Master Computer Science

Solving multi-agent reinforcements learning problems using Differential Evolution

Name:	Koen Bouwman
Student ID:	S1674307
Date:	[29/02/2024]
Specialisation:	Artificial Intelligence
1st supervisor:	Dr. A.V. Kononova
2nd supervisor:	Prof.dr. A. Plaat
3rd supervisor:	A. Wong MSc

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Acknowledgements

I would like to express my gratitude to Annie Wong Msc and Dr. Anna Kononova, for supervising this thesis, planning regular meetings, providing meaningful insights, and giving feedback. I am also grateful to Prof. Dr. Aske Plaat, for his feedback and his role as my second supervisor for this project. Furthermore, I would like to thank Diederick Vermetten MSc, for providing the modular differential evolution framework, and for guiding the use of this framework. Special thanks to Jacob de Nobel MSc for his help in optimizing the code and identifying bottlenecks in the run-time performance of the code.

Abstract

Multi-agent reinforcement learning (MARL) problems pose significant challenges to temporal difference (TD) based methods. Evolutionary algorithms (EAs) are optimization algorithms that can find global optima in multi-dimensional vector spaces. EAs have successfully solved single-agent RL problems before.

In this thesis, we apply EAs to MARL problems to see if they can deal with the challenges of MARL problems better than TD methods. To do this we will treat the weights of a neural network (NN) as a vector and use an EA to search the vector space for a global optimum. We will evaluate the performance of modular differential evolution (modDE) on the Switch2 and Checkers environments from ma-gym. We will use value decomposition networks (VDN) and a random search through the policy space as benchmarks to compare against modDE.

We found that that modDE can solve the Checkers and Switch2 environments. Matching the near-optimal score of VDN on these environments. However, it does turn out that dimensionality is a significant limiting factor when applying modDE to MARL problems. This means that modDE is well suited to solve MARL problems, provided that the dimensionality is small enough or that it can be reduced by other methods.

Contents

1 Introduction **1**

2 Background and related work **2**
2.1 Reinforcement Learning . . . 2
2.1.1 Markov Decision Process . . . 3
2.1.2 Partially Observable Markov Decision Process . . . 4
2.1.3 Value-based learning . . . 5
2.2 Deep Reinforcement Learning . . . 6
2.2.1 Perceptron . . . 6
2.2.2 Neural Networks . . . 8
2.2.3 Deep Q Networks . . . 10
2.3 Multi agent deep reinforcement learning . . . 11
2.3.1 Decentralized Partially Observable Markov Decision Process . . . 11
2.3.2 Challenges in Multi agent reinforcement learning . . . 12
2.3.3 Value Decomposition Networks . . . 15
2.4 Evolutionary Algorithms . . . 18
2.4.1 Representation, Phenotype, and Genotype . . . 20
2.4.2 Population and Diversity . . . 21
2.4.3 Mutation . . . 22
2.4.4 Recombination . . . 23
2.4.5 Selection . . . 25
2.4.6 Stopping criteria . . . 26
2.4.7 Advantages and disadvantages of EAs . . . 27
2.5 Differential Evolution . . . 27
2.5.1 Mutation . . . 27
2.5.2 Crossover . . . 28
2.5.3 Survivor Selection . . . 29
2.5.4 Parent Selection . . . 29
2.5.5 DE Algorithm . . . 30

3 Problem Statement **31**

4 Methodology **32**
4.1 Algorithms . . . 32
4.1.1 Diferential Evolution . . . 32
4.1.2 Value Decomposition Networks . . . 33
4.1.3 Random Search . . . 33
4.2 Environments . . . 34
4.2.1 ma-gym Checkers-v0 . . . 35
4.2.2 Compressed Checkers . . . 36
4.2.3 ma-gym Switch2-v0 . . . 38

5 Experimental Setup **39**
5.1 Plot choices . . . 39
5.2 Network Architectures . . . 39
5.2.1 Hidden Layers . . . 40

5.2.2 Bias . . . 43
5.3 Hyper parameter optimization . . . 43
5.3.1 Boundaries . . . 44
5.3.2 Hyperparameters for modDE . . . 45

6 Results **46**
6.1 Checkers . . . 46
6.2 Compressed Checkers . . . 48
6.3 Switch2 . . . 49

7 Discussion **50**
7.1 Overal performance . . . 50
7.1.1 Switch2 . . . 50
7.1.2 Checkers and Compressed Checkers . . . 50
7.2 Dimensionality . . . 51
7.3 Local Optima . . . 53

8 Conclusions and Future Research **53**

A modDE hyperparameters **58**

B Run time and dimensionality **59**

C Elitism for VDN **61**

D Code **63**

1 Introduction

Reinforcement learning (RL) is a branch of machine learning in which there are no ground truth answers. Instead, an agent must learn by trial and error from interacting with its environment. The agent receives feedback from the environment in the form of rewards and punishments. The goal of an RL agent is to maximize its long-term return. Neural networks (NNs) have been successfully used as policies for RL agents, for example, the Deep Q Networks (DQN) by Mnih et al. [28].

Multi-agent reinforcement learning (MARL) is, as the name suggests, a branch of RL with several agents that are learning at the same time. As pointed out by Wong et al. [41], introducing more agents into RL settings comes with the following challenges:

- Non-stationarity: If one agent changes its policy, the environment changes for all other agents.
- Credit assignment: To what extend did an agents own actions contribute to a group reward or penalty?
- Partial observability: An agent may not have access to all information about the state of an environment.
- Computational complexity: Modeling several agents and updating several policies can increase the computational complexity of MARL problems with respect to RL problems.

As stated by Eiben and Smith [8], evolutionary algorithms (EAs) are a class of optimization problems that take inspiration from natural evolution. Their goal is to find global optima, or near-optimal solutions to problems in multi-dimensional vector spaces. Moriarty et al. [12] suggest that EAs are well suited to deal with the challenges of MARL. In particular, it proposes that EAs can deal with non-stationarity and credit assignment more effectively.

In this thesis, we ask the following question: **Is differential evolution (DE) better suited to deal with the challenges of MARL problems than gradient-based RL methods?** In particular, is DE better equipped to handle the four challenges of non-stationarity (1), credit assignment (2), partial observability (3), and computational complexity (4)? To test this hypothesis, we will evaluate the performance of modular differential evolution (modDE) by Vermetten et al. [39], using the following environments as benchmarks:

- ma-gym: Switch2, by Koul [20].
- ma-gym: Checkers, by Koul [20].
- Compressed Checkers: a simplified version of ma-gym: Checkers, by Koul [20].

We will use value decomposition networks (VDN) by Sunehag et al. [37], and a random search RS through the policy space as baselines to compare the performance of modDE against.

The structure of this thesis is as follows: In Chapter 2, we discuss the background and preliminaries required to understand this thesis. In Chapter 3, we discuss the formal problem statement and hypothesis of this thesis. In Chapter 4, we discuss how we intend to verify this hypothesis and which algorithms and environments we use to evaluate our hypothesis. In Chapter 5, we perform some preliminary experiments to decide which network architectures and hyperparameters we want to use in our final experiments. In

Chapter 6, we show the results of our experiments and give a brief interpretation of the outcome of our experiments. In Chapter 7, we analyze our experiments and discuss the strengths and weaknesses of our approach. Finally, in Chapter 8, we conclude this thesis by evaluating our hypothesis and speculating on potential future research based on this thesis.

2 Background and related work

In this chapter, we will cover the background knowledge and preliminaries that are required to understand the remainder of this thesis. We will start by exploring single-agent reinforcement learning (RL) in Section 2.1. Then we will cover the use of neural networks (NNs) in RL in Section 2.2. We will extend the RL problems to a multi-agent reinforcement learning (MARL) setting in Section 2.3. In Section 2.4, we will discuss the theory behind evolutionary algorithms (EAs). We will go into greater depth into one class of EAs called differential evolution (DE) in Section 2.5.

2.1 Reinforcement Learning

Reinforcement learning (RL) is a machine learning technique in which an *agent* learns to complete an objective by interacting with its *environment*. The book by Sutton and Barto [38] provides an in-depth explanation of RL and frameworks such as the Markov Decision Process. The text and equations in this section are based on their book. The agent takes *actions* a from the action-set $\mathcal{A}$. The environment is everything that is beyond the agent's direct control. The agent can, however, interact with its environment through its own actions. An environment can be partially or fully observable to the agent. In the case of a fully observable environment, the agent knows everything about the *state* s of the environment. However, in the case of a partially observable environment, the agent only knows a subset of information about the environment through the *observations* $o \subseteq s$ it makes.

At every *timestep* t the agent receives a numerical *reward* $r_t \in \mathbb{R}$ from the environment, based on the state s_t of the environment and the action a_t performed by the agent in the given state. This reward can be positive, negative, or zero. The rewards are key in reinforcement learning, as the objective of any reinforcement learning agent is to maximize the cumulative rewards that it gets. The rewards are the only feedback the agent receives; it will have to change its behavior through trial and error, basing its success solely on the rewards it receives. Figure 1 shows one iteration of such an RL problem.

Let $\gamma \in [0, 1]$ be the *discount factor*, then we can define a (discounted) *return* G_t at timestep t as the discounted sum of all future rewards r_{t+k+1} for $k \in \mathbb{N}_0$.

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1} = r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \dots \tag{1}$$

The discount factor γ can be used to tune the priority of the agent with respect to immediate rewards versus future rewards. A γ closer to 0 will put more emphasis on immediate rewards, whereas a γ closer to 1 will put more emphasis on future rewards.
The *policy* π determines which action the agent takes in a given timestep. In the simplest case, the policy is a function that maps the state s of the environment to the desired action a to take for that agent: $\pi : s \in \mathcal{S} \rightarrow a \in \mathcal{A} : \pi(s) = a$. However, in the most general case, the policy can depend on all past states, actions, and rewards, and it may even be stochastic in nature:

$$\pi(a|s_0, \dots, s_t, r_0, \dots, r_t) \doteq P\left((a_t = a|s_0 = s_0, \dots, s_t = s_t, r_0 = r_0, \dots, r_t = r_t)\right) \tag{2}$$

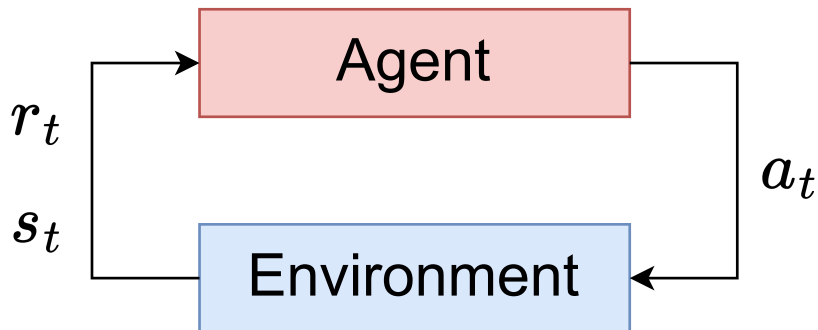

Figure 1: This figure shows one iteration of the RL loop. The agent receives a state s_t and reward r_t from the environment. Based on the these inputs and its policy π it will select an action a_t. This action influences the environment environment to transition from the current state s_t to the next state s_{t+1}. It returns this next state and the corresponding reward r_t back to the agent. This loop continues until the training of the agent is over.

Equation 2 is the general definition of a policy function π, defining the probability of taking action a after having observed states $s_0, \ldots s_t$ and received rewards $r_0, \ldots, r_t$. As the training process will attempt to maximize the return an agent receives by changing the policy over time. A policy that maximizes the return given to an agent is considered an optimal policy π^*.

The *transition function* $\mathcal{T}$ of an environment is what determines what the reward and next state of the environment will be. More formally, if there have been t timesteps so far: the transition function maps the past states $\{s_0, \ldots, s_t\}$, rewards $\{r_0, \ldots, r_t\}$, and actions $\{a_0, \ldots, a_t\}$ to the next state s_{t+1}. It can be defined as follows in Equation 3

$$\mathcal{T} = P\{s_{t+1} = s' | s_0, a_0, r_0, \ldots, s_t, a_t, r_t\} \tag{3}$$

2.1.1 Markov Decision Process

Most reinforcement learning problems can be formulated as a Markov Decision Process (MDP), as described by Sutton and Barto [38]. The MDP is a model with discrete timesteps t. Formally an MDP is a 4-tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \rangle$

- $\mathcal{S}$ is the finite *state space* of all possible states in the environment.
- $\mathcal{A}$ is the finite *action space* of all possible actions that can be taken by the agent.
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1] \subseteq \mathbb{R}$ is the *transition function*. It maps the combination of the current state s_t, the action a_t, and the next state s_{t+1} to the probability that the action a_t in state s_t leads to the next state s_{t+1}; formally $P(s_{t+1} | s_t, a_t)$.
- $\mathcal{R} : \mathcal{S} \times \mathcal{A}$ is the *reward function*. It maps the current state s_t and action a_t to a numerical reward. Roughly speaking, high rewards indicate that action a_t in state s_t was a good action and low rewards indicate that action a_t in state s_t was a bad action.

The key defining feature of an MDP is the Markov property. In simple terms, the Markov property means that the current state of the environment is sufficient to determine the probability distribution of the next state. Therefore, the transition function is independent of all previous states and actions and it is

solely a function of the current state S_t and action A_t. More formally, if there have been t timesteps so far, the transition function can be specified as the probability distribution defined in Equation 3 However, when the Markov property holds this probability is independent of all rewards and all past states (S_k with $k < t$), given the following distribution:

$$\mathcal{T} \doteq p(s'|s, a) \doteq P\{s_{t+1} = s'|s_t = s, a_t = a\} \tag{4}$$

The Markov property holds if and only if Equation 3 is equal to Equation 4, for all potential rewards r, potential future states s', current states s, and current actions a.

If this Markov property holds for a given problem, then we consider that problem to be an MDP. For any MDP we can also simplify the general policy π from Equation 2, since the past states and rewards are no longer necessary[1]. This simplified π is defined in Equation 5

$$\pi(a|s) \doteq P\{(a_t = a|s_t = s)\} \tag{5}$$

For example, the game of Mario could be considered to be an MDP. In this case, the agent would control the character Mario, the environment would be the level, including all enemies and hazards, and the reward could be based on the in-game score.

2.1.2 Partially Observable Markov Decision Process

The Partially Observable Markov Decision Process (POMDP) is an extension of the MDP. It models the fact that an agent does not have access to all information about the state of the environment. For example, an agent living in a grid world might only be able to see tiles that it is adjacent to, whilst the total environment is of course larger than the agents' immediate surroundings.

Formally the POMDP is defined as a 6-tuple: $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{O}, \Omega \rangle$. In this tuple, $\mathcal{S}, \mathcal{A}, \mathcal{T}$, and $\mathcal{R}$ function exactly the same for the POMDP as for the MDP. The two new symbols Ω and $\mathcal{O}$ both handle the observations. $\mathcal{O}$ is the finite observation space of all possible observations the agent can encounter. $\Omega : \mathcal{S} \times \mathcal{A} \times \mathcal{O} \rightarrow [0, 1] \subseteq \mathbb{R}$ is the observation probability function. It takes the current state s_t, the selected action a_t and a possible future state s_{t+1} and computes the probability that the action a_t in state s_t leads to the future state s_{t+1}.

