutations and elitist selection. Then the expected runtime, i.e. the expected number of f-evaluations, to reduce the initial approximation error to an α-fraction is $\Omega(\log 1/\alpha \cdot n)$ for $0 < \alpha \leq 1/2$*

If (16) is applicable here and we are using the (1+1) ES with isotropic mutations and elitist selection (we only have one search point and we choose the elite in each iteration to be the best search point between the current search point c and the mutant x) then the preconditions of the lower bound theorem apply and would give a lower bound on the runtime of $\Omega(\log 1/\alpha \cdot D)$ for $0 < \alpha \leq 1/2$. Notice that we have substituted D for n as we are in a dimensional space and the theorem uses n as the number of dimensions. However, it is not fully supported theoretically that the observations about (16) holds here which would make the theorem non-applicable. However, regardless, the upper bound of the next subsection may still apply if we pessimistically disregard all other optima than $w* = (1, 0, 0, \ldots, 0)$.

4.2.3.2 Upper Bound

From [12] theorem 19 states:

Theorem 2. *Let the (1+1) ES minimize a function $f : \mathbb{R}^n \rightarrow R$ in the considered function scenario using Gaussian mutations adapted by the 1/5-rule and elitist selection. Given that the initialization ensures $d_1/s = \Theta(n)$ the runtime/number of f-evaluations until the approximation error is no more than $\alpha \cdot d_1$ is w.o.p. (with overwhelming probability) $O(\log(1/\alpha) \cdot n)$, where $1/2 \geq \alpha = \exp(-n^{O(1)})$*

Here d_i denotes the distance from the optimum, i.e. the approximation error, at the beginning of the ith phase. A phase is in this context defined as n steps/iterations of the algorithm where the step size is held constant, which applies here as we use the 1/5-rule. Recall that the 1/5-rule reads: The scaling factor s is adapted after every n'th step; it is halved if less than n/5 of the respective last n steps have been successful, and otherwise doubled. A run of the (1+1) ES can thereby be seen as being partitioned into phases of n steps.

We again make a conjecture that (16) is applicable and we use Gaussian mutations. We also use the 1/5-rule and elitist selection. If we assume that $d_1/s = \Theta(n)$ is satisfied by the initialisation then we get the bound $O(\log(1/\alpha) \cdot n)$, where $1/2 \geq \alpha = \exp(-n^{O(1)})$. The details of the initialisation have not been sufficiently scrutinised to say whether or not the requirements to the initialisation are satisfied for our problem problem case, however, but the bound and theory behind it seems highly relevant here.

4.3 Problem With Local Optima

Another simple but interesting neuroevolultion optimisation problem proposed by Witt and Fischer [28] that features local optima can be defined as follows: Consider the unit circle and let the following sections, given in polar notation, be white:

- $(1, 0) - (1, \pi/3)$ (1–3 o'clock)
- $(1, 2\pi/3) - (1, \pi)$ (9–11 o'clock)
- $(1, 4\pi/3) - (1, 21\pi/12)$ (4.30–7 o'clock)

The rest of the points are black. We consider the setting without bias. This problem has two local optima and one global optimum in the fitness scape given by the number of classified points with respect to the angle of the hyper-plane through (0,0). These local optima will provide a challenge for hill climbing algorithms, but the (1+1) ES and CMA-ES may be able to escape the local optima by sufficiently strong mutations.

The first local optimum corresponds to normal vector (0, 1) (i. e., pointing to 12 o'clock and classifying the first and second quadrant positively. The second local optimum consists of the normal vector pointing at a time between 7:00 and 7:30 (green lines in Figure 1). Again $2\pi/3$ (4 hours). are correctly classified. The global optimum is achieved for the hyperplane that classifies the section from 1 o'clock to 7 o'clock positively (normal vector pointing at 4 o'clock), corresponding to a total length of $2\pi/3 + \pi/12$ (4:30 hours).

Figure **??** shows a plot of 1000 points equidistributed on the circle and coloured wrt. their class (yellow corresponding to white and purple corresponding to purple). The points are placed by uniformly at random choosing an angle in the interval $[[0, 2\pi]$ and the x and y coordinates then computed using cosine and sine respectively.

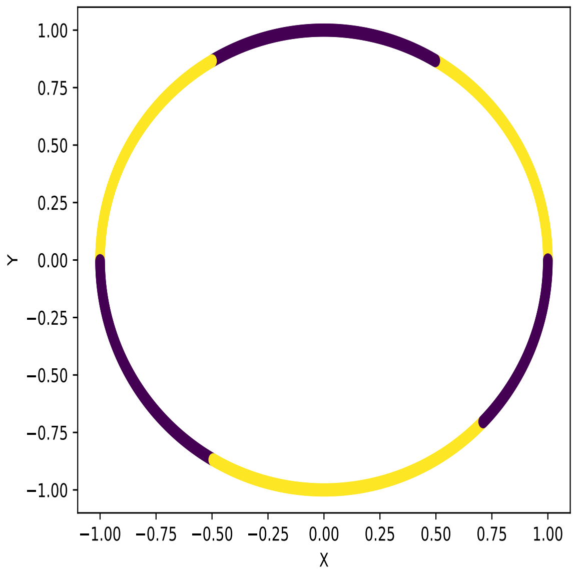


Figure 4: Depection of the points on the circle of each class. Yellow corresponds to the white class and purple to the black class.

We will analyse the runtime of the (1+1) ES and the CMA-ES empirically on this problem in section 7.

4.4 Reinforcement Learning

In order to show the efficiency and efficacy of the different solutions and how they compare a number of reinforcement learning problems are considered.

4.4.1 Cartpole Problem

The cartpole problem was originally formulated in 1983 by Barto and Sutton [4] and is about balancing a pole vertically on a cart that can move left or right. There exist a number of variations of this problem such as the double pole balancing task, which Stanley and Miikkulainen use to benchmark NEAT [21]. We shall in this project use the original cartpole formulation, which we characterise as follows:

- An observation is: Cart position x position, cart horizontal velocity, pole angle from vertical, pole angular velocity

- Actions available: Push cart left, push cart right
- Reward signal: +1 reward for every timestep
- Pre-termination: An episode ends either if pole angle is more than 12 degrees from vertical or if the cart position reaches end of display.
- Win condition: Episode length succeeds 500 timesteps. Originally the problem only requires 200 timesteps but we increase the difficulty by increasing the number to 500 timesteps.

Worth noting here is that the goal is to survive for as long as possible and there is no way to solve the level quickly and no different types of rewards.

Novelty search does not seem relevant for this problem as there does not seem to be multiple interesting ways of solving the problem with the optimal solution being to keep the post as upright as possible and the cart in the middle at all times. In principle it might be possible to instead tilt the pole back and forth with pendulum looking swings, but a 12 degrees maximum swing threshold before failing is not a lot to go by and such an approach is unlikely to be good in the long run.

4.4.2 Pacman Environment

Another reinforcement learning environment we shall try to benchmark on is an implementation of the classical game of pacman in which the agent controls pacman by moving around in a 2 dimeensional grid and must eat all food pellets and avoid getting caught by ghost agents unless pacman has eaten a capsule after which pacman will have a limited number of timesteps where it is possible to eat ghosts.

We use the problem as implemented at the Berkley university course on artificial intelligence [25] provided for this project with help and thanks to Tue Herlau. We characterise the environment more precisely as follows:

- An observation consists of: Pacman location and a list of wall, food pellet, ghost, and capsule locations. Locations are given as x-y coordinates.
- Actions available: Left, right, up, down, stop
- Reward signals: -1 for every time step, +500 for winning, -500 for losing, +10 for eating a food pellet, +200 for eating a scared ghost.
- Pre-termination: The agent loses as soon as a non-scared ghost enters the same tile as pacman.
- Win condition: All food-pellets have been eaten
- Extra notes: Notice that there is no time constraint on the agent completing a level; an agent can therefore get stuck and never solve a level. In order to deal with this we impose a time limit on the maximum number of timesteps an agent is allowed to use. If the agent does not solve the level within this time we give it a reward of -10000. This will deter agents from spending too long solving levels and to avoid problems with agents never solving a level. We shall vary the exact time limit depending on the complexity of the level (refer to section 7 for details).