For a problem to be considered to be a POMDP, it should still be a stationary problem, and the Markov property should still hold. Therefore, transition function $\mathcal{T}$ should still be the same as it is in Equation 4 for both the MDP and POMDP. However, the agents no longer observe the full state s_t environment. Instead they only observe a part of the environment; their observation o_t. Therefore the policy π must now be dependant on the observation, rather than the state s, as is depicted in Equation 6

$$\pi(a|o) \doteq P\{(a_t = a|o_t = o)\} \tag{6}$$

A visual representation of one iteration of the POMDP is shown in Figure 2. Note that the figure similar to Figure 1, visualizing reinforcement learning in the MDP framework. The only distinction between the two figures is that Figure 2 has an extra step. Namely, the observation function Ω.

[1]However, it can be useful in some ML paradigms to still keep some dependency on past states or rewards, so this step is not strictly necessary

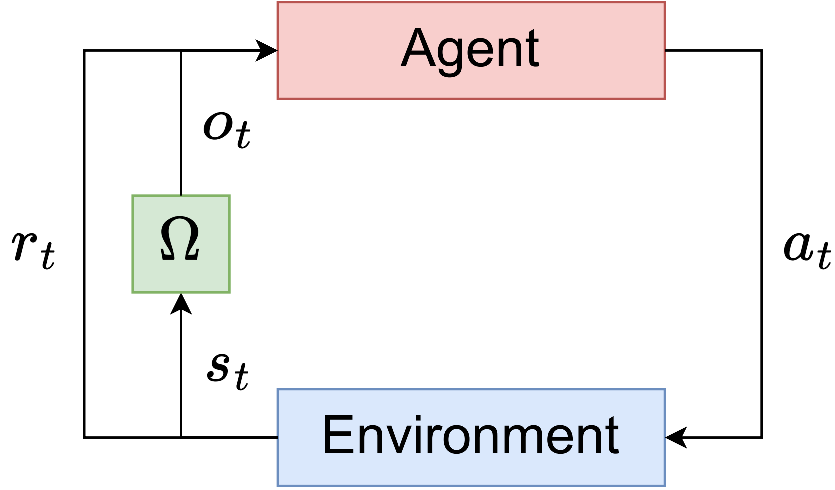

Figure 2: This figure shows how the agent interacts with the environment in the POMDP framework. At each timestep t the environment outputs the state s_t and reward r_t. The stochastic observation function creates the observation o_t. The agent then selects an action a_t which it executes, causing the environment to transition to the next state s_{t+1}, completing one loop iteration.

2.1.3 Value-based learning

There are three main branches in RL: value-based, policy-based, and model-based RL. Model-based and policy-based RL are out of scope for this thesis. In model-based learning, one algorithm attempts to create a simplified model of the environment so that another algorithm can then plan a strategy using the model, rather than the more complicated environment. Moerland et al. [29] goes into great depth on the topic of model-based RL. In policy-based learning, an agent will update its policy based directly on its interactions with the environment, without first computing a simpler model or value function.

Value-based learning is a branch of reinforcement learning that aims to learn the value of a state s. This value represents the total discounted reward that the agent is expected to receive if it continues playing by its policy π. More formally: given state s at timestep t, the agent's (current) policy π, and the return G_t defined in Equation 1, the value function $v_\pi(s)$ is defined in Equation 7. In which:

$$v_\pi(s_t) = \mathbb{E}_\pi[G_t|s_t] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^{t+k} \cdot r_{k+t+1}|s_t\right] \tag{7}$$

Given the value function based on the state s_t and knowledge about the action space $\mathcal{A}$, we can further define the Q-value $Q_\pi(s_t, a_t)$ for each action a_t taken in state s_t by modifying the value function to include the constraint that action a_t is played at timestep t. More formally: given state s_t and action a_t at timestep t, the agent's (current) policy π, and the return G_t defined in Equation 1, the Q-value $Q_\pi(s_t, a_t)$ is defined in Equation 8.

$$Q_\pi(s_t, a_t) = \mathbb{E}_\pi[G_t|s_t, a_t] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^{t+k} \cdot r_{k+t+1}|s_t = s_t, a_t = a_t\right] \tag{8}$$

If the Markov property holds, we can even simplify the computation of Q-values by defining Q-values recursively, this results in Equation 9, famously known as the Bellman equation [1].

$$Q_\pi(s_t, a_t) = r_t + \gamma \cdot Q_\pi(s_{t+1}, a_{t+1}) \mid a_{t+1} = \pi(s_{t+1})$$
$$Q_\pi(s_t, a_t) = r_t + \gamma \cdot Q_\pi(s_{t+1}, \pi(s_{t+1})) \tag{9}$$

When the environment is simple enough it is possible to directly learn all Q-values and store them in a Q-table. A Q-table contains the states as columns and actions as rows or vice versa, the cells of the Q-table represent the Q-value of the state-action-pair corresponding to the column and row of that cell. For more complex environments there are too many states and actions to store the Q-values of all state-action pairs. In this case, it is necessary to approximate the Q-function. The most common models to approximate Q-functions are neural networks, more on them in Section 2.2.

When training Q-values directly we can use temporal difference learning as described by Equation 10. In this equation, the Q-value $Q_\pi(s_t, a_t)$ for policy π, state s_t, and action a_t at timestep t is updated by adding α times the temporal difference (denoted between the square brackets) to the old value. The learning rate α is a hyperparameter in the range $[0, 1]$ that determines how quickly the agent learns. A higher learning rate increases the speed of convergence, but it may lead to less stability as the agent might jump over an optimal solution. The temporal difference itself is based on the reward r_t and the difference between the old Q-value and the old value of the maximum Q-value in the next state s_{t+1} multiplied by the discount factor γ.

$$Q_{\texttt{new}}(s_t, a_t) = Q_{\texttt{old}}(s_t, a_t) + \alpha \cdot \left[r_t + \gamma \cdot \max_{a'} \left(Q_{\texttt{old}}(s_{t+1}, a') \right) - Q_{\texttt{old}}(s_t, a_t) \right] \tag{10}$$

2.2 Deep Reinforcement Learning

Deep reinforcement learning is a field of reinforcement learning in which the policy of the agent(s) is defined by a neural network (NN) see Section 2.2.2. In this section, we {I, since you are writing this alone will explore how NNs work and how they can be used as a policy for a value-based RL agent.

2.2.1 Perceptron

The perceptron, an early prototype/building block of a neural network, was first developed in 1943 by McCulloch and Pitss [26]. It was developed to be a classifier that could distinguish between two different classes. A perceptron can take p different inputs $\{q^1, \dots, q^p\}$, it will then take the weighted sum of these inputs and the bias $q^0 \doteq 1$, based on the weights $w^0, \dots, w_p$. The activation function φ is then applied to this weighted sum. This results in the output o, as defined by Equation 11.

$$o = \varphi \left(\sum_{i=0}^{p} w^i \cdot q^i \right) \tag{11}$$

Figure 3 shows a schematic overview of the perceptron (denoted in red), it consists of the weighted sum in blue and the activation function in yellow. This image was inspired by Mitchell [27].

There are several different activation functions φ. Examples include the step, tanh, sigmoid, relu, and softmax functions. In the paper by Ittiyavirah et al. [18], the authors analyze the effect of different activation functions on a model that learns to approximate the cosine function. The shapes of these functions might vary significantly, but their primary purpose is often the same: to introduce non-linearity into the computation. This allows the perceptron to capture more complex relationships. Another possible

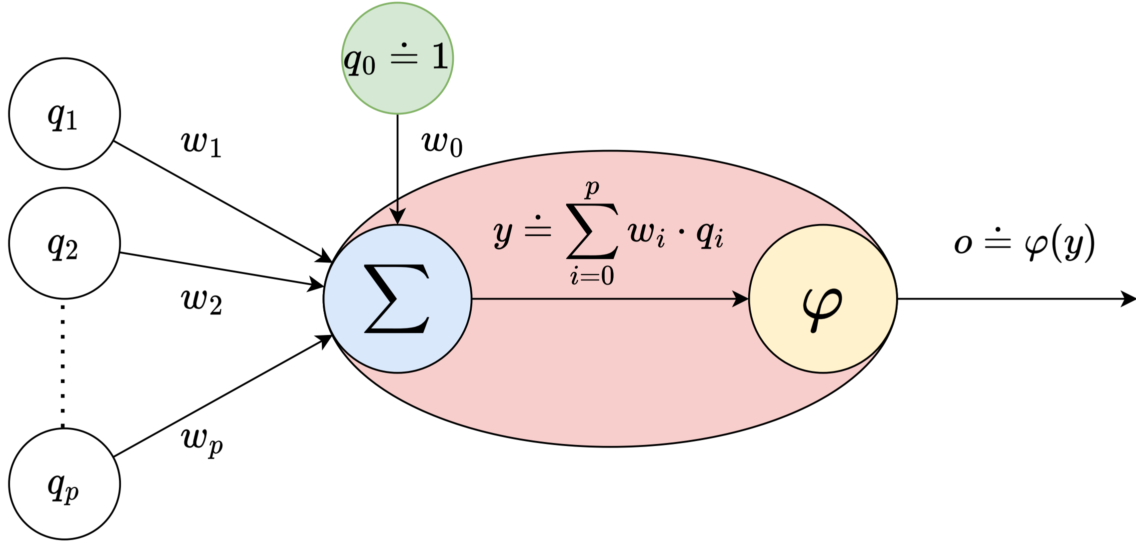

Figure 3: An overview of the perceptron, denoted in red. On the left we can see the inputs $q^1, \ldots, q^p$, including the special input $q^0 = 1$; the bias, in green. The perceptron can be trained by tuning the weights $w^0, \ldots, w^p$. Internally, the perceptron first takes the weighted sum of all the inputs and the bias, this happens in the blue node. After that, the activation function φ is applied to the weighted sum y, this happens in the yellow node. The final output is represented in Equation 11

purpose of the activation function is to squash the output of the perceptron into a specific range. For example, the step function allows the perceptron to be used as a binary classifier.
In order to train a perceptron we must first define a loss function $\mathcal{L}$. This loss function should represent a difference between the desired output O_* and the actual output of the perceptron O_p, i.e. if O_p and O_* are very similar, the loss would be small, whereas if O_p and O_* are very different the loss would be large. Common loss functions include the absolute difference: $\mathcal{L} = |O_p - O_*|$ and the mean squared error: $\mathcal{L} = (O_p - O_*)^2$. The former is more common in single perceptrons, whereas the latter is more common in neural networks or multi-layer perceptrons.

When training the perceptron, the goal is to minimize the loss. To do this, the gradient $\vec{\nabla}_{\mathcal{L}}$ of the loss function with respect to the weights needs to be computed. $\vec{\nabla}_{\mathcal{L}}$ is defined in Equation 12, as are the weights vector $\vec{w}$ and the input vector $\vec{q}$. Writing out the vectors in this fashion also allows us to simplify the output of the perceptron to be the activation function applied to the inner product of $\vec{w}$ and $\vec{q}$: $o = \varphi(\vec{w} \cdot \vec{q})$

$$\vec{\nabla}_{\mathcal{L}} \doteq \left(\frac{\partial \mathcal{L}}{\partial w^0}, \ldots, \frac{\partial \mathcal{L}}{\partial w^p}\right), \ \vec{w} \doteq \left(w^0, \ldots, w^p\right), \ \vec{q} \doteq \begin{pmatrix} q^0 \\ \vdots \\ q^p \end{pmatrix} \tag{12}$$

Since the computation for all w^j are analogous, we will only elaborate one derivation of $\frac{\partial \mathcal{L}}{\partial w^j}$. The loss function is a function of O_p and O_*: $\mathcal{L}(O_p, O_*)$. For this derivation, we can treat O_* as a constant since it is not dependant on $\vec{w}$, and thus not dependant on w^j in particular. Thus by the chain rule we find that $\frac{\partial \mathcal{L}}{\partial w^j} = \frac{\partial \mathcal{L}}{\partial O_p} \cdot \frac{\partial O_p}{\partial w^j}$. Knowing that O_p is simply the activation function φ applied to the weighted sum of inputs y we can rewrite this equation to $\frac{\partial \mathcal{L}}{\partial w^j} = \frac{\partial \mathcal{L}}{\partial \varphi} \cdot \frac{\partial \varphi}{\partial w^j}$. By examining Equation 11 and Figure 3, and by using the chain rule we can find that $\frac{\partial \varphi}{\partial w^j} = \frac{\partial \varphi}{\partial y} \cdot \frac{\partial y}{\partial w^j}$. For the perceptron, we can trivially compute $\frac{\partial y}{\partial w^j}$ to be q^j, since the other terms of the summation of y are not dependant on w^j. All computations

result in the final expression for $\frac{\partial \mathcal{L}}{\partial w_j}$ in Equation 13:

$$\frac{\partial \mathcal{L}}{\partial w^j} = \frac{\partial \mathcal{L}}{\partial \varphi} \cdot \frac{\partial \varphi}{\partial y} \cdot q^j \tag{13}$$

Once the full gradient vector $\vec{\nabla}_{\mathcal{L}}$ has been computed we can perform the gradient descent step to update the weights. This step is similar to the temporal difference step described in Equation 10 in the sense that there is a learning rate α which can control the size of the update step. The rule to update the new weights of the perceptron $\vec{w}'$ from the old weights $\vec{w}$ is described in Equation 14

$$\vec{w}' = \vec{w} - \alpha \cdot \vec{\nabla}_{\mathcal{L}} \tag{14}$$

2.2.2 Neural Networks

The concept of the perceptron can be extended by feeding the output of one (set of) perceptron(s) directly to the input of another (set of) perceptron(s), creating a (neural) network of interconnected perceptrons. From here onward we will refer to the perceptrons in the network as neurons and to their respective outputs as their activation. This can be layered: the output of the first layer of neurons serves as the input for each neuron in the second layer, whose activations serve as the input of the third layer, and so on. This system is called a multi-layer perceptron, more commonly known as the (artificial) neural network (NN); first introduced by Gardner and Dorling [10]. Figure 4 shows a schematic overview of a small scale neural network.

Unlike perceptrons, NNs can have more than one output. Therefore, they can be used to classify objects into multiple classes, rather than the binary classification that perceptrons allow. Most commonly the classes are one-hot encoded to the outputs of the NN. Thus an NN with n outputs can be used to classify objects into n classes. This means that the network in Figure 4 is likely used to classify objects into four classes. NNs can also be used for reinforcement learning instead of classification. In this case, discrete actions can be one-hot encoded to the outputs, analogously to the classification case, and continuous actions can take the network outputs directly.