We observe that contrary to the cartpole problem a good solution solves the level fast and the faster the better unless it pays off to try to eat a capsule and eat ghosts before finishing eating all the food. The problem is also clearly much harder than the cartpole problem. Since the goal of this project is investigate neuroevolution methods, principles and techniques, we will simplify the problem by carrying out some preprocessing of the inputs (see section 5).

The implementation by the course responsibles at Berkley University and features a level generator to make custom levels, which we shall use to make levels with various features and different complexity. There are are a number of ways an observation of a state can be represented. One idea would be to give the 2D grid tile values as input where a tile has a value encoding what kind of element is on the square like ghost, food pellet, wall etc. However this representation is the most raw state representation and would require the agent to learn how value translates to game elements and how each element affects the game and ultimately the rewards. A way to simplify the situation would be to preprocess the level and divide it into a tensor of dimension $6 \times width \times height$ with each of the 6 layers corresponding to a type of element in the level, so the first layer would be the location of walls (1 if there is a wall at position (i,j) and 0 otherwise), the next layer would be the location of food pellets and so on. This preprocessing steps would likely help making the problem easier for the agents as the translation from tile values is already made. However, there is still a great leap from this input to finding out which action to take in a state as the agent still has to learn useful features from this input. The problem can be simplified further by exploiting our domain knowledge and constructing a vector of useful features that directly model useful aspects of the environment. By performing feature engineering the agents can then focus on learning how the core game situations affect rewards. See section 5 for details on the feature engineering we perform.

For the pacman environment novelty search could potentially be helpful, since there are multiple ways to navigate the levels. There is also some clear trade-offs worth exploring, namely whether the agent should try to eat all the food pellets quickly or if the agent should try to get the capsules and eat some of the ghosts before eating the food. Also if the agent should play cautiously and only try to eat food if it there are no ghosts close by or if the agent should risk getting caught in obstacles in the level like dead ends in order to eat food quickly. Novelty search could help incentivising exploring these different types of behaviours by not focusing too much on the fact that the longer the agent takes to finish the levels the lower the reward unless it makes good use of the capsules.

5 Design

5.1 Achieving Equidistributed Points on a Hyper-Sphere

In order to investigate the runtime of the (1+1) ES and the CMA-ES algorithms and see how the analysis of section 4.1 holds up empirically we must first enable ourselves to implement the problems.

For the problem with out local optima we have for the special case $D = 2$ it is easy to achieve perfect equidistribution by using that the circumference is 2π, which we just divide by the number of points n to get the angle of the position vector of the point. For the special case $D = 3$ we can use one of the two generation methods described [5]. For the general case for any dimension $D \geq 2$ uniform sampling will work. Figure 5 show the uniform sampling method for $D = 3$ (left) and the random placement method of [5]. As we can see they both seem to generate an approximately equidistributed set of points on the hyper-sphere wrt. to the probabilistic interpretation of the notion of equidistribution.

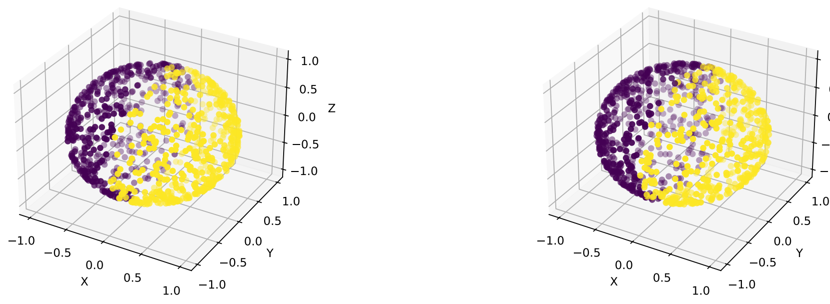


Figure 5: Generating points on the hyper-sphere. Two different sampling methods. Left: Uniform sampling on the hyper-sphere. Right: Random placement strategy in [5].

For the problem with local optima we only consider the 2 dimensional case. The generation of points can be done as described above by dividing 2π with n to get the angles of the position vector. The class labels are then defined based on the specification in 4.1 and a point is white if it lies within one of the three specified intervals, otherwise it is labelled black.

5.2 State Representation for Cartpole and Pacman

The cartpole problem is already a quite simple reinforcement learning problem and the raw observations of the environment already contain useful information. We shall therefore apply the algorithms directly given this as input.

The pacman problem is considerably more complex than the cartpole problem, and as stated earlier for this problem we will use two preprocessing steps when the agent should choose an action that will help achieve faster convergence for the algorithms learning. We shall 1) allow the agent to know which actions are legal in a given state and 2) preprocess a state and action pair by extracting useful features. By only considering the legal actions in a state, the agent can be said to exploit some knowledge of the dynamics of the MDP as it will then never choose an invalid action and thereby not need to learn the q-function for that state and action pair. This relaxation of the problem only abstracts away the aspect that the walls are impenetrable in the game and the agent will still need to learn how to navigate sophistically in the levels by choosing among the valid move actions. Preprocessing the state and action pairs by feature engineering is a simplification and abstract away a lot of the the complexity of understanding the environment in terms of higher level features, which is an aspect of the learning process that the agent would otherwise have had to do. The preprocessing is expected to make convergence of the algorithms much faster and will allow for more experimentation and easier success within the time limit of the project.

We construct a feature vector with the following features:

- Number of ghosts one step away from the tile the agent will end up in
- Distance to nearest food pellet from the tile the agent will end up in
- Distance to nearest scared ghost from the tile the agent will end up in
- Whether or not a food is eaten at the tile the agent will end up in
- Whether or not a scared ghost is eaten at the tile the agent will end up in
- A bias input of constant value 1.0

There are multiple features we could have chosen, but these features should contain enough information about a given environment state to make the agent able to choose reasonable actions. We expect the agent to learn to minimise the distance to food and avoid the ghosts close by. The features used here are made based on intuition and knowledge about the game mechanics and minor inspiration from the Berkley university course student project [1]. One could do experiments with the effect of each of these features and potentially construct more such as the average distance to food pellets. A feature that might be very useful as well, which has not been used in the project would be whether or not a tile with an active non-scared ghost is entered to really help the agent avoiding ghosts. The feature with the number of ghosts on adjacent tiles is capturing some of this, but in a slightly different less direct way.

5.3 (1+1) ES with 1/5th Rule

As described earlier we use the 1/5 rule for self-adjusting the mutation strength for the (1+1) ES. We implement the 1/5 rule using the approach in [2], which adapts s in every iteration. In the paper Auger also benchmarks two variants. In Variant 1, in case of equality between the objective function values of the offspring and parent the step-size stays constant and the current estimate of the solution stays constant as well. In the second variant, in case of equality between the objective function values of the offspring and parent, the step-size stays constant but the individual in the next generation is set to the value of the offspring. We limit ourselves to only benchmarking the main formulation of the (1+1) ES algorithm with the 1/5 rule.

6 Implementation

6.1 Program Structure

The code for the project implementation is structured in two main parts: One part for the runtime analysis and one part for the reinforcement learning related implementation.

6.1.1 Reinforcement Learning Part

For the reinforcement learning part the program can be run using the terminal with a standard library python parser for arguments such as the algorithm to use, the problem to use (either cartpole or pacman, the number of iterations/generations and the device (either CPU or Cuda). The DQN and GA implementations each have their own files while NEAT encompasses multiple files for various aspects of the algorithm. All the algorithms are subclasses of the abstract Agent class which defines the two required functions to implement: train and run_model, which run the training loop and run a trained model, respectively. A utils class provides auxiliary functions and constants and the feature extraction for the pacman environment. During training models are saved if they surpass previous found models in fitness.