When considering multiple neurons (perceptrons), the activation a_l^j of a single neuron n_l^j is based on the inner product of its inputs $\vec{a}_{l-1}$ and the weights $\vec{w}_l^j$ connected to it. Therefore, it makes sense to layer the individual weights-vectors in layer l $\vec{w}_l^j$ as rows of a weights-matrix W_l.
Then, if we multiply the input for that layer as a column vector to the left, and apply the activation function φ_l for that layer, we will get the activation of each of the neurons for that layer in the vector $\vec{a}_l$. This computation is highlighted in Equation 15. Note that the activation function used in each layer does not necessarily need to be the same.

$$\vec{a}_l = \varphi_l \left(W_l \cdot \vec{a}_{l-1} \right) \tag{15}$$

Since the activation for each layer is defined analogously and since it is recursively dependent on the activation of the previous layer, we can write out the final output vector $\vec{o}$ of the three-layered NN in Figure 4 in Equation 16. In this equation, the subscripts 1, 2, and O represent the first hidden layer, second hidden layer, and output layer, respectively.

$$\vec{O} = \varphi_O \left(W_O \cdot \varphi_2 \left(W_2 \cdot \varphi_1 \left(W_1 \cdot \vec{q} \right) \right) \right) \tag{16}$$

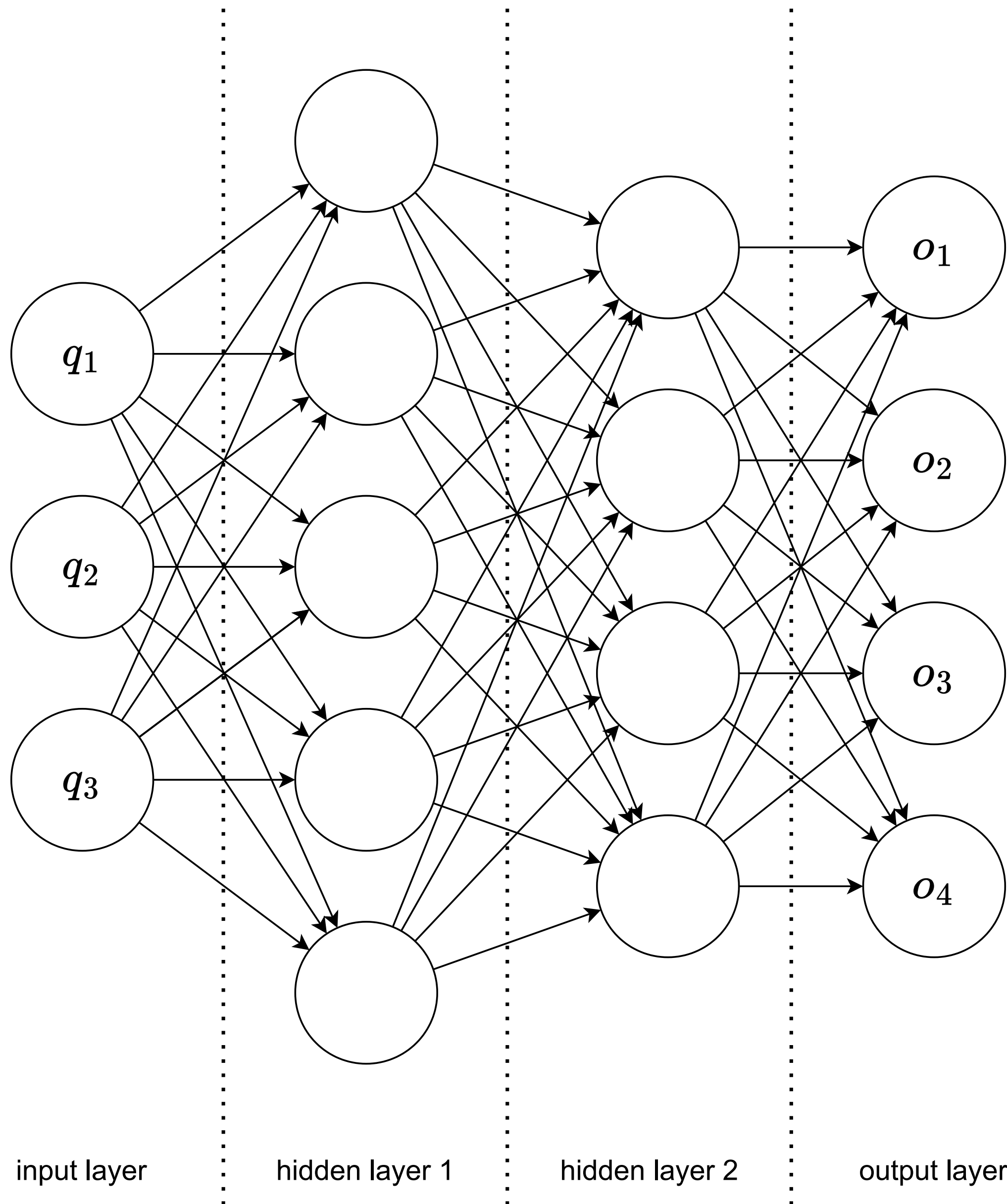

Figure 4: A schematic overview of a neural network (or multi-layer perceptron). This network has three inputs and four output nodes. It also has two hidden layers: the first hidden layer has five nodes (perceptrons) and the second hidden layer has four nodes.

This example can trivially be extended to several more layers of the NN, but this example with two hidden layers should illustrate the point. Computing the output of the network by passing the input vector through the layers from the input layer to the output layer is called the forward pass, or forward propagation because the numbers are passed from the front of the network to the back.

Defining a loss for an NN with multiple outputs involves summing the component-wise loss of each entry of the desired output $\vec{O}_*$ and the actual output $\vec{O}$: $\mathcal{L}(\vec{O}, \vec{O}_*) \doteq \sum_j \mathcal{L}(O^j_*, O^j)$.
The full computation of the gradient matrices $\nabla^l_\mathcal{L}$ for each layer l, or the individual gradients $\frac{\partial \mathcal{L}}{\partial w^j_l}$ is out of the scope for this thesis. However, Goodfellow et al. [11] provides an in-depth explanation of the math

behind backpropagation. The computation of $\nabla_{\mathcal{L},l}$ is called the backward pass or backpropagation. This is because the gradients with respect to the weights in the earlier layers are dependent on the gradient of weights in the later layers, meaning that these have to be computed first, hence the computation starts at the back of the network.

Once the gradient matrices $\nabla_{\mathcal{L},l}$ has been computed, the training of the NN is analogous to that of the perceptron: we simply scale $\nabla_{\mathcal{L},l}$ by the learning rate α and subtract it from the weights matrix W_l to update the weights using gradient descent, as is depicted in Equation 17

$$W_l' = W_l - \alpha \cdot \nabla_{\mathcal{L},l} \tag{17}$$

There are two properties of NNs that would make them capable of solving RL problems, these are:

- The combination of several layers and non-linear activation functions allows NNs to capture complex non-linear relationships between several inputs and their desired outputs. Or in RL terms: they allow the relationship between complex input states/observations and the best action to be captured.
- NNs scale well with the dimensionality of problems. This means that complex RL tasks that can only be modeled by high-dimensional environments can potentially also be solved by NNs.

Section 2.2.3 goes into more detail about how NNs can be adapted to solve RL problems.

2.2.3 Deep Q Networks

For very high dimensional reinforcement learning problems, learning the Q-value of all possible state-action pairs and storing them in a table becomes infeasible. Under such circumstances, employing a neural network to approximate the Q-function can be an effective alternative. The Deep Q Network (DQN), first introduced by Mnih et al. [28], is one of the prominent examples.

The observation of the agent is taken as the input for DQN. A DQN has one output for every action the agent could take. The value of the output represents the Q-value of the corresponding action, given the observation that was used as the input.

The first DQNs were trained on Atari games, taking the raw pixel data as inputs. Hence, the first layers of a DQN are convolutional layers to effectively recognize patterns in image data. These convolutional layers are followed by fully connected layers which are used to compute the Q-values of each action in the observation.

DQNs make use of experience replay, introduced by Lin et al. [24], which involves storing past experiences $e_t \doteq (s_t, a_t, r_t, s_{t+1})$, based on the reward r_t and the state transition $s_t \rightarrow s_{t+1}$ after playing action a_t at timestep t.[2]. During training the agent saves its current experience e_t in the replay buffer. If the replay buffer is full, the oldest experience will be removed to make room for the newest. It then samples a random batch of experiences from the replay buffer. This batch, rather than the experience e_t will be used to update the weights of the network. The key advantage of experience replay is that it breaks the correlation between successive actions. This allows for faster conversion because the input for the DQN is now more independently and identically distributed.

[2] or $e_t \doteq (o_t, a_t, r_t, o_{t+1})$, with observations $o_t \rightarrow o_{t+1}$ in case of a partially observable environment.

When updating the weights θ for a DQN based on the output from a network with these same weights θ, we encounter the issue of instability. This means that updating the weights to move to a given target output will also change that target output. Effectively causing the agent to chase a moving target. This problem can be alleviated somewhat by introducing a target network θ', as explained by Granger et al. [9]. The target network, rather than the Q-network, will be used to sample the Q-values of the actions in the next state. The original Q-network will be trained normally, whilst the target network will only be updated every k^{th} timestep. This will keep the target stationary for a few timesteps so that the Q-network can reach the target before it moves. The loss $\mathcal{L}$ is defined as the mean-squared-error-loss between the target y_t and the output of the Q network $Q(s_t, a_t|\theta_t)$, as defined in Equation 18.

$$\mathcal{L}_t(\theta_t) \doteq \mathbb{E}_{s_t,a_t}\left[(y_t - Q(s_t, a_t|\theta_t))^2\right] \tag{18}$$

In which t is the current timestep, o_t, a_t, and θ_t are respectively the state, action, and weights of the Q-network at timestep t. The target y_t is defined as: $y_t \doteq \mathbb{E}_{s_{t+1}}\left[r_t + \gamma \cdot \mathtt{max}_{a_{t+1}}(Q(s_{t+1}, a_{t+1}|\theta'_t))\right]$, in which γ is the discount factor, r_t is the reward at timestep t and θ'_t are the weights of the target network at timestep t.

In RL and other search algorithms, there needs to be a balance between exploration; discovering new parts of the search space, and exploitation; searching in areas known to be promising. To balance between exploration and exploitation DQN uses the ϵ-greedy search strategy. This means that at each timestep t, there is a probability ϵ that the agent selects a completely random action (exploration) and a $1 - \epsilon$ probability of selecting the best action known so far: $\mathtt{Max}_{a_t^i}(Q^i(o_t^i, a_t^i))$ (exploitation). Typically, the value of ϵ decreases over time, in most ϵ-greedy algorithms. This means that the earlier stages of training have more exploration, whilst the later stages have more exploitation.

There are many possible advancements to the DQN implementation by Mnih et al. [28], such as the Double DQN introduced by van Hasselt [14] and the dueling layers introduced by Wang et al. [40]. In their paper, Hessel et al. [15] compare all advancements known at the time, and they introduce the rainbow method which combines as many of them as possible into a single method.

2.3 Multi agent deep reinforcement learning

Some problems require more than one learning agent to interact with the same environment. These are the so-called multi-agent reinforcement learning (MARL) problems. As all agents learn from the environment, they will inevitably influence each other's training process. The adapting of one agent effectively changes the environment for all other agents, introducing non-stationarity into the environment. Moreover, because the agents learn from each other, and a changing/adapting/learning agent changes the environment for all other agents, we can no longer assume that the past actions of an agent do not influence the transition function for that agent. This breaks the Markov property, which means that MDP can no longer be used to model MARL problems. These kinds of MARL problems can be described by the Decentralized Partially Observable Markov Decision Process (DecPOMDP). The mathematical description of DecPOMDPs used in this thesis is based on the literature by Oliehoek et al. [32] and Wong et al. [41].

2.3.1 Decentralized Partially Observable Markov Decision Process

In this section, we will be indexing over agents $i \in I$ and timesteps $t \in \mathbb{N}$. We will denote the timesteps in the subscript and the agent in the superscript, such that a_t^i is the action taken by agent i at timestep

t. The DecPOMDP is an extension of the (partially observable) MDP described in Section 2.1.1. Like the (PO)MDP the DecPOMDP is a model with discrete timesteps and finite spaces for actions, states, and observations. The number of agents is also finite, with N denoting the total number of agents. Formally it is defined as the tuple: $\langle I, \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \mathcal{O}, \Omega \rangle$ in which:

- $I \doteq [1, N] \subseteq \mathbb{N}$ is the set of all agents N agents.
- $\mathcal{S}$ is the finite state space of the environment.
- $\mathcal{A} \doteq \times^i \mathcal{A}^i$ is the joint action space of all action spaces $\mathcal{A}^i$ for agent $i \in I$.
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1] \subseteq \mathbb{R}$ is the transition probability function.
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function.
- $\mathcal{O} \doteq \times^i \mathcal{O}^i$ is the joint observation space of all agents i.
- $\Omega : \mathcal{S} \times \mathcal{A} \times \mathcal{O} \to [0, 1] \subseteq \mathbb{R}$ is the observation probability function.

Any action $\vec{a}$ in the joint action space $\mathcal{A}$ is an element of the Cartesian product $\times^i \mathcal{A}^i$ of the individual action spaces $\mathcal{A}^i$ for all agents $i \in I$. This means that $\vec{a}$ is of the form $(a^1, \ldots, a^N)$, with $a^1, \ldots, a^N \in \mathcal{A}^1, \ldots, \mathcal{A}^N$. respectively. The observations are defined analogously: $\vec{o} \in \mathcal{O} \doteq \times^i \mathcal{O}^i$, with $\vec{o}$ of the form $(o^1, \ldots, o^N)$, with $o^1, \ldots, o^N \in \mathcal{O}^1, \ldots, \mathcal{O}_N$. At a given timestep t, $\vec{a}_t$ and $\vec{o}_t$ are respectively, the joined action and observation of all agents at timestep t. The individual action and observation of agent i at timestep t are denoted as a_t^i and o_t^i respectively.

The observation probability function Ω maps a combination of a state $s_t \in \mathcal{S}$, an action $\vec{a}_t \in \mathcal{A}$, and an observation $\vec{o}_t \in \mathcal{O}$, to the probability $P(\vec{o}_t | s_t, \vec{a}_t) \in [0, 1] \subseteq \mathbb{R}$ that the combination of the state s_t and action-vector $\vec{a}_t$ would result in the observed vector $\vec{o}_t$. Similarly to the POMDP, this function ensures that each agent does not have access to all the information about the environment. Rather each agent has its own (local) observation of the environment. The state space $\mathcal{S}$ and transition function $\mathcal{T}$ also behave analogously to their (PO)MDP counterparts. $\mathcal{T}$ maps the combination of the current state s_t, the current action vector $\vec{a}_t$, and the next state s_{t+1} to the probability $P(s_{t+1} | s_t, \vec{a}_t)$ that state s_{t+1} would arise from state s_t given that the agents have performed action a_t.
The reward function $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ computes the reward for all agents given the current state $s_t \in \mathcal{S}$ and action vector $\vec{a}_t \in \mathcal{A}$. This reward is the same for all agents because the DecPOMDP models a cooperative environment in which all agents have the same objective.

Figure 5 shows one iteration of the DecPOMDP model at timestep t. From this figure, we can see that the agents interact with the environment by choosing their own action a_t^i based on their own observation o_t^i and the collective reward r_t. By concatenating the observations and actions into vectors we can treat the collection of all agents as we would the agent in the POMDP.

2.3.2 Challenges in Multi agent reinforcement learning

There are several challenges when dealing with several agents in a MARL setting, as highlighted by Wong et al. [41]. For the context of this thesis we will highlight the following four challenges to MARL:

- The computational complexity involved with simulating and training several agents.

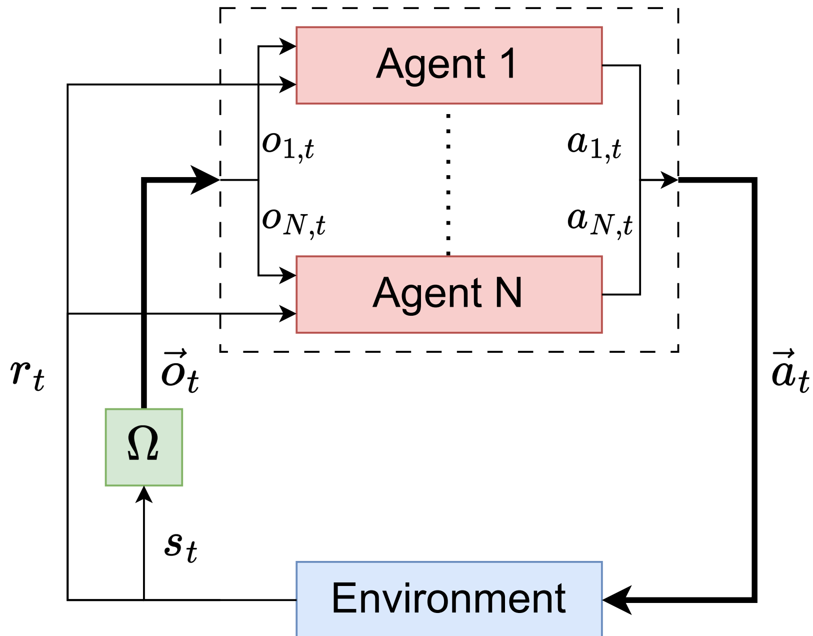

Figure 5: This figure shows how the agents interact with the environment in the DecPOMDP framework. At each timestep t the environment outputs the state s_t and reward r_t. The stochastic observation function creates the observation vector $\vec{o}_t$. Each agent i receives the reward r_t and its own observation o_t^i as inputs. Using its own policy i then produces its action a_t^i. All actions for timestep t are concatenated into the vector $\vec{a}_t$, which is passed back to the environment. The environment then transitions into the next state s_{t+1} after which the loop is repeated.

- The assumption that the environment is stationarity for a given agent, does not hold when there are several learning agents.
- When several agents perform actions simultaneously it becomes a challenge to assign the credit to the agent that caused that reward to be obtained.
- The partial observability of the environments might also pose a challenge to learning agents.

Note that these challenges are not mutually exclusive from one another, they can compound on each other to exacerbate their issues.

Computational complexity Sample efficiency is already a significant bottleneck in the single-agent RL setting. For example, as mentioned by Ding and Dong [6] a learning relatively simple game like pong already requires an agent to observe about ten thousand samples. However, this bottleneck is even more pronounced in multi-agent settings due to the non-stationarity and the lack of the Markov property. Not only are there more agents and policies to model, requiring more computational resources, but the changing policy of one agent effectively changes the environment for all other agents. If the environment and thereby, the optimal policy change over time, it can be expected that agents need even more training samples to find the optimal policy.

Non-stationarity In MARL problems multiple agents interact within the same environment. This means that the state transitions from a given state to the next state no longer depend solely on the actions of any single agent. Moreover, if a given agent learns, thereby changing its policy, it will effectively

change the environment for all other agents. This means that the optimal policy for any given agent changes over time as the policy of other agents changes. Therefore we can state that the environment is no longer stationary. This in turn causes most RL algorithms that require stationarity for convergence to fail.

Credit Assignment In the DecPOMDP framework, all agents receive the same collective rewards. This can make it difficult for agents to determine their own contribution to the joint rewards. In the worst case, this can lead to the lazy-agent problem, in which one or more agents perform seemingly random actions whilst the others solve the problem. This generates rewards for all agents, thus reinforcing the random behavior of the lazy agents. Alternative frameworks have agents obtaining more local reward signals as a direct result of their own actions. This can lead to selfish behavior in agents; they will attempt to increase their own rewards as much as possible, even at the cost of the total joint reward.

Partial Observability In partially observable environments, agents do not have all the information about the full state of the environment. This does create challenges in RL settings, but these have been overcome before. A good example of this would be the Atari games DQN plays, by Mnih et al. [28] This means that partial observability in itself is not an unsolvable problem. However, it does exaggerate the other problems that are prevalent in the MARL setting:

- When an agent i cannot observe another agent j obtaining a reward for the group, it becomes even more difficult for agent i to determine its contribution to this reward.
- On one hand, partial observability means that the input space for the agents is smaller, since entries in the observation space are smaller than those in the state space, thus allowing for smaller policies. On the other hand, partial observability also means that the agents need more samples to train, as each sample only contains part of the information about the full state. This increases the computation complexity of the algorithm.

Centralized Training and Decentralized Execution There are many ways to deal with MARL problems. Wong et al. [41] go into great detail about various strategies developed for MARL over time. For this thesis the most relevant strategy is centralized training with decentralized execution, as introduced by Kraemer and Banerjee [22] It is a hybrid strategy of two strategies developed before it: a fully centralized strategy: the centralized controller and a fully decentralized strategy: the independent learners.

In the centralized controller framework, the observations of all agents are concatenated into a single observation. This observation is passed to a single centralized controller with a single policy. Based on its policy, the centralized controller will choose actions for all agents to perform. This effectively reduces the MARL problem to a single-agent RL problem. However, the number of possible actions for the controller scales exponentially with the number of agents because the total action space is a Cartesian product of all individual action spaces. Therefore, the centralized controller suffers greatly from the issue of increased computational complexity.

In the independent learners framework, each agent has its own policy, which is optimized solely based on its own actions and observations. This way each agent effectively treats all other agents as part of its environment. This greatly reduces the complexity of the policy required, because the sum of the number of outputs of the individual policies is significantly smaller than that of the centralized controller. However, this approach ignores the issues of credit assignment, non-stationarity, and the lack of the Markov property.

The centralized training with decentralized execution still has N policies for N different agents, which all execute their actions independently from one another. However, during training the agents can access information that is hidden from them during execution. This information includes the observations, actions, rewards, policies, and gradients of the other agents. This knowledge-sharing during the training mitigates the credit assignment problem because all agents can see the combined observation and action that lead to a reward or penalty.

2.3.3 Value Decomposition Networks

In this thesis we will evaluate Value Decomposition Networks (VDNs), which were first introduced by Sunehag et al. [37]. VDN uses a centralized training and decentralized execution approach. The central assumption that allows VDNs to function is that the total collaborative Q-function can be additively decomposed into the individual Q-functions of all agents. Equation 19 formally expresses this assumption.

$$Q_\pi(s_0, \dots s_t, a_0^1, \dots, a_0^N, \dots a_t^1, \dots a_t^N) \approx \sum_{i \in I} Q_\pi^i(s_0, \dots, s_t, a_0^i, \dots a_t^i) \tag{19}$$

Note that the Markov property does not hold for MARL problems, nor does the stationarity assumption. Therefore Equation 19 shows the full history of all past states $(s_0, \dots, s_t)$ and all past actions $(a_0^i, \dots a_t^i)$ for every agent i.

The Q^i are learned by applying the temporal difference rule as described by Equation 10 in Section 2.1.3. The reward r_t in this case is the total cooperative reward. VDNs make no extra distinction as to which agent supposedly obtained this reward. In the case of partially observable environments, the state s_t in the Q-function also needs to be replaced by the observation o_t^i of agent i. When taking into account that each agent only optimizes over its own actions a_t^i we find that the update rule for a VDN's temporal difference rule changes to Equation 20, in which $\alpha \in [0, 1] \subseteq \mathbb{R}$ is the learning rate.

$$Q_{\texttt{new}}^i(o_t^i, a_t^i) = Q_{\texttt{old}}^i(o_t^i, a_t^i) + \alpha \cdot \left[r_t + \gamma \cdot \max_{a_{t+1}^i} \left(Q_{\texttt{old}}^i(o_{t+1}^i, a_{t+1}^i) \right) - Q_{\texttt{old}}^i(o_t^i, a_t^i) \right] \tag{20}$$

Like DQN, discussed in Section 2.2.3, VDN also uses experience replay, target networks, and an ϵ-greedy search strategy. However, unlike DQN, VDN also uses recurrent layers as described by Rummelhart et al. [34], a dueling layer as described by Wang et al. [40], and eligibility traces as introduced by Harb and Precup [13].

In a recurrent neural network (RNN) or a recurrent layer, there is an input and output similar to a regular NN layer. However, the output of the previous iteration acts as a hidden state, which is concatenated to the outside input of the current iteration to form the total input of the current iteration. The resulting output will then be concatenated with the outside input for the next iteration, this continues indefinately. This way, recurrent layers can preserve information about previous inputs. In an RL context, it also allows them to take actions at timestep t based on observations of previous timesteps. RNNs can suffer from the vanishing and exploding gradients problems. To combat this, two alternative architectures were proposed, the LSTM and the GRU.

- The Long Short Term Memory (LSTM), as introduced by Hochreiter and Schmidhuber. [16], which has two hidden states: the short-term and long-term memories. The short-term memory also functions as the output of the LSTM, whilst the long-term memory is only passed to future iterations of the LSTM. The LSTM has three gates to determine which information is preserved and/or passed to the output. These are the forget gate, the input gate, and the output gate.

- The Gated Recurrent Unit (GRU), as introduced by Cho et al. [4], which was inspired by the LSTM. It has only one hidden state like the RNN. The three gates of the LSTM have two counterparts in the GRU: the reset gate and the update gate.

If you are interested in a more detailed explanation of the differences between RNNs, GRUs and LSTM please read: Olah [31].
A dueling layer in a Q-learning NN is used to split Q-values into state-value and action-value components, respectively the value $V(s)$ and advantage $G(s, a)$. The sum of these two defines the Q-value $Q(s, a) \doteq V(s) + G(s, a)$. The advantage is constrained by the fact that it must sum to zero when summed over all actions for a given state s: $\sum_a G(s, a) = 0$. Splitting the Q-value like this does not make a difference for the final output. But it does increase the contrast between the different actions when the value $V(s)$ of the state is high compared to the difference in advantage $G(s, a)$ for the different actions. This makes it easier to distinguish between different actions, increasing the ease of training.

VDNs also use eligibility traces over several timesteps to update their policy. Unlike the temporal difference, an eligibility trace evaluates the value V of state s_{t+n} after n timesteps. Combining that with the rewards obtained in all future states leading up to s_{t+n}. The eligibility trace of n future timesteps, with rewards r_t and discountfactor γ is defined in Equation 21.

$$R_t = \sum_{i=0}^{n-1} \gamma^i \cdot r_{t+i} + \gamma^n \cdot V(s_{t+n}) \tag{21}$$

Figure something highlights the overall architecture of a VDN for a 2-agent environment. Note that the architectures for both agents are identical, although each agent has its own weights θ, and thus its own policy π. Both agents have three layers: a fully connected layer with the ReLu activation function, followed by an LSTM layer, which also has the ReLu activation function. The final layer of both agents is a Dueling layer, this layer provides the final Q-values for each action that the agent can take. The agent will then choose its action based on the ϵ-greedy search strategy, based on the Q-values that it determined. Finally, during training, the Q values of both agents are summed up. The sum of these Q-values is then used to define a loss and perform backpropagation, minimizing that loss.

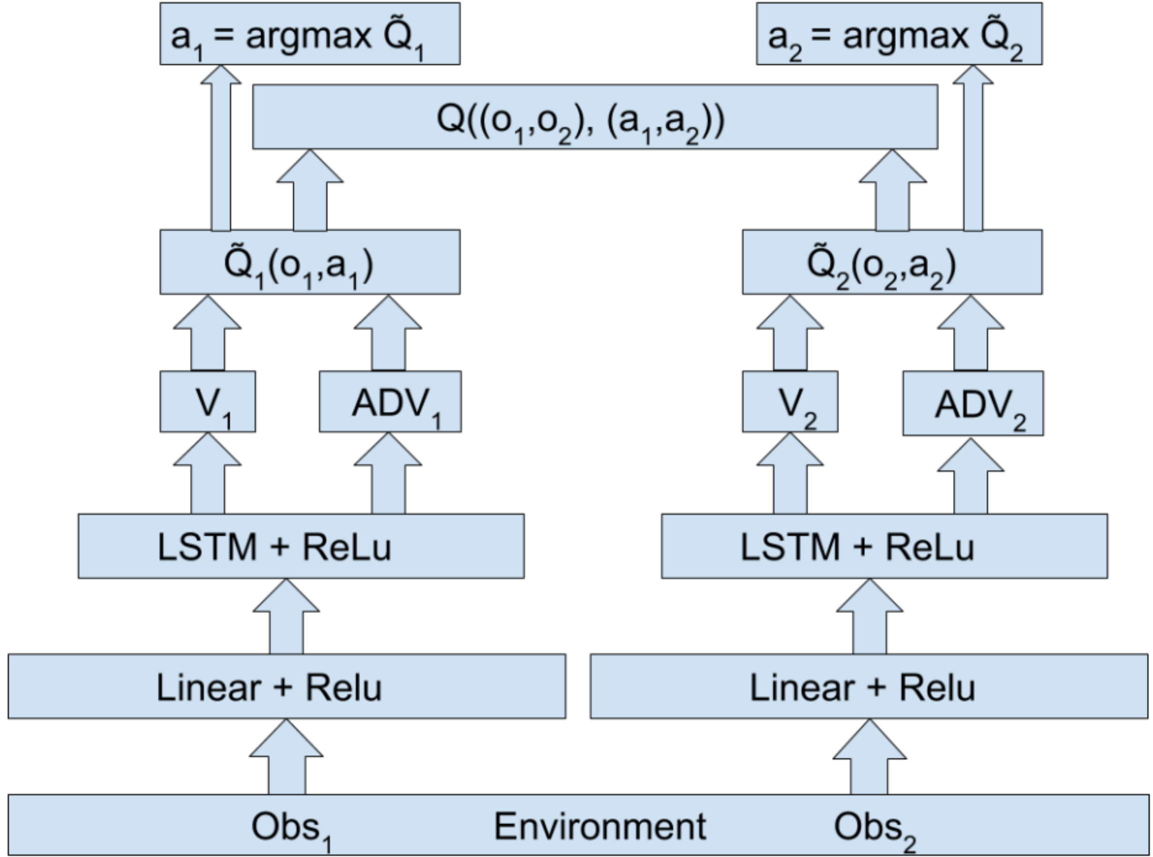

Figure 6: This figure illustrates the architecture and the flow of information in a 2-agent VDN. From the bottom to the top, the environment provides the observations Obs_1, Obs_2 that are the inputs for the respective agents. The first layer for both agents is a linear NN layer followed by the ReLu activation function. The second layer is an LSTM layer which also followed by the ReLy activation function. The final layer of the network is the dueling layer, which computes the value V of the state and the advantage ADV of all possible actions in that state. V and $ADV(a)$ are then summed up to compute the individual Q-values $\widetilde{Q}(o, a)$. The individual Q-values can be summed during training to form the combined Q-value $Q((o_1, o_2), (a_1, a_2))$. However, the selection of actions is solely based on the individual Q-values $\widetilde{Q}(o, a)$. This figure was taken from Sunehag et al. [37]

2.4 Evolutionary Algorithms

This section will discuss the theory behind Evolutionary Algorithms (EAs). The information in this Section is mostly based on the book by Eiben and Smith [8]; this book will provide a more in-depth discussion about all kinds of evolutionary algorithms. EAs are optimization algorithms inspired by biological natural selection. The core idea behind EAs is that there is a population P of Π potential solutions; referred to as individuals, that can each be assigned a fitness by some objective function. EAs are most effective on problems where it is possible to clearly define a score or fitness to a solution i.e. optimization problems, such as aerodynamic optimization, resource optimization, etc.
EAs typically consist of six phases:

- **Initialisation**: First, the population P is initialized from the domain $\mathcal{I}$. This initialization is often random, but this is not strictly necessary. This step is also the only step that is typically not repeated unless restarts are involved in the training procedure.

- **Parent Selection**: In this step, μ parents used in the recombination or mutation steps are selected. In the most general case, a single reproduction interaction requires n parents and generates m offspring. Therefore the general selection function looks as follows: $\mathcal{S}_p : \{P \times \mathbb{R}\} \rightarrow \{(P)^n\}$. In the most common case of $n = 2, m = 1$ that translates to: $\mathcal{S}_p : \{P \times \mathbb{R}\} \rightarrow \{P \times P\}$. After all recombination and mutation steps have been completed, a total of λ offspring will have been created. In Section 2.4.5 we will discuss how parents are selected.