The implementation uses the PyTorch library for neural networks, which provides a suite of classes and functions for using neural networks effeciently in python. DQN and GA extend the module class and use linear layer submodules and activation functions of the torch.nn submodule. NEAT is implemented using PyTorch as well and also extends the module class but the forward pass is implemented lightly differently as explained in section 6.2. PyTorch allows for easily running the code using Cuda, which can provide significant speed-up gains compared to running the code on a CPU depending on the nature of the implementation.

The cartpole problem is used by using the gym package made by OpenAI. The pacman environment is used by including the source code files for the implementation of the game made for the artificial intelligence course at Berkley University.

Inspiration for the implementation of DQN is a tutorial guide on PyTorch's website `https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html` (which is a pixel-based variant, however), for the GA some inspiration is drawn from [13] and for the implementation of NEAT inspiration has been gathered from `https://github.com/ddehueck/pytorch-neat`.

6.1.2 Runtime Analysis Part

The runtime analysis part of the project consists of an implementation of the (1+1) ES, a wrapper for CMA-ES that uses the pycma [10] module and classes for encoding the two problems considered in this project.

6.2 NEAT

As NEAT dynamically changes the neural networks (phenotypes) of the population during runtime by adding more hidden nodes and connections, it is not readily possible to simply implement the feedforward neural network in the PyTorch framework like it is for DQN and GA. In order to still leverage the features of PyTorch and make it compatibility with the implementation of the rest of the algorithms, we treat each neuron as a linear PyTorch module component with the number of input equal to the number of connections for that node and one output. In the forward pass, a map is built that maps each neuron to an output of that neuron. The neurons are sorted in topological order which is possible by virtue of the network being feed-forward without any backward connections. The outputs are then propagated through the network by building a tensor which is fed the input followed by applying an activation function. The output is then recorded in the map. In the end, the outputs of the output neurons are saved as a tensor and returned. Note that this is not a form of backpropagation; merely a forward pass.

This way of implementing the forward pass has the advantage of adhering to the format of the PyTorch interface for modules and it is therefore easy to integrate the algorithm into a training loop and running it either on the CPU og on the GPU on a given problem like for DQN and GA. The main drawback of this implementation is that makes multiple small propagations of input through the neurons instead of taking advantage of matrix multiplication of inputs from one layer to the next like for instance it is done with computations between fully connected linear layers. It may be possible to implement the forward pass to identify layers and the existent connections between the layers and then do computations by matrix multiplication to get the outputs of many units at once, but the simpler implementation has been chosen here. Another drawback is that we topologically sort the nodes and create tensors and other auxiliary objects to carry out the computations in every forward pass to handle the dynamics of the network. Since forward passes of the networks are made frequently extra time consuming operations will have an impact.

6.3 Input and Output of DQN and GA Neural Networks

Implementing a neural network for outputting the expected value of a state and action pair, $Q(s, a)$ directly based on the theoretical definition entails a need to run a forward pass of the network for each possible action in a given state. However, this can be optimised to only require a single forward pass of the network by letting the input only be the state and the number of outputs equal the number of actions. The output then corresponds to the value of each of the actions in the state and the best action can be computed as the maximum utility over the outputs.

6.4 Equidistributed Sampling

The implementation of the sampling of points on the hyper-sphere for D dimensions is carried out with the code below:

```python
def generate_points(self) -> List[np.ndarray]:
    points = []
    for i in range(self.N):
    v_prime = np.random.normal(0, 1, size=self.D)
    v = v_prime / (np.linalg.norm(v_prime))
    points.append(v)
    return points

```

We sample from a standard normal distribution for each of the D dimensions to get a vector in the D-dimensional space. After this the vector is divided to have unit length, so the point will lie on the hyper-sphere.

For the two-dimensional special case the implementation as stated earlier can be done as follows:

```python
def generate_points(self) ->  List[np.ndarray]:
    points = []
    for phi in np.linspace(0, 2*math.pi, num=self.N):
    x = math.cos(phi)
    y = math.sin(phi)
    point = np.array([x, y])
    points.append(point)
    return points
```

where we use the linspace function of numpy to achieve equidistributed angles, which are then translated to x and y coordinates.

7 Evaluation and Results

We now run benchmarks for the runtime analysis and evaluate the performance of NEAT, GA and DQN algorithms on the cartpole balancing and pacman problems. The main hardware specification used for all of the following experiments is as follows: An Intel i5-4690K quad-core 6M Cache CPU, Asus GeForce RTX 2060 SUPER GPU (with 8GB GDDR6 VRAM) and 16 GB 1833MHz DDR3 RAM.

For the reinforcement learning part we will allow ourselves to use the terms reward and fitness somewhat interchangeably as long as it is clear from the context what is meant, as fitness is a common term for evolutionary algorithms for how well they are performing.

7.1 Runtime Analysis

We initially consider the runtime of the (1+1) ES on the neuroevolution problem with no local optima, where we both benchmark with respect to the number of points n and the dimension D. Following this we empirically study the runtime for the neuroevolution problem with local optima.

7.1.1 (1+1) ES on Problem with No Local Optima

From the analysis in section 4.1 we expect the runtime as a function of the number of points n to increase almost linearly for fixed D. Figure 6 (left) shows the number of f-evaluations used to find the optimum solution as a function of n. The number of f-evaluations increases quite rapidly from 100 to 200 points, but after this seems to increase more consistently in a linear fashion. Averaging over more samples would likely make the increase less jittery. The runtime in seconds to find the optimum for each n gives fewer fluctuations and also seem to increase linearly for the values of n tested.

Next we run the algorithm for the problem in three dimensions shown in figure 7. On the plot to the right of the runtime in seconds the increase again seems linear except for the deviation at $n = 800$ where the algorithm spend considerably longer in one of the runs. It is hard to interpret the results of the plot from the left, but neither of the plots suggest that the runtime increases exponentially at least.

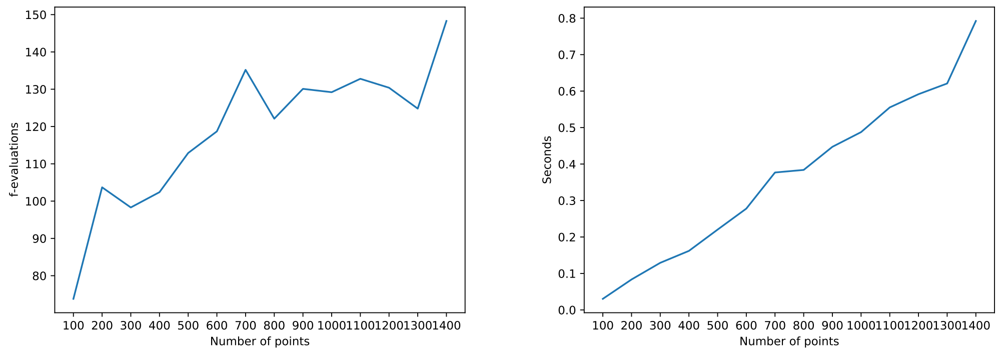


Figure 6: (1+1) ES, $D = 2$. Averaged over 20 runs for each value of n.

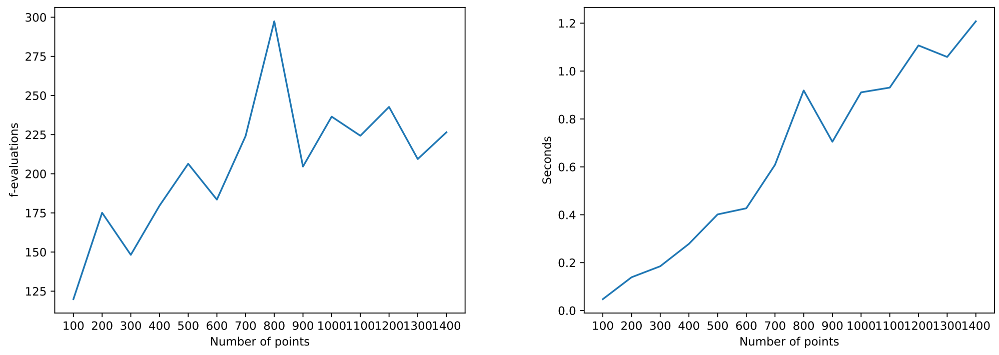


Figure 7: (1+1) ES, $D = 3$. Averaged over 20 runs for each value of n.

Finally consider figure 8 that shows the result of running (1+1) ES for $n = 1000$ for different values of D. We first observe the massive spike for $D = 8$ in the left plot, which slightly obfuscates the asymptotic increase, and to a lesser degree in the plot to the right. The reason for this is that the algorithm failed to find the optimum in one of the 10 samples averaged over to get the benchmark value for $D = 8$ within the maximum number of iterations set to 10000. Interesting to observe is how the seconds per iteration of the algorithm increases in a non-linear fashion as seen by the right plot. The time spent per iteration is much higher than for $D = 2$ for instance. This may be explained by the increased number of function evaluations spent as seen in the left plot as well as the time it takes to perform mutations and computing the fitness increasing with the number of dimensions.

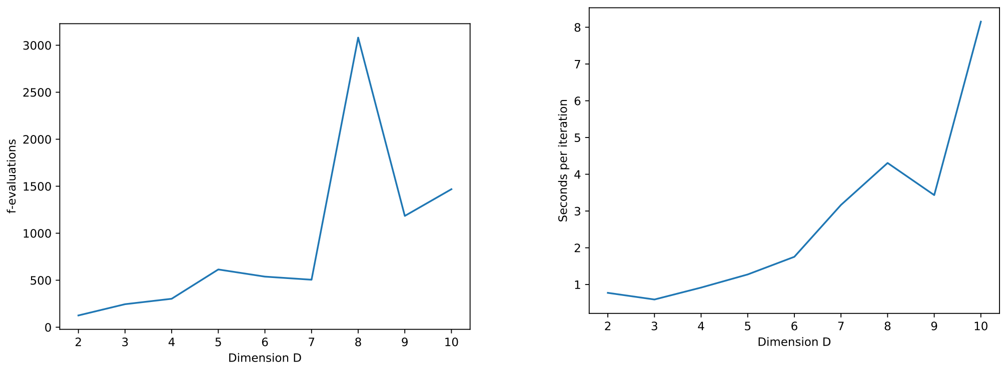


Figure 8: Left: For each value of D the algorithm is run 10 times and the number of f evaluations then averaged. The algorithm found the optimum 10/10 times for all values D except $D = 8$ where it did not find a solution within the maximum number of 10000 iterations allotted in one of the 10 runs, hence the massive spike. Right:

7.1.2 Problem 2 with Local Optima

Figure 9 and 10 show the result of running (1+1) ES and CMA-ES on the problem with local optima. The left plots show the average fitness achieved and the right plot the number of times the optimum was found before terminating.

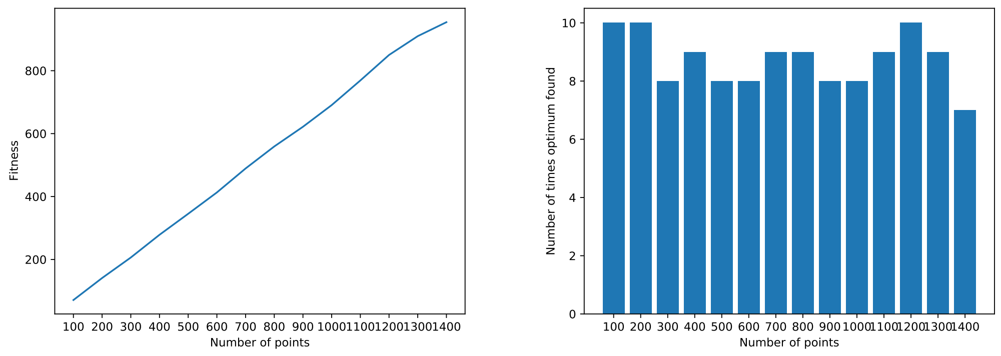


Figure 9: (1+1) ES, averaged over 10 runs for each value of n.

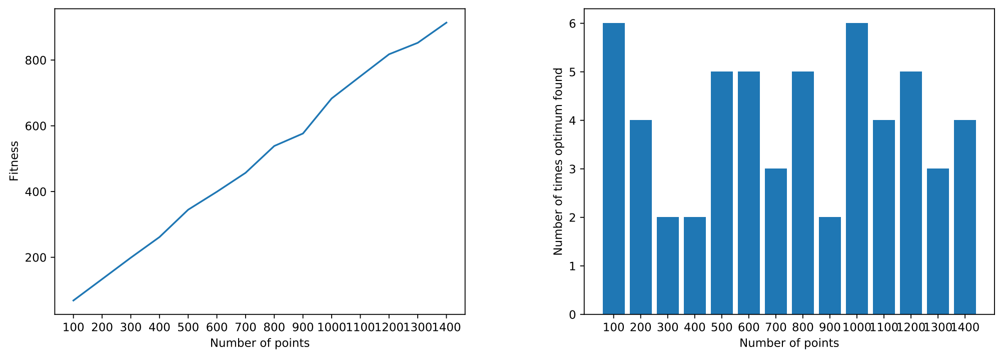


Figure 10: CMA-ES, averaged over 10 runs for each value of n.

We see that the fitness increases linearly for both algorithms as a function of n, which makes sense as the fitness of the optimum solution increases linearly in fitness with n as well. The (1+1) ES seems to find the optimum solution for all the tested values of n around 80 % of the time and gets stuck in a local optimum 20 % of the time. We observe that the number of times it finds the optimum solution seems to be unaffected by n, which makes sense as the fitness landscape has the same structure when looking at the fitness as a function of the angle of the hyper-plane.

The result for CMA-ES is to a large the degree the same, though it varies more in how often it finds the optimum. The reason CMA-ES finds the optimum fewer times on average than (1+1) ES for the problem may be explained by the settings of the CMA-ES algorithm, where for the test no changes were made in the termination treshold for the algorithm and it may therefore be content with a local optimum solution. Changing the settings of the pycma CMA-ES interface may change this result.

7.2 Reinforcement Learning Problems

To benchmark the three algorithms in a way that allows for a fair comparison we must restrict the maximum training time for each algorithm by the same amount. There are a couple of options for what metric we impose the constraint on. We could bound on iterations, but since DQN uses the notion of episodes whereas NEAT and GA use generations there is not readily a way to set each of these so they match in a fair way. Moreover, because NEAT and GA use a population of different networks and sizes the computation time for each generation will also differ. Another option would be to limit the number of times the fitness function is evaluated corresponding to the number of times an episode is generated of the environment MDP, but these epsiodes may be of vastly different lengths depending on how long the epsiode takes to complete based on the actions of the policy of the agent as well as the amount of computation time each of the algorithms spends other than invoking the fitness function. We shall therefore set the maximum training time for each algorithm to be a fixed number of seconds as this metric is disregarding the differences just mentioned. We set a value for the time limit based on the complexity of the problem in question.

Another concern we address is the effect of the random initialisations of the algorithms. The weights of the networks in the algorithms are randomly initialised and as a result the initial networks may be better or worse. It is know from other research that Q-learning methods can have problems stabilising and converge to a good solution if the initialisation is poor and it is therefore common when applying DQN to restart the algorithm after some number of episodes of training if the average reward is below a certain threshold. Ideally we should restrict the training time of the algorithms across all restarts made, but for simplicity we will allow each algorithm that they are reset in case the fitness is not above some low threshold that normally should be easy to beat within a reasonable number of episodes and generations. Since the time spent tuning hyper-parameters for each algorithm is also not prioritised (refer to the following), this is deemed a reasonable approach.