- **Recombination**: this is applied to n individuals; the parents, combining some of their traits into m new individuals; the offspring or children. In the most common case, we see that $n = 2, m = 1$, similar to sexual reproduction, from which it takes inspiration. In general, this function would look like this: $\mathcal{R} : (P)^n \rightarrow (P')^m$. However, in the most common case of $p = 2, c = 1$ this simplifies to $\mathcal{R} : P \times P \rightarrow P'$. In Section 2.4.4 we will go into further detail about recombination.

- **Mutation**: this is only applied to one individual, and will always generate only one new individual, which will differ slightly from its parent. This function looks like this: $\mathcal{M} : P \rightarrow P'$. In Section 2.4.3 we will discuss mutation in further detail.

- **Evaluation**: In this step, each individual x_i is evaluated on the environment, and assigned a score; its fitness y_i. This function looks as follows: $\mathcal{E} : P \rightarrow P \times \mathbb{R}$

- **Survivor Selection**: In this step, the survivors that pass on to the next generation are selected based on their fitness. This function looks as follows: $\mathcal{S}_n : \{P \times \mathbb{R}\} \rightarrow \{P' \times \mathbb{R}\}$. In Section 2.4.5 we will discuss the most common methods for selecting offspring.

Figure 7 illustrates the overall flow chart of an EA and illustrates how the different phases interact with each other. This figure takes inspiration from Eiben and Smith [8].
EAs are a subset of the so-called generate-and-test algorithms. This means that they do not create partial solutions to build up solutions on a step-by-step basis. Instead, a generator first generates a full candidate solution, after which the quality of this solution is tested. Or in EA terms, an individual is generated, and then its fitness is evaluated.

EAs use the recombination and mutation phases to introduce variations into the population. The evaluation and selection phases then ensure that the best individuals are retained in the population, increasing the population's fitness over time. An overview of the pseudo-code for EAs can be seen in

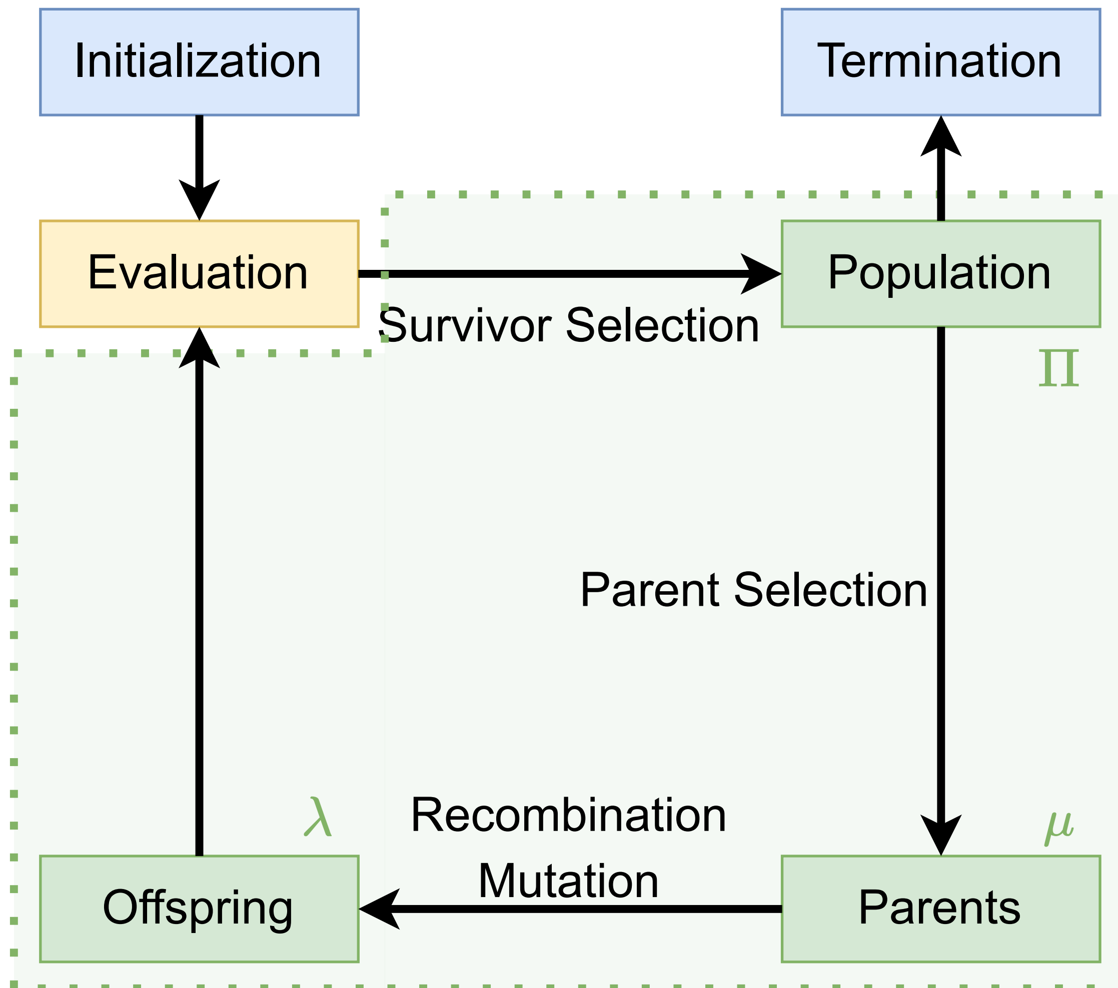

Figure 7: This is a flow chart of an evolutionary algorithm (EA). The steps that are considered part of the algorithm proper are highlighted in green. The process starts by initializing the algorithm and the population. The population is the core of the EA. The first step of the EA loop is to evaluate the population so that the parents can be selected. Once the parents are selected the offspring can be generated via the mutation and recombination operaters. Finally, the Offspring are evaluated such that the population for the next generation can be selected amongst the offspring (and the parents). This loop is repeated until the termination criteria are satisfied.

Algorithm 1.[3] This pseudocode was inspired by the pseudocodes from Wong [43], Bäck and Schwefel [3], and Ostermeier et al. [33].

The following sections will cover the general theory behind EAs and their various phases. The content is primarily based on the book by Eiben and Smith [8]. This book goes into more detail about AEs, so please read it for a more in-depth discussion of EAs.

[3]Note that the most common case of $n = 2, m = 1$ was used in this algorithm.

Algorithm 1: Evolutionary Algorithm

```
t ← 0                                                          ▷ generation counter
P(0) ← {x_1, ..., x_λ} ∈ I                                     ▷ Initialize population
F(0) ← {(x_1, Φ(x_1)), ..., (x_Π, Φ(x_Π))}
while ¬T do                          ▷ Continue until termination criterion is reached
    P'(t) ← S_p(F(t))                                          ▷ Select parents
    P''(t) ← ∅
    for (x_i, x_j) in P' do
        x'_i ← R(x_i, x_j)                                     ▷ Perform recombination
        P''(t) ← P''(t) ∪ {x'_i}
    end
    P''' ← ∅
    for x'_i in P''(t) do
        x''_i ← M(x'_i)                                        ▷ Mutate new individuals
        P'''(t) ← P'''(t) ∪ {x''_i}
    end
    F' ← ∅
    for x_i in P''' do
        (x_i, Φ(x_i)) ← E(x_i)                                 ▷ Evaluate all individuals
        F'(t) ← F'(t) ∪ {(x_i, Φ(x_i))}
    end
    F(t+1) ← S_n(F(t) ∪ F'(t))                                 ▷ select next generation
    P(t+1) ← {x_i : (x_i, Φ(x_i)) ∈ F(t+1)}
end
```

2.4.1 Representation, Phenotype, and Genotype

Most optimization problems are not formulated in a way that allows us to optimize them directly using EAs. Therefore, it can be necessary to come up with a mapping from potential solutions; the phenotype, to some list or numerical representation; the genotype. For example, when optimizing a problem involving graphs (phenotype), it could be useful to represent the graphs as an adjacency matrix (genotype), this matrix can be flattened into a single list for an even compacter genotype. It is critical that the genotype space is sufficiently expressive to include a representation of the optimal phenotype. Since this optimal phenotype is typically unknown beforehand, the best practice is to have a complete generator i.e. have the genotype space represent the full phenotype space.

Using genotypes and phenotypes, a generator can easily generate new individuals by specifying a genotype. The algorithm can easily convert that genotype to a phenotype to get a candidate solution. This candidate solution can then be evaluated using the evaluation or fitness function to assign a score to the individual. The fitness function's accuracy and precision are key, as this ultimately decides what an optimal solution is. Therefore it can be crucial to perform multiple evaluations in a stochastic environment.

One of the most common representation types is that of a vector or an array. This can be an integer array, floating-point vector, or even a bit-string. Other common representations are the permutation list representation and the tree representation. Each of these has its own respective mutation and crossover operators as well as differing definitions of diversity. We will go into more detail about the diversity, mutation, and crossover for real-valued floating-point representations in Sections 2.4.2, 2.4.3, and 2.4.4, respectively. We will not cover binary, integer, or permutation representation as we

are not applying it in this study. They are, however, covered extensively in the book by Eiben and Smith [8].

A real-valued vector representation would look as follows: each individual $x_i \in P$ is represented by an $d-$dimensional vector $\vec{x}_i \doteq (x_i^1, ..., x_i^d)$. Such that for every index $k \in [1, d],\ x_i^k \in \mathbb{R}$.

2.4.2 Population and Diversity

One of the key strengths of EAs is that they do not optimize one solution at a time. Instead, an EA will have a population of individuals which are optimized in steps called generations. The diversity of this population is a measure of how many different individuals are present within a population and of the average distance between individuals in the genotype space. Note that two individuals with the same fitness or even phenotype do not necessarily have the same genotype. But the reverse does hold true: two individuals with the same genotype will always have the same phenotype. They will have the same fitness too, if the evaluation function is not stochastic.

A more diverse population allows for more exploration: making it easier to escape local optima. Whereas a less diverse population around an optimum allows for more exploitation of the known geometry of that part of the search space.[4] During the initial stages of a typical EA run, the diversity of the population tends to increase as the algorithm traverses the search space. However, towards the end of the run, the diversity will decrease, as the search converges towards the (global) optimum.

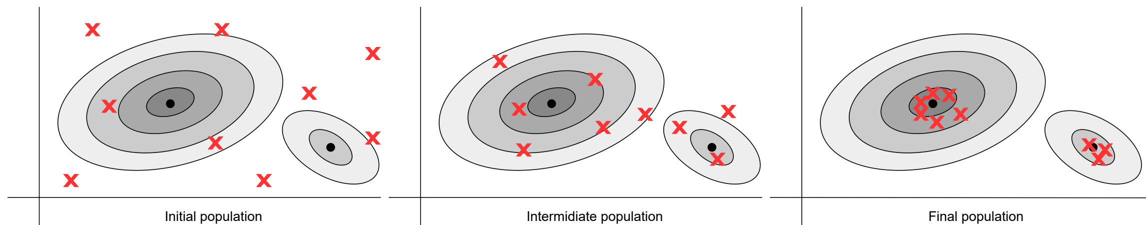

Figure 8: This figure illustrates the diversity of nine individuals in a simple 2-dimensional environment. The red 'X' denote the individuals, the shades of gray and the contour lines denote the fitness values, and the black dots represent the (local) optima. On the left, we see the initial distribution of the population, which in this case is random. In the center, we see that the population has moved closer towards the two local optima and therefore closer to each other as well. On the right, we see that all individuals have converged onto one of the two local optima.

When running an EA it is the population that changes not the individuals themselves. The population is only changed by replacing old individuals with new individuals. Variation operators introduce change into the population by creating new individuals the children from existing individuals: the parents, they always act only on the genotypes of individuals, not directly on the phenotypes. The two most common variation operators are mutation and recombination. Mutation acts on only a single individual, whilst recombination acts on two or more individuals, more on these operators can be found in Sections 2.4.3 and 2.4.4, respectively.

[4] note that this optimum might not necessarily be the global optimum.

2.4.3 Mutation

The mutation operator always has a single parent and a single offspring. The offspring is typically similar to the parent i.e. close to it in terms of the genotype space. The way in which the child differs from the parent is stochastic. There are also operations that act in a non-stochastic way on single individuals. These include operations to increase the feasibility of an individual: for instance, rounding specific numeric values to the closest integer value. Another example of a non-stochastic operation on single individuals is applying corrections to some values of an individual to keep it within the bounds specified by the search space. Note that non-stochastic operations such as these are never classified as mutation, and that true mutation is always (at least partially) stochastic. If any genotype can mutate into any other genotype with a non-zero probability, then that implies that an EA including mutation will, in theory, always converge to the global optimum if it runs long enough. However, in practice the run time required can be astronomical in the worst case, meaning this theoretical guarantee does not always hold in practice. This does, however, illustrate that the mutation operator can be key in the creation of "new" genotypes.

For an individual represented by a $d-$dimensional floating point vector the mutation operator $\mathcal{M} : \vec{x}_i \rightarrow \vec{x}'_i$ works as follows: $(x_i^1, \ldots, x_i^d) \rightarrow (x_i^{1\prime}, \ldots, x_i^{d\prime})$ with at least one index k such that $x_i^k \neq x_i^{k\prime}$ to ensure that $\vec{x}_i \neq \vec{x}'_i$. When implementing mutation, it is key to ensure that the offspring do not differ too much from their parents. For real-valued representations, there are two schools of thought for ensuring that: either change only some of the entries x_i^k of the vector $\vec{x}_i$: uniform mutation, or change them all, but by a small amount: non-uniform mutation.

Uniform mutation In this type of mutation there is a probability p^k assigned to each entry x_i^k with index k, of the vector $\vec{x}_i$ representing individual i. The probability p^k represents the chance that x_i^k will be replaced by a uniformly sampled new random value $x_i^{k\prime}$. In the simplest case, p^k is constant for all $k \in [1, d]$, but it is also possible to have p^k that are different for different values of k. These different p^k can be saved in a vector $\vec{p} \doteq (p^1, \ldots, p^d)$. There are even implementations in which the vector $\vec{p}$ is also subject to evolution during the run of the EA.

Non-uniform mutation In most nonuniform mutation methods this kind of mutation takes the form of a step Δx_i^k added to each old value x_i^k. In order to ensure that the offspring is not too different from the parent this Δx_i^k needs to be small, at least the expected value needs to be small. To achieve this it is typical that Δx_i^k is normally distributed with a mean around zero, as indicated by Equation 22. This way the step is equally likely to be positive or negative. The standard deviation σ can be tuned to determine the expected step-size $|\Delta x_i^k|$ for each parameter x_i^k.

$$p(\Delta x_i^k) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-\frac{(\Delta x_i^k)^2}{2(\sigma)^2}} \tag{22}$$

Figure 9 shows a visualization of these different σ strategies in a two-dimensional example, with a contour at the expected values of $\vec{\Delta}_i$, given the dot at $\vec{x}_i$. On the left, we can see that the expected values of $\vec{\Delta}_i$ form a circle (or hyper-sphere in d dimensions) when σ_i is constant for each entry k. On the right, we can see that the expected values of $\vec{\Delta}_i$ form an ellipse when σ_i is different for each entry k.

Self-adaptive non-uniform mutation The key idea behind self-adaptive mutation is to not have one global σ for all individuals, but rather have an individual σ_i for each individual $\vec{x}_i$. σ_i can then be added to the genome of $\vec{x}_i$, such that it evolves with $\vec{x}_i$. It is then advantageous to first update/mutate

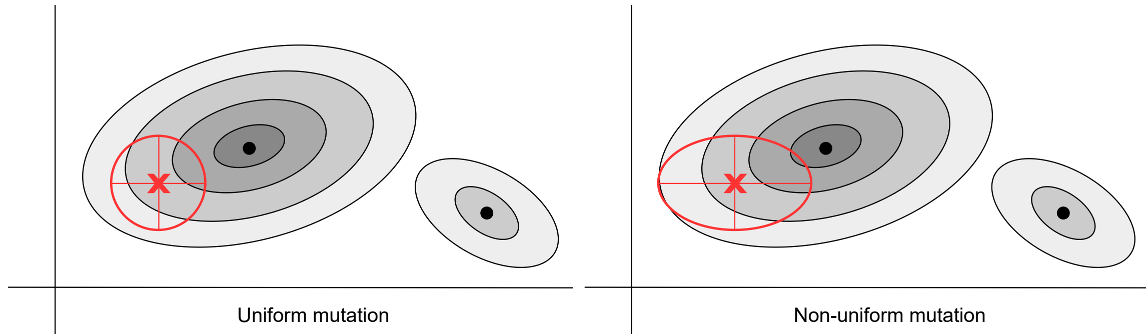

Figure 9: A 2-dimensional visualization of different strategies for σ. The big dot within the shape represents the current individual $\vec{x}_i$. The contour surrounding this dot represents the expected placing of $\vec{x}'_i$ given the current standard deviation vector: $\vec{\sigma}_i$. On the left, we see that each dimension/parameter/axes has the same σ, in other words: $\sigma_i^1 = \sigma_i^2$. This results in a circular contour for σ. On the right, we see that each dimension has its own σ, $\sigma_i^1 \neq \sigma_i^2$ resulting in an elliptical contour orthogonal to the axes of the plot. This Figure was inspired by Eiben and Smith [8]

the value of σ_i, then use the new value of σ'_i to mutate $\vec{x}_i$ into $\vec{x}'_i$. This way the new value σ'_i is indirectly evaluated by the performance of $\vec{x}'_i$.

This can even be taken one step further by not only having unique σ_i for each individual $\vec{x}_i$ but also having a unique σ_i^k for each entry x_i^k of $\vec{x}_i$. This resulting vector $\vec{\sigma}_i \doteq (\sigma_i^1, \ldots, \sigma_i^d)$ can be likewise be added to the genome of $\vec{x}_i$ for co-evolution.

2.4.4 Recombination

The recombination or crossover operator always has $n \geq 2$ parents and it can generate one or more children by combining the genotypes of the parents. This operator takes inspiration from biological sexual reproduction, which always has two parents. This is why recombination in EAs typically involves only two parents, but extending most implementations of recombination to $k \in \mathbb{N}$ parents is trivial. In the remainder of this section, we will discuss the two most common strategies of recombination: discrete recombination and arithmetic recombination, as well as their respective sub-strategies. The information in this section is gathered from the book by Eiben and Smith [8], which also contains a detailed explanation of other recombination strategies such as blend recombination and recombination for other representation types.

Discrete Recombination For real-valued vector representations it is possible to perform discrete recombination by creating a new individual from two parents by directly copying the vector entries from the parents to the respective entries of the child.[5] After selecting the parents $\vec{x}_i$ and $\vec{x}_j$ we still need to decide which entry to select from which parent to create offspring $\vec{x}'$. There are two main approaches to this: The first approach is uniform recombination, in which the parent for each entry is selected (uniformly) at random. The second approach is to introduce one or more crossover-points, such that all entries are copied from $\vec{x}_i$ until the first crossover-point is reached. From then on all entries are copied from $\vec{x}_j$, until the next crossover-point is reached. Once that point is reached the entries will be copied from $\vec{x}_i$ again until the next crossover-point is reached. This pattern continues until there are no more entries to be copied and a full vector has been constructed.

[5]This can trivially be expanded to n parents but that is beyond the scope of this thesis

Figure 10 shows an example of these discrete recombination approaches for a set of nine-dimensional vectors, this figure was inspired by Eiben and Smith [8].

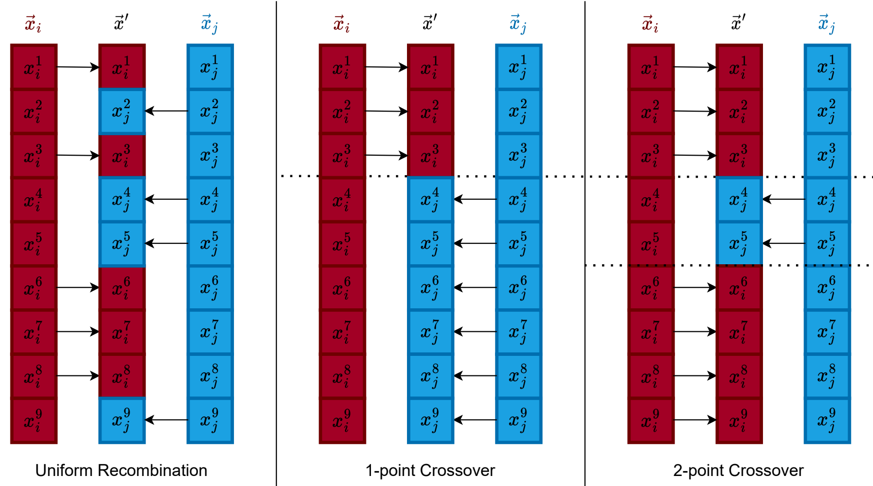

Figure 10: A nine-dimensional example of different discrete recombination strategies from two parents $\vec{x}_i$ and $\vec{x}_j$ creating offspring $\vec{x}'$. On the left we see uniform recombination, in the center, we see a 1-point crossover at index 4, and on the right, we see a 2-point crossover at indices 4 and 6.

Arithmetic Recombination Arithmetic recombination combines some or all of the entries by taking a weighted average over all of the selected parents. In the most general case with n parents: $\vec{x}_q$ for $q \in [1, n]$, the entries with index k can then be combined as follows:

$$x'^k = \sum_q \alpha_q \cdot x_q^k \tag{23}$$

with the condition that the weights are normalized: $\sum \alpha_q = 1$. In the most common case with two parents: $\vec{x}_i, \vec{x}_j$ Equation 23 can be simplified to :

$$x'^k = \alpha \cdot x_i^k + (1 - \alpha) \cdot x_j^k \tag{24}$$

When there are n parents arithmetic recombination will result in n children, each copying some entries from one of the parents and taking the combined value for the entries that were not copied. Similarly to discrete recombination, there are several strategies for deciding which entries of the vector will be copied from a parent and which will be combined from several parents, we will discuss three approaches in the remainder of this section. Firstly, there is simple arithmetic recombination, which is analogous to $n-$point crossover in the sense that it selects one or more indices that indicate when to alternate between copying from parent q and taking the weighted average as computed in Equation 23. Secondly, there is single arithmetic recombination. As the name suggests this method only changes one entry per parent and copies all other entries. Thirdly, there is whole arithmetic recombination. In this case, a new vector is created consisting fully of the combined entries of all parents. This also means that it will result in n times the same vector when n parents are selected. Figure 11 shows a visualization of these three approaches for arithmetic recombination.

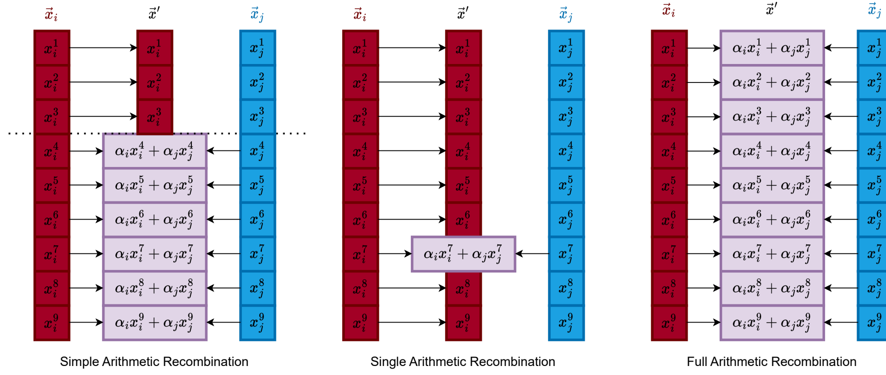

Figure 11: A nine-dimensional example of different arithmetic recombination strategies from two parents $\vec{x}_i$ and $\vec{x}_j$ creating offspring $\vec{x}'_i$. On the left we see simple recombination which switches after index 3, in the center, we see single arithmetic recombination at index 7, and on the right, we see full arithmetic recombination.

2.4.5 Selection

The mutation and recombination operators discussed in Sections 2.4.3 and 2.4.4, respectively create λ new individuals; potentially increasing the size of the population to $\mu + \lambda$. Since these processes are (at least partially) randomized, there needs to be some mechanism to ensure that the average fitness of the population increases over time. This is the function of the selection operator. There are two main types of selection: Parent selection and Survival selection, any EA must include at least one of these selection methods in order to evaluate and ensure the fitness of the population. Parent selection involves finding the right μ parents for the recombination or mutation stage, the idea is that parents with a high fitness are more likely to create offspring with a high fitness. Survival selection involves reducing the population size from $\mu + \lambda$ back to μ, by selecting μ individuals to survive. Typically the individuals with a higher fitness have a better chance of surviving.
Alternatively, survival selection can take place directly after a new individual has been created. In this case, the child is competing directly against its parent. This is often called (μ, λ) selection.
There are several factors that can be considered when selecting individuals, in the rest of this section we will highlight several selection strategies that are commonly used in EAs. For a more detailed overview of these and other strategies, please read the book by Eiben and Smith [8].

Fitness based selection Perhaps the most obvious selection criterion is the fitness of the individual. In fact, selection based on fitness in either the parent selection or survivor selection phase is required for the convergence of the algorithm. A benefit of selecting based on fitness is that this can improve the mean fitness of the population quite rapidly. However, selecting solely on fitness may also lead to premature convergence to a locally optimal solution.

Selection based on fitness can be as straightforward as selecting the μ best individuals to reproduce or to continue to the next generation. When selecting survivors in this way, this strategy is called replace worst, more on this in the book by Eiben and Smith [8]. In some algorithms, one (base) parent produces one offspring. In this case, this offspring might directly compete against its parent to be selected for the next generation. An extreme form of the replace worst strategy is called Elitism, in this strategy only the single fittest individual is guaranteed to carry on to the next generation, the other individuals are also subject to some other selection strategy, be that probabilistic, age based or some other strategy.

There is also fitness proportional selection, in which each individual is assigned a probability to be selected, this probability is proportional to the fitness of the individual, as the name suggests. The probability $fps(s)$ assigned to an individual i is defined in Equation 25. In this equation, f_i denotes the fitness of individual i.

$$fps(i) = \frac{f_i}{\sum_{j=1}^{\mu} f_j} \tag{25}$$

As we can see Equation 25 creates a selection probability that is directly proportional to the fitness of a given individual. One issue that may arise from this is that selection pressure will be lower when fitness values are close together, especially when the average fitness increases toward better solutions. For example, consider the case with two individuals: i_1 and i_2, with respective fitness values 2 and 4. At this point there is a $\frac{2}{3}$ probability of selecting i_2. After some generations, there are two individuals: i_1 and i_2, with respective fitness values 8 and 10. At this point there is a $\frac{5}{9}$ probability of selecting i_2. Which is significantly less, despite having the same absolute difference. To combat this phenomenon we can introduce windowing, which transposes all fitness values by subtracting the minimal fitness from all of them. The probability $wfps(s)$ assigned to an individual i when windowing is introduced is defined in Equation 26. In this equation, f_i denotes the fitness of individual i, and $f_{\min}$ denotes the lowest fitness in the population.

$$wfps(i) = \frac{f_i - f_{\min}}{\sum_{j=1}^{\mu}(f_j - f_{\min})} \tag{26}$$

Note that this selection strategy will never select the worst individual in the population, since the numerator of this fraction will be $f_{\min} - f_{\min} = 0$.

Age Based selection When selecting survivors it can be beneficial to consider the age of an individual. Potentially removing older individuals with a higher fitness might seem counter-intuitive. However, it can be necessary to escape locally optimal solutions in certain circumstances. Age based selection can be implemented in two main ways: by removing an individual from the population after it has survived a predefined number of generations, or by decreasing the survival probability of an individual proportional to its age. The latter strategy tends to also be combined with fitness based selection. In either case, it is key that the algorithm also selects for fitness at some point. Either by combining age and fitness based survivor selection, or by having a (partially) fitness-based parent selection.

2.4.6 Stopping criteria

In theory, an EA could run forever. In practice, we would like the algorithm to stop eventually, to obtain a usable solution to the problem. Fortunately, EAs are an anytime algorithm, meaning they can be stopped at any time and they will still produce a solution, albeit a sub-optimal one. In the case of EAs, this solution will be the individual with the highest fitness. In general, there are 5 possible criteria to determine when an EA stops:

- If the optimal score is known or if a desired score is specified the algorithm can be stopped once it has approached the optimal score closely enough or if the desired score has been exceeded. At this point, a suitable solution has been found and there is no need for further optimization.
- The optimization can be stopped if a pre-specified number of generations or fitness evaluations has been exceeded. This can be done to prevent the algorithm from running forever. Or it can be done to create a fair comparison against other algorithms using similar resources.

- The optimization can be terminated after a pre-specified amount of time has elapsed, for the same reasons as mentioned above.
- The improvement of the fitness over the last x amount of generations or fitness evaluations has not been enough, indicating that the optimization has plateaued.
- The diversity of the population has fallen below a certain threshold, indicating the search has started to converge on some (local) optimum.

A key observation about EAs is that the Mutation and Recombination steps are stochastic. Which part of each parent is selected for which child during recombination, as well as which parts of the individual are changed during mutation, are determined stochastically. This makes EAs gradient-free algorithms.[6]

2.4.7 Advantages and disadvantages of EAs

A key advantage of EAs is that a diverse population allows the algorithm to escape local minima in the search space more easily, thus making EAs suitable for rugged search spaces, with a lot of local optima. The population also makes the algorithm more robust against random noise in the evaluation: If a good solution is discarded due to randomness, another, similar solution likely exists within the population.
The population of solutions does come at the cost of greater computational cost since all new individuals need to be evaluated at every generation.

EAs are also typically simple to implement; only requiring the user to specify the evaluation function and phenotype representation. EAs also do not require many conditions for the functions that can be optimized, only really requiring that the evaluation function does not change too radically from one generation to the next. As opposed to for example reinforcement learning, which requires the Markov property to hold for a given problem.

One final drawback is that EAs can scale poorly with the dimensionality of a problem. An increased dimensionality increases the size of the search space, making it less likely to randomly find the solution. The population size of the algorithm also typically scales with the dimensionality of the problem, thus requiring more evaluations per generation for problems with a larger dimensionality.
For a more in-depth look into EAs we recommend the book by Eiben and Smith [8].

2.5 Differential Evolution

Differential Evolution (DE) is an EA that was introduced by Storn and Price [36] in 1997. It was designed with continuous search spaces in mind. As such, the population in DE consists of a set of Π, d-dimensional real-valued vectors, where d corresponds to the number of parameters of the objective function $f : D \subseteq \mathbb{R}^d \rightarrow \mathbb{R}$. The initial population is sampled uniformly at random over the entire search space $\mathcal{I} \subseteq \mathbb{R}^d$; defined by the bounds of each parameter.

2.5.1 Mutation

The mutation phase for the DE algorithm does not involve creating a valid new individual by randomly tweaking some or all of the entries of the vector. Instead, the mutation operator requires three vectors to

[6] Unless gradients are somehow included in the objective function

create one mutation vector. This works as follows: Each vector $\vec{p}$ in the population will act as a parent vector once per generation.