7.2.1 Cartpole Problem

We set the maximum running time to 2 minutes. We allow DQN to restart maximum 3 times if the mean reward after 100 epochs is below 30. We allow GA and NEAT to restart if their mean fitness/reward are below 30 after 2 generations. These values are chosen based on some quick preliminary testing and should normally not incur the need for a restart.

For the hyper-parameters we will for DQN base these around the guide at PyTorch website `https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html`. Since that version of DQN uses images as input instead of pixels, however, the number of hidden units can likely be set lower. We therefore carry out a quick test with three settings to see which value we should use in the following. Table 1 shows the results. We see that 64 hidden units gives the highest validation performance, so we pick this number.

	No. hidden units		
	32	64	128
Validation mean reward	221.3	500.0	467.3

Table 1: DQN. Averaged over 10 runs.

For NEAT we will use the hyper-parameter for cartpole used by `https://github.com/ddehueck/pytorch-neat` and for GA we will use the setting in [13].

The final hyper-parameter setting is given below (refer to the source code of the implementation of the project for details of the precise meaning of each parameter):

```
# DQN
Optimizer: Adam
NUM_HIDDEN_UNITS = 64
BATCH_SIZE = 64
GAMMA = 0.999
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 1000
TARGET_UPDATE = 10
REPLAY_MEMORY_CAPACITY = 1000
LEARNING_RATE = 0.001

```

```
# GA
MUTATION_STRENGTH = 0.01
POPULATION_SIZE = 50
NUM_SELECTED_INDIVIDUALS = 10
NUM_HIDDEN_UNITS = 32

```

```
# NEAT
USE_BIAS = True
ACTIVATION = 'sigmoid'
POPULATION_SIZE = 150
VERBOSE = True
FITNESS_THRESHOLD = 1000
CONNECTION_MUTATION_RATE = 0.80
CONNECTION_PERTURBATION_RATE = 0.90
ADD_NODE_MUTATION_RATE = 0.5
ADD_CONNECTION_MUTATION_RATE = 0.5
PERCENTAGE_TO_SAVE = 0.80
SPECIATION_THRESHOLD = 3.0
CROSSOVER_REENABLE_CONNECTION_GENE_RATE = 0.25

```

Figure 11, 12 and 13 show plots of the average reward obtained during the training phase for DQN, GA and NEAT, respectively, on the Open AI Gym CartPole-v1 problem from 2 minutes of training.

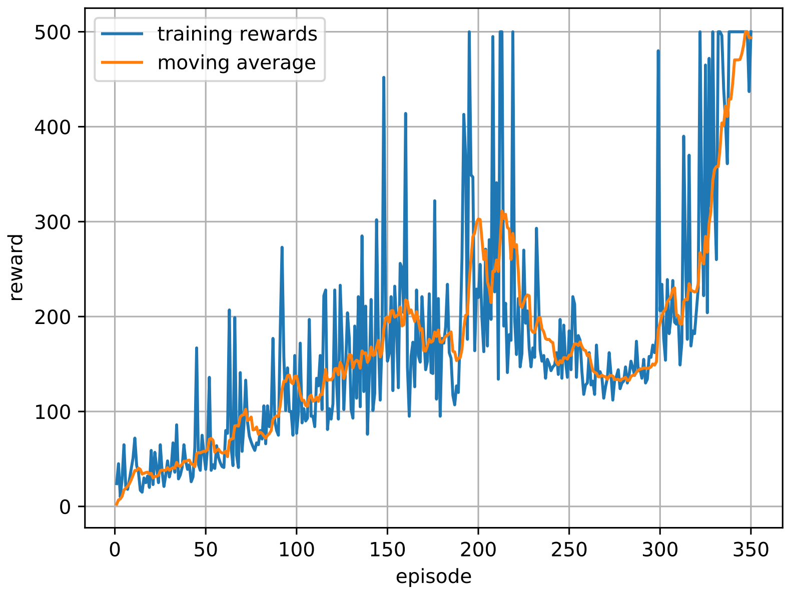


Figure 11: DQN on cartpole

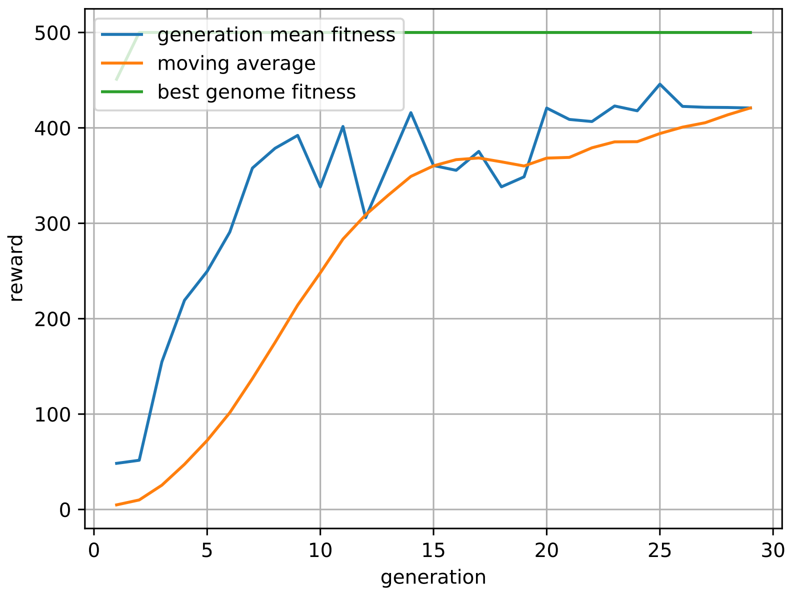


Figure 12: GA on cartpole

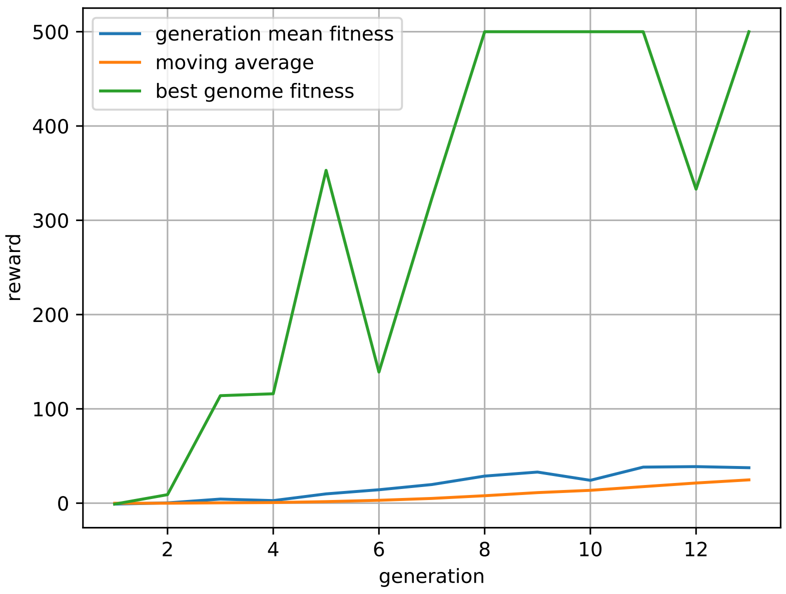


Figure 13: NEAT on cartpole

Looking at figure 11 we see that the training reward after the 350 episodes it completed in 2 minutes is around 500. However, it already hit the maximum around 200 episodes, but then the training rewards dropped quite significantly before increasing again. This observations shows that DQN can be unstable during training, which is also a well known problem on hard problems in the literature [15]. Even for this simple problem some instability can be observed. This could likely be remedied by better tuning of the hyper-parameters or more tricks to decorrelate the trajectories.

From figure 12 we see that GA is able to find an optimal solution very quickly after just a couple of generations as seen by the best genome fitness being 500. The generation mean reward increases steadily after this as well.

Comparing this to figure 13 of the performance of NEAT we see that the mean reward of the population is much lower and it takes longer before a network that achieves the maximum possible reward is found. We also see that the generations for GA take approximately half as long compared to NEAT as GA can complete twice as many generations in the same amount of time.

Next we evaluate the validation performance by running the policy given by the best network produced by each algorithm 100 times on independent instances of the cartpole problem and compute the average reward obtained. The results are shown in table 2.

Metric	DQN	GA	NEAT
Validation mean reward	467.85	487.75	500.0
Validation std reward	84.78	61.05	0.0
Validation max reward	500.0	500.0	500.0
Validation min reward	189.0	97.0	500.0

Table 2: Algorithm validation performance on cartpole problem. Maximum possible reward is 500. Averaged over 100 runs.

From table 2 we see that all three algorithms achieve a high mean reward close to the 500 maximum possible score. NEAT achieves the best result and is able to get the highest possible reward in all 100 runs with GA being second best. This result confirms that neural networks evolved using evolutionary algorithms can achieve good results for a simple reinforcement learning problems and be competitive with some of the classical reinforcement learning methods. The result thereby shows that evolving the topology and the weights of the network during training instead of backpropagation and also just by evolving the weights can lead to good solutions.

All the three algorithms were able to achieve the maximum reward in at least one of the runs. However, despite GA finding an optimum solution quickly during training, the final network was not as consistently high performing as the one NEAT found. This can be explained by GA starting out with networks that already have 32 neurons and by pertubating the weights in a way that mimicks random search in the beginning finds a good solution. NEAT more systematically builds up the network topology which takes longer but ends up with a better solution. The number of hidden neurons of the best network NEAT produced 23, which also shows that a lower number of hidden neurons can be more effective here. The fact that the mean fitness of the population in NEAT is much lower than the mean fitness of the population of GA makes sense considering NEAT using the notion of species. It is part of the algorithm to try to maintain diversity, which here seems to make the fitness of the some species and genomes quite low, but the best network produced can potentially benefit from this.

7.2.2 Pacman

While pacman is an entirely different domain from cartpole with other kinds of features, we shall use a similar set of hyper-parameters for the algorithms as we did for cartpole. The differences from the setting from cartpole are listed below:

```
# DQN
NUM_HIDDEN_UNITS = 128
```

```
# GA
NUM_HIDDEN_UNITS = 64
MUTATION_STRENGTH = 0.1
```

```
# NEAT
POPULATION_SIZE = 75
```

The number of hidden units has been increased for DQN and GA to allow for a greater potential for capturing the relationship of the features. For GA we increase the mutation strength to allow for more significant changes and exploration of the space of possible weight settings. For NEAT the population size is halved in order to decrease the time it takes to complete a generation.

One reason for the decision to only change the hyper-parameters to a low degree is that the feature engineering we employ described in section 5 simplifies the problem to a large degree. As the input features are highly usable already and there are not too many inputs, we expect simple networks to still have a good potential for doing well. Though the problem is definitely still more complex than the cartpole problem. Another reason is that there are many hyper-parameters in each algorithm that can be tweaked so an extensive search for optimal hyper-parameters will be time consuming. Furthermore, in order for the benchmarks to allow for a fair comparison of the algorithms, investing a lot of time tweaking hyper-parameters for one algorithm warrants the need to do a similarly thorough hyper-parameter search for the other algorithms as well. Otherwise the hyper-parameter setting aspect will play a bigger role. Of course, ideally, a search for the best hyper-parameters should be performed for each algorithm and then the benchmarks performed, but the choice has been made to spend time and effort on the project elsewhere.

For the pacman levels we will not allow any restarts as the levels change and it is difficult to find a good minimum value to say that the algorithm is allowed another run. There are also many tests to run and it will be the most fair when comparing the algorithm to not allow any of the algorithms an advantage of a restart.

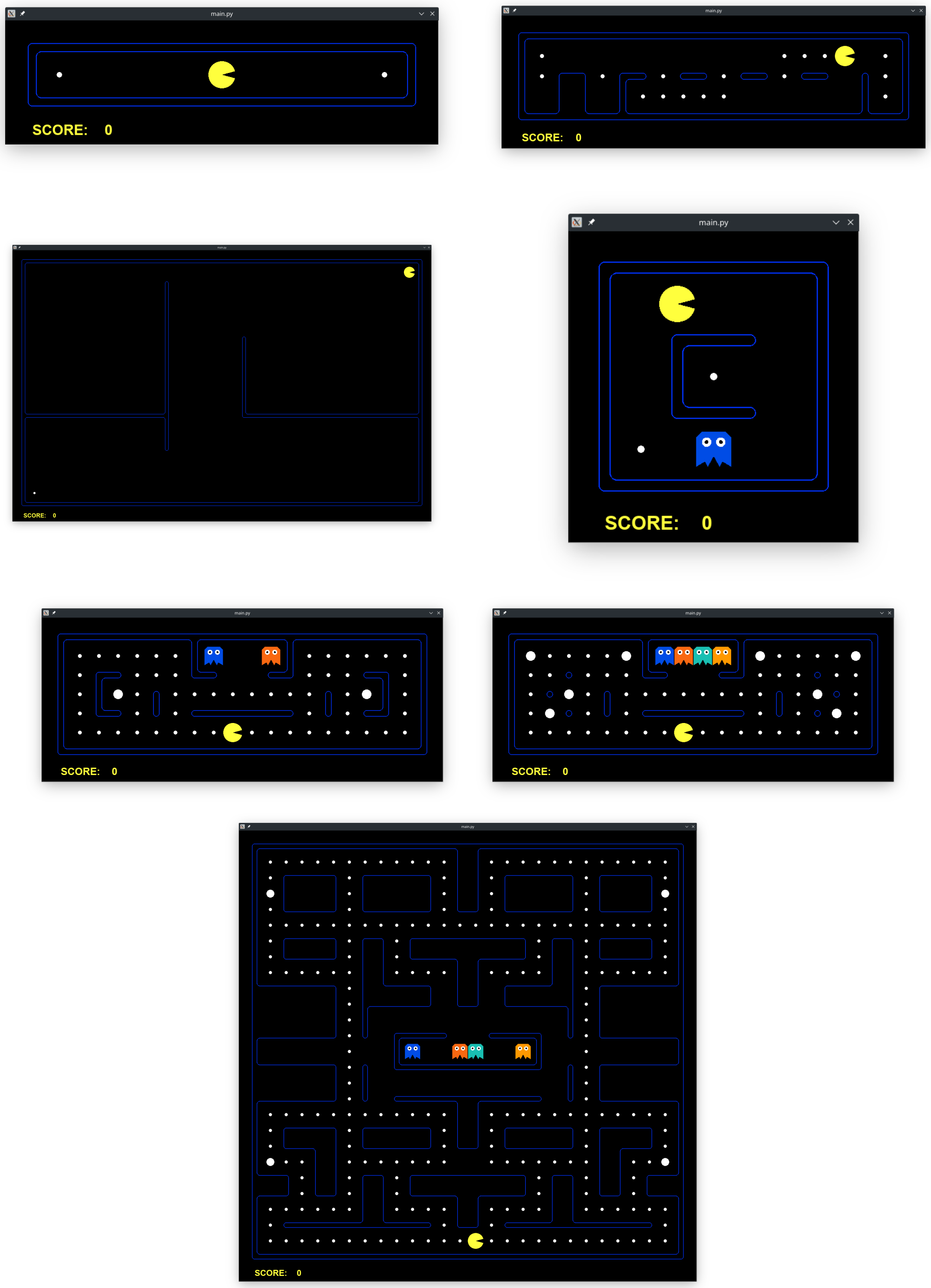


Figure 14: The pacman benchmark levels. From the top going left to right rowwise are: `customLine`, `smallSearch`, `openMaze`, `smallGrid`, `smallClassic`, `powerClassic`, `originalClassic`.