From the remaining population three different vectors $\vec{b}$, $\vec{r}$, and $\vec{t}$ are selected from the population such that all four vectors: $\vec{p}$, $\vec{b}$, $\vec{r}$, and $\vec{t}$ are distinct. A new mutation vector $\vec{m}$ is then formed by combining $\vec{b}$, $\vec{r}$, and $\vec{t}$ as follows: $\vec{m} = \vec{b} + F \cdot (\vec{r} - \vec{t})$ A visual example for the case $d = 2$ can be seen in Figure 12. This mutation vector is not counted as a valid individual within the population, instead, it will be subject to recombination first, as we will discuss in the next section.

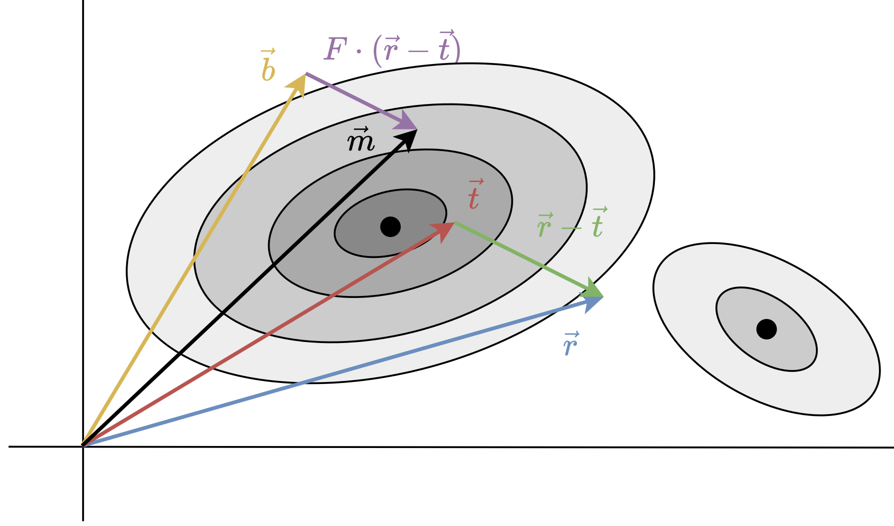

Figure 12: A two-dimensional example of the creation of mutation vector $\vec{v} = \vec{x}_b + F \cdot (\vec{x}_r - \vec{x}_t)$. In this example the parameter F has been set to 0.75 and the objective function has elliptical contour lines.

2.5.2 Crossover

Unlike most EAs, in the crossover for DE the crossover does not happen between two parents in the existing generation. Instead, the crossover happens between the parent vector $\vec{p}$ and the mutation vector $\vec{m}$. More formally: for each parent vector $\vec{p}_i$, with its corresponding mutation vector $\vec{m}_i$, a candidate vector $\vec{c}_i$ is created. There are two main forms of crossover for DE binomial and exponential crossover. Figure 13 gives a visualization of this process for a simple 9-dimensional example.

Binomial crossover is performed by looping over each index $j \in [1, d]$ of the candidate vector $\vec{c}_i$ and randomly selecting the entry at index j form either the mutation vector $\vec{m}_i$ or the parent vector $\vec{p}_i$. Such that: $\vec{c}_i^j = \vec{m}_i^j$ with probability CR and $\vec{c}_i^j = \vec{p}_i^j$ with probability $1 - CR$. Finally, one index k is selected to ensure that at least one entry of the trial vector is selected from the mutation vector: $\vec{c}_i^k = \vec{m}_i^k$; ensuring that $\vec{c}_i \neq \vec{p}_i$.

Exponential crossover is performed by first selecting an index $k \in [1, d]$ uniformly at random. The entry at index k is copied from the mutation vector to the candidate vector: $\vec{c}_i^k = \vec{m}_i^k$. Incrementally, a number $g = \mathcal{U}(0, 1)$ is generated uniformly at random for every following index. If $g < CR$ this entry is also copied from the mutation vector and the loop continues for the next entry of $\vec{c}_i$. This loop wraps back

to the beginning of the vector if the last index of $\vec{c}_i$ is reached. If $g \geq CR$, or index k is reached, this loop stops. At that point, all remaining entries in $\vec{c}_i$ are copied from $\vec{p}_i$.

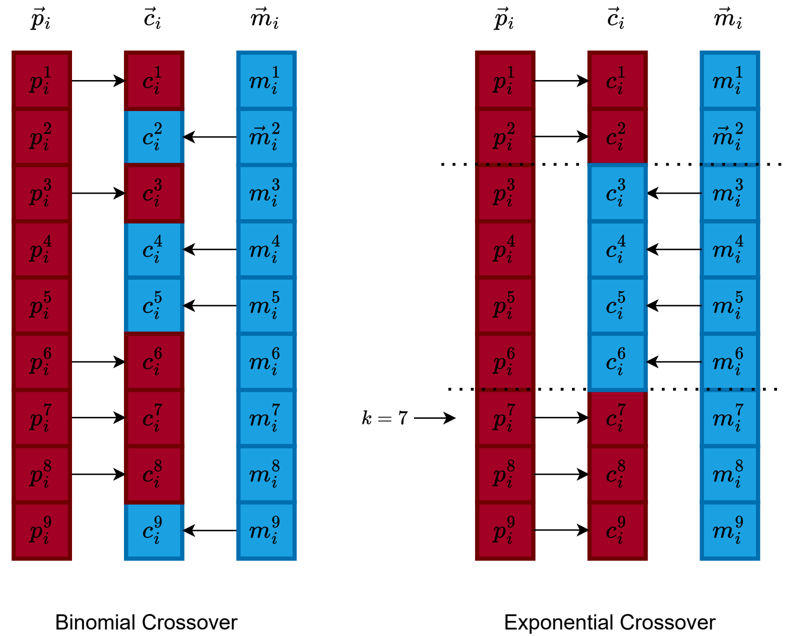

Figure 13: A five-dimensional example of the creation of candidate vector $\vec{c}_i$ from the parent vector $\vec{p}_i$ and mutant vector $\vec{m}_i$. On the left, binomial crossover is shown, indices 1, 3, 6, 7, and 8 were taken from the parent vector, and indices 2, 4, 5, and 9 were taken from the mutant vector. On the right, exponential crossover is shown, the starting index was set to $k = 7$ and the fifth generated number was larger than the crossover rate CR, stopping the loop at index 3.

2.5.3 Survivor Selection

In the survivor selection phase of DE the child competes directly against its parent in the (μ, λ) format: The fitness of the new candidate vector $\vec{c}$ (child) is evaluated and compared against the parent vector $\vec{p}$. Amongst these two individuals, the one with the highest fitness will carry on to the next generation, whilst the other one will be discarded. This elitist way of selecting the next generation ensures that solutions cannot get worse over time because the parent is only discarded if the child's fitness is greater than the parent's fitness. Thus making it impossible to end up with worse solutions in the next generation.

2.5.4 Parent Selection

Every individual in DE is selected to be the parent for crossover exactly once per generation. Therefore, the parent selection in DE happens for the base, reference, and target vectors. There are several strategies when selecting these vectors. Most commonly, these vectors are either selected randomly or from among the fittest k individuals. The combination of the selection method for the base vector $\vec{b}$ and the crossover method dictates the name of the DE variation. For example, when the base vector $\vec{b}$ is selected randomly and the crossover happens binomially, the DE variation is called rand1bin.

2.5.5 DE Algorithm

Combining the steps from Sections 2.5.1, 2.5.2 , and 2.5.3 will result in the full DE algorithm as seen in Algorithm 2. The parent selection in this version of the algorithm is random and the crossover strategy is binomial. This abbreviates to the rand1bin version of DE. This algorithm is inspired by Boks et al. [2].

Algorithm 2: Differential Evolution

```
Input: f                                                        ▷ objective function
Input: B                                                        ▷ bounds of each variable
Input: G                                                        ▷ budget; max number of generations
Output: P                                                       ▷ final population

for p⃗_i in P do
    p⃗_i ← U(B^min, B^max)                                       ▷ Randomly Initialise population
end
for g in [1, G] do                                              ▷ Loop over all generations
    for p⃗_i in P do                                             ▷ Loop over all individuals
        Choose b⃗_i, r⃗_i, t⃗_i ∈ P uniformly at random s.t. b⃗_i, r⃗_i, t⃗_i, and p⃗_i are distinct.
        m⃗_i ← b⃗_i + F · (r⃗_i − t⃗_i)                                ▷ Mutation
        for j in [1, n] do
            if U(0, 1) < CR then
                c⃗_i^j ← m⃗_i^j                                     ▷ Crossover
            end
            else
                c⃗_i^j ← p⃗_i^j                                     ▷ keep old entry
            end
        end
        k ← U(0, n) ⊆ ℕ
        c⃗_i^k ← m⃗_i^k                                ▷ Ensure Crossover happens at 1 index: k
    end
    for i in [1, M] do
        if f(p⃗_i) < f(c⃗_i) then
            p⃗_i ← c⃗_i                                             ▷ Elitist selection
        end
    end
end
return P                                                        ▷ return the final population
```

3 Problem Statement

As discussed in Section 2.3.2, the multi-agent setting provides several issues for most traditional RL methods. In this thesis, we will address these four main challenges, as illustrated by Wong et al. [41]:

- Non-stationarity: introducing extra agents breaks the stationarity assumption, since the transition function is no longer just dependent on the action of a single agent but joint action of all agents instead. This means that the Markov assumption is also no longer valid because the transition function for a single agent shifts as the policy of all other agents changes. This in turn means that the optimal policy for any given agent changes over time as the policies of all other agents change over time. Meaning that each agent is chasing a moving target.
- Credit assignemnt: It can be difficult for agents to determine if their own actions, or the actions of other agent, contributed to a reward. In the worst case, this can lead to the lazy agent problem in which one or more agents do not contribute to the common objective.
- Computational Complexity: Simulating several learning agents at once costs more resources to run. Moreover, sample efficiency is already a challenge in single-agent RL tasks. This is even worse in MARL settings in which challenges like non-stationarity and credit assignment can increase the number of samples required for convergence even further.
- Partial Observability: Agents do not have the full information about the state of the environment available to them. This is also a challenge in single-agent RL problems. On its own, it can be overcome. However, in MARL problems it can exacerbate other problems like credit assignment.

In this thesis, we hypothesize that EAs, in particular, differential evolution (DE) can deal with non-stationarity better than TD methods. This is because DE does not require the Markov Property or the stationarity assumption to hold for convergence. Furthermore, DE should be more resilient to non-stationarity and changing environments since they have a population of different solutions. As long as the population is diverse enough and the target moves slowly enough the population should be able to chase the optimal target, as hypothesized by Moriarty et al. [12].

Moriarty et al. [12] highlight that credit assignment is handled implicitly by EAs. Because individuals are evaluated after a full episode has been completed, rather than at every timestep. Therefore, the rewards are tied to sequences of several actions rather than individual actions. The selection operators will ensure that the fittest individuals have more offspring in the next generations. In contrast, credit assignment often needs to be handled explicitly by most temporal difference methods. Because the policy is updated after every action based on gradient decent which, in turn, is based on the reward received after performing that action.
We also speculate that the effect of partial observability is also reduced, this is because EAs are evaluated after a full episode. This should give agents more time to explore the environment as they sample the environment multiple times before they are evaluated.

In summary, we hypothesize that EAs can deal with the challenges of MARL environments better than temporal difference-based methods can. In particular, we speculate that non-stationarity, partial observability, and credit assignment will pose less of an issue to AEs than to gradient-based methods.

4 Methodology

In this chapter, we will discuss how we intend to test our hypothesis: EAs can better deal with the challenges posed by MARL than TD methods. In Section 4.1, we discuss the algorithms we want to evaluate to test this hypothesis. In Section 4.2, we will discuss the environments we use as benchmarks to evaluate these algorithms. In Section 5.2, we discuss which architectures were evaluated and which one was ultimately chosen for the experiments. Section 5.3 discusses the optimizations of the hyperparameters for DE on the compressed checkers environment, the resulting configuration is used in all other experiments.

4.1 Algorithms

In this section, we will explain the implementation and application of the algorithms that we have evaluated: VDN, DE, and RS. All of these algorithms rely (partially) on random number generation to explore the search space. That means that all of our experiments are stochastic in nature. Therefore we run five replications of all experiments and average the results, to combat the effects of random noise.

4.1.1 Diferential Evolution

When dealing with DE we need to convert the NNs of all agents into a single floating-point vector that can be used as the genotypes for the algorithm. This can simply be done by flattening the weights matrix of every agent into a floating-point vector, and then concatenating the vectors of all agents into a single vector. Figure 14 shows an example of this for a two-agent game in which both agents have tiny networks, with 6 parameters each. Note that this way of performing DE on a set of NNs is analogous to centralized learning with decentralized execution because all agents are always evaluated together, but they all have their own (not necessarily different) observations and policies and they take actions without communicating to one another.

For these experiments, we use the modular differential evolution (modDE) implementation by Vermetten et al. [39]. This framework is integrated into the larger IOHprofiler framework by Doerr et al. [7]. ModDE is a collection of different algorithms. It allows for a wide range of different operator combinations. For the experiments run in this thesis we chose the rand1bin configuration of modDE. The full list of hyperparameter settings can be found in Appendix A. When trying to optimize the hyperparameters for modDE, we varied the following hyperparameters:

- The mutation factor F.
- The crossover rate CR.
- The upper and lower bounds b.
- The population size Π was varied strictly as a function of the dimensionality d: $\Pi \doteq 5 \cdot d$
- The budget B, in terms of fitness evaluations, was based on the population Π: $B \doteq 1000 \cdot \Pi = 5000 \cdot d$. This way, the budget would allow for 1000 generations.

Various network architectures for the agents were explored in Section 5.2, using the compressed checkers environment (see Section 4.2.2) to benchmark their performance. However, all experiments in the results section 6 were run with the best-performing architecture found in Section 5.2.

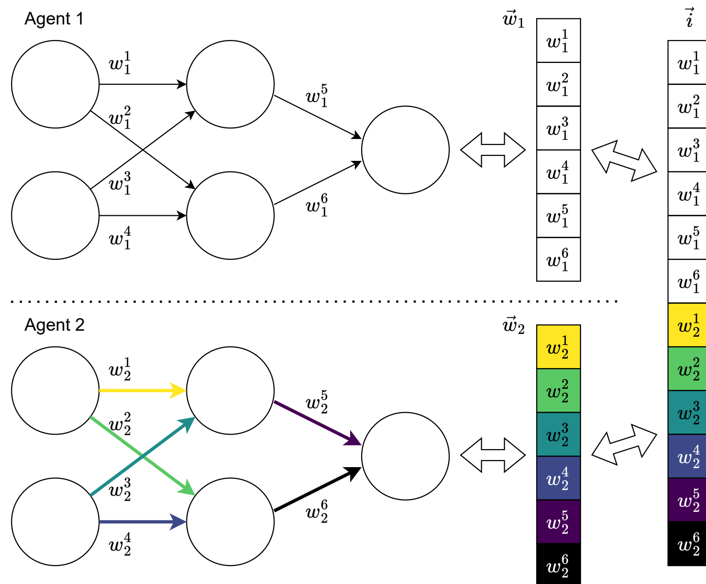

Figure 14: This image shows how two agents can be joined into a single individual for modDE. On the top-left and bottom-left are two small NNs. The weights of these NNs can be stored into a single vector, as happens in the center. For the bottom agent, we have also color-coded the weights and their placement in the vector. On the right, we can see that the two vectors $\vec{w}_1$ and $\vec{w}_2$ are concatenated to form individual $\vec{i}$.