We shall benchmark the algorithms on a variety of levels with different complexity and features. Figure 14 shows a screenshot of 7 pacman levels we will use for the benchmarks. The levels have except for the `customLine` level been chosen among the many existing levels from the Berkley pacman game repository and the names used here refer to the respective level filenames. The levels have the following features:

- `customLine.lay` is a custom level made for the purpose of this project with a narrow tunnel with one food pellet on either end. The agent can only go left and right because of the narrow tunnel layout. This will provide a very basic test to see if the algorithms able to go to one end and then the other. A random policy will solve this level easily, but would not achieve the optimum solution on average most likely.
- `openMaze.lay` is a simple level with no ghosts and one food pellet. The agent will still have to move around the walls to get the food.
- `smallSearch.lay` features multiple food pellets and no ghosts. The agent should only concern itself with getting all the food quickly and it is best to start eating the food to the right first.
- `smallGrid.lay` has only two food pellets but one ghost the agent needs to avoid.
- `smallClassic.lay` is a small instance mimicking a classic layout of the pacman game with both multiple food pellets, ghosts and some capsules the agent can use.
- `powerClassic.lay` is similar in layout to `smallClassic.lay` but has many capsules. This will test the ability of the agent to make the most of the capsules available to hunt ghosts to score extra points instead of only going after the food pellets.
- `originalClassic.lay` is the original pacman game layout.

We will benchmark the algorithms by allowing them to train for 5 minutes on each level followed by computing their validation score by running the best policy given by the best network from training on the environment for the level in question 25 times and computing the mean reward received. For each level we the algorithms are reset before training, so the training phases are independent of each other. For the levels `customLine` and `openMaze` we set the time limit of the game to 100 timesteps, for `originalClassic` we set the time limit to 1000 timesteps and for the rest of the levels we set it to 500. These timelimits should give the agent amble time to complete the respective levels, but not bring the training to a halt whenever the agent does not complete the levels.

The benchmark results are given in tables 3, 4, 5 for DQN, GA and NEAT, respectively.

	DQN			
Level	Mean	Std	Max	Min
`customLine.lay`	502.0	0.0	502.0	502.0
`openMaze.lay`	456.0	0.0	456.0	456.0
`smallSearch.lay`	-10500.0	0.0	-10500.0	-10500.0
`smallGrid.lay`	-58.68	501.42	507.0	-504.0
`smallClassic.lay`	55.72	528.27	987.0	-358.0
`powerClassic.lay`	126.92	515.69	1345.0	-360.0
`originalClassic.lay`	508.72	1035.94	3596.0	-365.0

Table 3: Validation mean reward for DQN on Pacman for various levels. Training time 5 minutes. Averaged over 25 trials.

	GA			
Level	Mean	Std	Max	Min
`customLine.lay`	502.0	0.0	502.0	502.0
`openMaze.lay`	-10100.0	0.0	-10100.0	-10100.0
`smallSearch.lay`	622.0	0.0	622.0	622.0
`smallGrid.lay`	-139.56	484.92	507.0	-504.0
`smallClassic.lay`	724.16	500.69	987.0	-361.0
`powerClassic.lay`	421.2	594.97	1316.0	-324.0
`originalClassic.lay`	656.28	714.28	2535.0	-231.0

Table 4: Validation mean reward for GA on Pacman for various levels. Training time 5 minutes. Averaged over 25 trials.

	NEAT			
Level	Mean	Std	Max	Min
`customLine.lay`	502.0	0.0	502.0	502.0
`openMaze.lay`	456.0	0.0	456.0	456.0
`smallSearch.lay`	622.0	0.0	622.0	622.0
`smallGrid.lay`	58.96	499.02	507.0	-504.0
`smallClassic.lay`	-238.36	273.26	987.0	-446.0
`powerClassic.lay`	767.04	550.52	1660.0	-254.0
`originalClassic.lay`	2292.84	1171.51	3423.0	-196.0

Table 5: Validation mean reward for NEAT on Pacman for various levels. Training time 5 minutes. Averaged over 25 trials.

Looking at the results we see that there is no algorithm with the best mean reward score across all levels. However, NEAT, is the best overall on average and is only worse on the level `smallClassic`, where it is actually both worse than DQN and GA. However, NEAT is able to find a solution just as good as DQN and GA in at least one of the 25 trials judging by the max score for that level.

All the algorithms are able to consistently solve `customLine`, which is arguably the easiest of the levels and shows that the algorithms do work properly and the policy is not just random.

The difficult aspect of the `openMaze` level is the initial exploration of the state space before the agent gets to the tile with the single food pellet located quite a distance away from the agent. GA fails to solve the level and never learns a useful policy, whereas DQN and NEAT navigate as quickly as possible to the food pellet. DQN was able to get to the food pellet in one of the early episodes where it performs many random moves and after that was able to capitalise on that learning outcome while NEAT successfully found a member in a species that got to the food. The fact that GA did not solve this level could be because it got a bad initialisation or the mutation strength is a bit too low to be ideal on this level where exploration is very important initially. This also highlights an interesting difference between GA and NEAT: NEAT contrary to GA incorporates mechanisms aimed at protecting species and diversity whereas GA in this case converged to a population of suboptimal solutions from which it could not make further progress.

For the level `smallSearch` on the other hand GA is able to consistently in all 25 validation trials find a solution that is on a par with the solution NEAT has found while DQN is unable to solve the level. The mean reward of DQN is the lowest possible, -10500 (-500 for running 500 timesteps and -10000 for timing out), which means it for some reason gets stuck right from the beginning and does not even pick up a single food pellet. This may be explained by DQN being unstable and sometimes not converging properly.

On `smallGrid` it seems all the algorithms roughly 50 % of the time complete the level and avoid the single ghost and the other 50 % of the time get caught by the ghost. The fact that this happens for all the algorithms suggests that perhaps the features available are not always enough to enable the agent to complete all levels perfectly the majority of the time. Looking at the layout of the level an agent may go for the food pellet in the middle but get trapped by the ghost with no way to escape. Since the feature pertinent to the active ghost is only different from 0.0 when the ghost is nearby the agent cannot possibly foresee that it will get trapped by going for the food in the middle. Moreover, the agent can not by the features tell the two food pellet positions apart and can only go by the distance. Nevertheless, the algorithms do learn to go for the food and avoid the ghost in the pathway around the edge; otherwise the mean reward would be substantially lower.

As noted before, NEAT does not find a good solution on `smallClassic`, but finds a solution where it does pick up some food pellets before getting caught by a ghost, which can be seen from the fact that the mean reward is higher than the penalty for getting caught by a ghost (-500). On this level GA does very well, which can be explained by GA finding a solution by chance and that elite solution then gives rise to multiple also well performing variations in the following generations.

On `powerClassic` all the algorithms achieve positive mean reward, but this is also something we might expect as there are many food pellets in the level that the agent will stumble upon giving it a high amount of protection while eating food. Judging by the high max scores for the algorithms, which is well above the number of points that can be achieved from the food and completion of the level alone, we can also see that the algorithms eat one or more of the ghosts during some of the validation trial runs.

Lastly, the `originalClassic` level which resembles the original pacman game, NEAT performs the best and has a significantly higher mean reward than DQN and GA. All the algorithms do well on this map though, which may be explained by the observation that there are fewer ghosts relative to the number of food pellets and size of the level compared to for instance `smallClassic`, which makes it less likely for the agent to get caught by getting trapped.

8 Discussion and Future Work

The results and observations made in section 7 confirm that a simple genetic algorithm like the one in [24] and the tried and true NEAT algorithm [21] can achieve just as good results and even better in some cases on the cartpole balancing problem and a simplified version of the pacman environment. For cartpole the hyper-parameter setting was based on settings from other sources, so this gives extra degree of confidence that the hyper-parameters are good for the problem and makes the benchmarks fair when compared. Looking in the literature we also see a similar story with NEAT being among the top algorithms in a study from 2008 [9] where multiple different algorithms are compared on Cartpole. The best algorithkm, however, is Cooperative Synapse Neuroevolution (CoSyNE), which is also a neuroevolution algorithm.

For the pacman environment we could have invested more time searching for optimal hyper-parameters for each algorithm, which is one idea for future work. For NEAT there are many hyper-parameters that could be tweaked and it could be interesting to study the effect of the different settings more closely. For GA there are relatively few hyper-parameters to tune, so the number of combinations to try is considerably lower. This highlights one of the advantages of the GA algorithm. For DQN we could study the trade-off between exploration and exploitation by varying the rate of the percent chance the algorithm will perform a random action during training.

Although the mean reward for GA was lower than DQN and NEAT on multiple of the levels, GA was performing very well considering its simplicity. The simplicity of the GA algorithm is another great benefit it has. It is very simple in both how it works and it is straight forward to implement. Testing more settings of the mutation strength could potentially make it perform even better. An interesting idea for future work would be to experiment with other genetic algorithms for training neural networks. Crossover could be added and other kinds of selection mechanisms like tournament selection could be used. The paper by Such et al. [24] is relatively recent and the experimentation of other genetic algorithms for training neural networks on difficult reinforcement learning domains such as Atari is a promising area of applied research. In the implementation of this project direct encoding of genomes to phenotype neural networks was used, which was sufficient for the problems considered. By instead using the indirect encoding scheme in [24] the algorithm can be scaled up easily and it would not be difficult to make each generation able to run in parallel on multiple CPUs/GPUs.

Another interesting field of research is to analyse the runtime of the (1+1) ES more rigorously on the two simple problems considered in this project in section 4.1. From there the complexity could also be increased to more advanced problems, neural networks and other evolutionary algorithms. It could also be very interesting to combine some ideas from the (1+1) ES and CMA-ES with the reinforcement learning part of the project by investigating how a self-adjusting evolutionary algorithm would perform when used for training neural networks.

Novelty search was considered and discussed throughout the project, but has not been implemented. This is due to the time it would require to both implement it for one or more of the algorithms and the time to make some good experiments that would highlights it effect. Instead time and effort was spent elsewhere. Nevertheless, novelty search would be a very interesting topic for future work as the the idea of maintaining diversity and refrain from directly searching for solutions is a brilliant idea and can be very effective in some cases. For instance diversity could potentially greatly help on a domain like the Atari game Montezuma's Revenge, which has for a long time been one of the most difficult Atari domains for many algorithms, because many algorithms fail at getting the initial reward of picking up the first key and using it. Concerning Montezuma's Revenge, recently Lehman, Stanley, the authors behind the original paper on novelty search together with Clune and Huizinga have published a paper where there agent scores 43,000 points, which is almost 4 times the previous state-of-the-art score on Montezuma's Revenge. [6].

Finally a topic idea for future work would be to benchmark the algorithms on more difficult domains or increase the complexity of the pacman environment by using a pixel-based version instead. This would amount to increasing the size of the networks that each algorithm optimises and training for a much longer period of time. It will also be extra important to choose the hyper-parameters carefully, which would again highlight the idea of further hyper-parameter search as part of the project.

9 Conclusion

In this project an overview of some of the main ideas and techniques in the field of neuroevolution has been given based on a survey of the key results of the literature as well as recent research. Furthermore, a simple genetic algorithm (GA) and the NEAT algorithm have been implemented and benchmarked on the cartpole balancing problem and pacman environment. Both algorithms maintain a population of genomes encoding neural networks, which are optimised using an evolutionary algorithm in contrast to classical approximative reinforcement learning techniques such as DQN. The results of the benchmarks show that both GA and NEAT are competitive with DQN on the two problems tested with NEAT achieving better scores than DQN. The results are in line with the results of recent research also showing great potential of evolutionary algorithms for training neural networks for reinforcement learning problems.

During this project an attempt has also been made to analyse the runtime of evolutionary algorithms including the (1+1) ES on a simple neural network for a simple classification problem. There seems to be a very limited amount of theoretically derived runtime bounds in the literature for evolutionary algorithm optimising neural networks, which is therefore a new potential frontier and area of research.

The survey of the literature on neuroevolution as well as the experience and results achieved by the work of this project has given multiple ideas and motivation for interesting future work. It will also be exciting to follow future research in neuroevolution in the literature in the decade to come.

References

[1] Jing An Abeynaya Gnanasekaran Jordi Feliu Faba. *Reinforcement Learning in Pacman.* Unknown year.

[2] Anne Auger. "Benchmarking the (1+ 1) evolution strategy with one-fifth success rule on the BBOB-2009 function testbed". In: *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers.* 2009, pp. 2447–2452.

[3] Anne Auger and Benjamin Doerr. *Theory of randomized search heuristics: Foundations and recent developments.* Vol. 1. World Scientific, 2011.

[4] Andrew G Barto, Richard S Sutton, and Charles W Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE transactions on systems, man, and cybernetics* 5 (1983), pp. 834–846.

[5] Markus Deserno. "How to generate equidistributed points on the surface of a sphere". In: *If Polymerforshung (Ed.)* (2004), p. 99.

[6] Adrien Ecoffet et al. "Go-explore: a new approach for hard-exploration problems". In: *arXiv preprint arXiv:1901.10995* (2019).

[7] Benjamin Eysenbach et al. "Diversity is all you need: Learning skills without a reward function". In: *arXiv preprint arXiv:1802.06070* (2018).

[8] Adam Gaier and David Ha. "Weight agnostic neural networks". In: *Advances in Neural Information Processing Systems.* 2019, pp. 5364–5378.

[9] Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. "Accelerated neural evolution through cooperatively coevolved synapses". In: *Journal of Machine Learning Research* 9.May (2008), pp. 937–965.

[10] Nikolaus Hansen. *pycma Python module.* Visited November 2020. URL: `https://pypi.org/project/cma/`.

[11] Nikolaus Hansen and Andreas Ostermeier. "Completely derandomized self-adaptation in evolution strategies". In: *Evolutionary computation* 9.2 (2001), pp. 159–195.

[12] Jens Jägersküpper. "Algorithmic analysis of a basic evolutionary algorithm for continuous optimization". In: *Theoretical Computer Science* 379.3 (2007), pp. 329–347.

[13] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more.* Packt Publishing Ltd, 2018.

[14] Joel Lehman and Kenneth O Stanley. "Novelty search and the problem with objectives". In: *Genetic programming theory and practice IX.* Springer, 2011, pp. 37–56.

[15] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[16] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[17] Jean-Baptiste Mouret and Jeff Clune. "Illuminating search spaces by mapping elites". In: *arXiv preprint arXiv:1504.04909* (2015).

[18] Tim Salimans et al. "Evolution strategies as a scalable alternative to reinforcement learning". In: *arXiv preprint arXiv:1703.03864* (2017).

[19] David Silver. *UCL Course on RL*. Visited September 2020. URL: `https://www.davidsilver.uk/teaching/`.

[20] 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), pp. 185–212.

[21] Kenneth O Stanley and Risto Miikkulainen. "Evolving neural networks through augmenting topologies". In: *Evolutionary computation* 10.2 (2002), pp. 99–127.

[22] Kenneth O Stanley et al. "Designing neural networks through neuroevolution". In: *Nature Machine Intelligence* 1.1 (2019), pp. 24–35.

[23] Christopher Stanton and Jeff Clune. "Curiosity search: producing generalists by encouraging individuals to continually explore and acquire skills throughout their lifetime". In: *PloS one* 11.9 (2016), e0162235.

[24] Felipe Petroski Such et al. "Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning". In: *arXiv preprint arXiv:1712.06567* (2017).

[25] Berkley University. *UC Berkeley CS188 Intro to AI – Course Materials*. Visited October 2020. URL: `http://ai.berkeley.edu/project_overview.html`.

[26] Unknown. *Equidistributed sequence*. URL: `https://en.wikipedia.org/wiki/Equidistributed_sequence`.

[27] Carsten Witt. "Tight bounds on the optimization time of a randomized search heuristic on linear functions". In: *Combinatorics, Probability & Computing* 22.2 (2013), pp. 294–318.

[28] Carsten Witt and Paul Fischer. *Runtime Analysis of Neuroevolution*. Sept. 2020.