4.1.2 Value Decomposition Networks

As a baseline, we will use the VDN implementation provided by Koul [21]. This implementation is identical to the one used by Mu et al. [30]. However, it does differ from the one implemented by Sunehag et al. [37], described in Section 2.3.3, in three key ways. Firstly, it starts with two fully connected layers rather than one. Secondly, it uses a GRU for its recurrency instead of an LSTM. And finally, it ends with a fully connected layer instead of a dueling layer. Figure 15 illustrates this architecture in the two-agent case.

When training these VDNs we left all hyperparameters at their default value, as decided by Koul [21]. Except for the budget or "max_episodes" parameter. which was varied based on the environment: 100 000 for Checkers and 20 000 for Switch2.

4.1.3 Random Search

,

DE is a stochastic algorithm that searches the search space without using gradients. It evaluates a large number of data points to find the best solution. Because of these two facts, it seems fair to compare DE against a random search algorithm with the same budget, to see how much the more selective search of DE influences the optimization process.

To implement random search we simply generate one NN per agent, and evaluate a set of randomly generated agents by playing the environment based on the generated policies. In EA terms, we generate an individual randomly and evaluate its fitness once. If the fitness is higher than the previous best we update the previous best and save the individual. The NN architectures are kept identical between the RS and modDE searches.

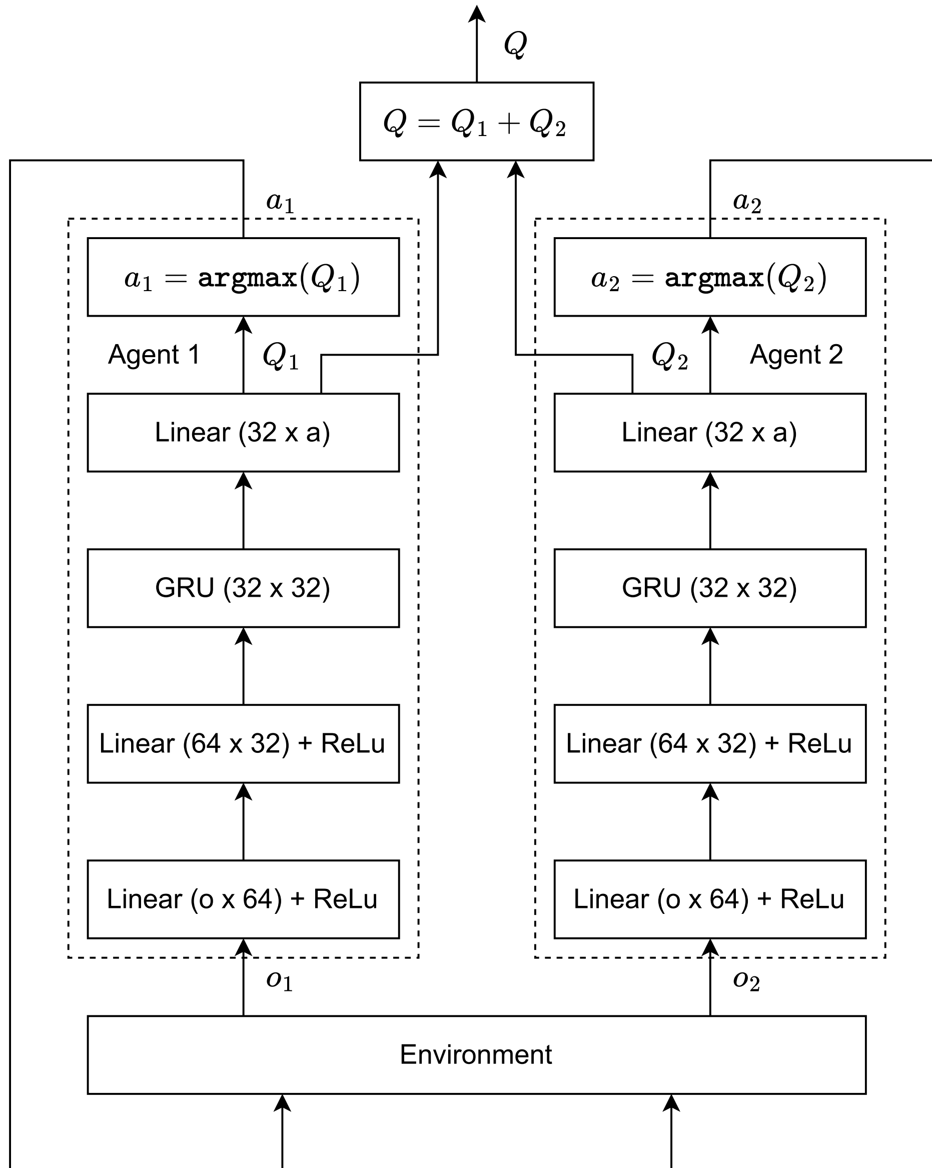

Figure 15: This figure shows the architecture of a VDN for two agents, as implemented by Koul [21]. This figure was inspired by Sunehag et al. [37]. Each agent starts with two fully connected layers with 64 and 32 nodes, respectively, followed by a GRU with 32 nodes and a final hidden layer with a nodes, with a being the number of possible actions and o being the dimensionality of the observation. The actions chosen are based on the individual Q-values of the agents, whilst the training is based on the combined Q-value of both agents.

4.2 Environments

In this section, we will discuss the environments that we use to evaluate the algorithms discussed in Section 4.1. Both (compressed) Checkers and Switch2 are deterministic environments. The policies of our agents are also deterministic (once the agent has been generated). Therefore there is no need to evaluate any given set of agents more than once on these environments.[7]

[7] The search algorithms themselves are still stochastic, so over a full experiment, we still need to run 5 replications.

4.2.1 ma-gym Checkers-v0

Checkers (Checkers-v0) is a partially observable cooperative multi-agent environment. For this thesis, we use the implementation of Checkers developed by Koul [20]. The objective of checkers is to maximize the combined reward of the two agents. The game is played on a grid world, each agent can observe the three-by-three space around itself, as well as their own absolute coordinates. An agent can move down, left, up, or right, or it can stay on its current tile. Each tile on the grid can contain an apple, a lemon, or an agent. A tile can also be empty, but it cannot contain more than one item at a given timestep. If an agent moves to a tile occupied by the other agent the move will be cancelled.[8] If an agent moves to a tile containing an apple it gets a reward and if it moves to a tile containing a lemon it gets a penalty. The rewards are summed up and shared between the agents. Apples and lemons will be removed permanently once an agent occupies their tile. The game ends when all apples have been removed from the game (i.e. when positive rewards can no longer be obtained), or when a predefined maximum number of timesteps has elapsed, this is set to 100 timesteps for all experiments in this thesis. To encourage strategies that are efficient in the number of timesteps a small idle penalty of 0.01 is given to each agent at every timestep.

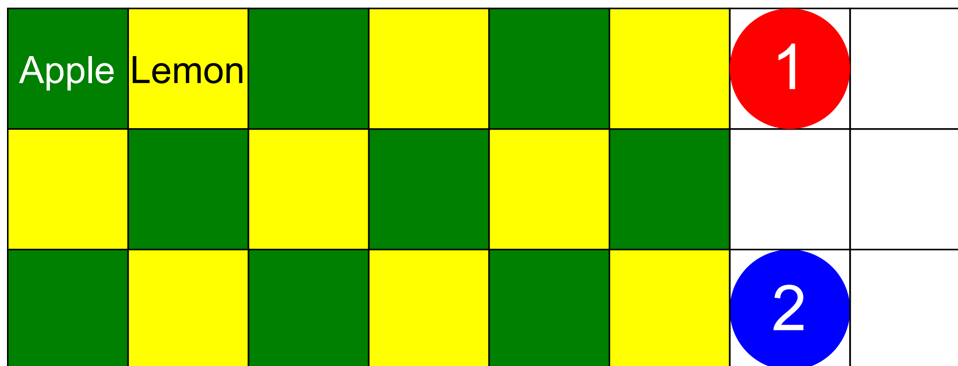

Figure 16: The start position of the default configuration of the Checkers-v0 environment. This environment contains nine apples and nine lemons, laid out in a checkerboard pattern. The red agent is the sensitive agent, which receives ten times the penalties and rewards of the less sensitive agent in blue.

In the initial configuration, the apples and lemons are laid out in a checkerboard pattern, this is where the environment gets its name from. The default configuration of the checkers environment is a three-by-eight grid, in which the first 6 columns are filled with apples and lemons, as demonstrated in Figure 16. Because of the checkerboard pattern of apples and lemons, some of the lemons need to be eaten in order to reach all the apples on the board. In other words, some penalties need to be taken, in order to maximize the total reward.

There are two agents in the checkers environment: one regular, less sensitive agent, which receives rewards of ± 1 for eating apples or lemons, and one sensitive agent, which receives rewards of ± 10 for eating apples or lemons.

Because of the difference in reward scaling, the optimal strategy for checkers would be to have the sensitive agent eat all available apples and have the less sensitive agent as few lemons as possible

[8] Technically, due to the way the game is coded, the regular/second agent could move to the spot occupied by the sensitive/first agent if the sensitive/first agent moves away from that tile during the same turn. But the reverse is not true.

whilst still opening a path for the sensitive agent to eat all the apples. All of this should be executed in as few timesteps as possible due to the idle penalties. In the three-by-eight set up, this would mean that the less sensitive agent would only need to eat the three lemons in the middle row, incurring a penalty of 3. This way the sensitive agent can eat all nine apples generating a reward of 90. This leaves both agents with a total reward of 87, not factoring in the idle penalties at every timestep.

4.2.2 Compressed Checkers

The observation of a single agent in the checkers environment is made up of the coordinates of the agent, followed by the one-hot encoded data of all of the tiles in the three-by-three area around the agent, as depicted in Figure 17, panels 1, 2, 3, 4, and 6. The one-hot encoding of the tiles checks for lemons, apples, agent1, agent2, and walls. And it does this for all 9 tiles. This means that in total, the observation of a single agent is a 47-dimensional vector ($47 = 2 + 9 \cdot 5$).

The observations in the Checkers environment can be simplified/compressed significantly. Firstly, the agent itself will always occupy the tile in the center because it only sees the three-by-three area around itself. This allows us to remove that tile from the observation as it is simply a constant that will not add new information to the observation. Secondly, since the agent will always be in the center of its own observation and never in any other tile, we can merge the one-hot encoding for agent1 and agent2. Thirdly, the "wall" is not present anywhere in the environment, so we can cut it from the observation. Lastly, the agents do not need to know their coordinates to solve the game; they can function perfectly well knowing only what is around them, allowing us to cut two more inputs. In total, this leaves us with an observation with a dimensionality of 24 ($24 = 8 \cdot 3$), if we choose to compress the observations. This compression of the observation, as well as the derivation of the original observation, are visualized in Figure 17, panels 5 and 6.

Compressing the observation like this will not change the core challenge of cooperation between two agents. However, it will significantly reduce the dimensionality of the input by removing redundant information, which makes the task of extracting useful information from this input significantly easier. Therefore we will treat this version of checkers, in which the observations are compressed, as a separate environment, which we will simply refer to as Compressed Checkers.

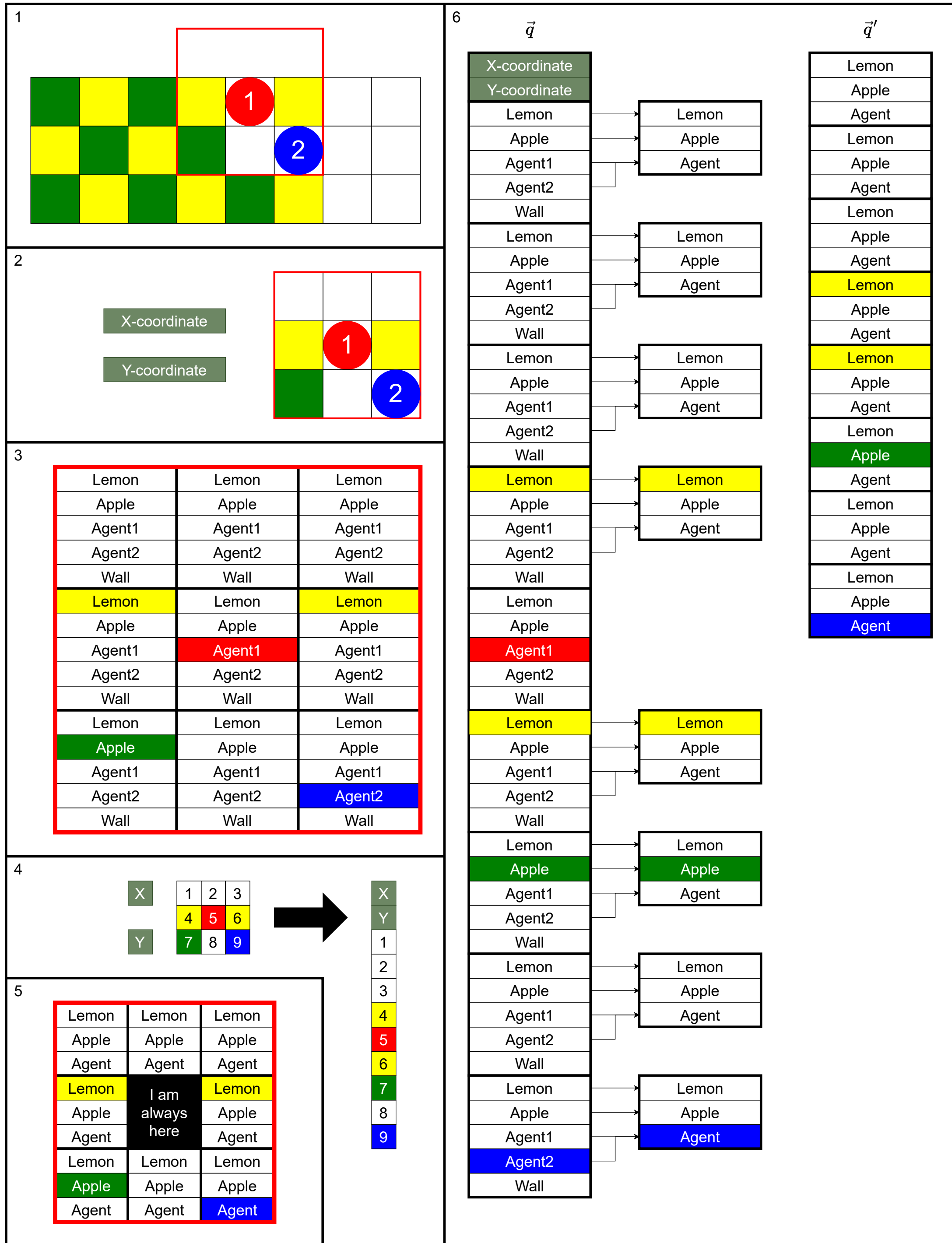

Figure 17: This figure shows the full derivation of the observation vectors for the Checkers-v0 environment. Panel 1 shows the full game state of the checkers environment. The red box highlights the observation of the sensitive agent. Panel 2 shows the full observation for this agent, including the coordinate values. The content of the tiles is one-hot encoded, as displayed in panel 3. The entries that have a value of 1 are highlighted in the respective colors. Panel 4 shows how this encoded observation is assembled into a vector. Not all entries in this vector are necessary, therefore, panel 5 shows which entries should remain after compressing the observation. Finally, panel 6 shows the full observation $\vec{q}$ and the compressed observation $\vec{q}'$, as well as how $\vec{q}'$ is composed from $\vec{q}$.

4.2.3 ma-gym Switch2-v0

Like Checkers, Switch2 (Switch2-v0) is a cooperative multi-agent environment. But unlike Checkers, Switch2 is fully observable. We use the version of Switch2 developed by Koul [20]. In this version of Switch2, there are two agents who need to reach a square on the opposite side of the board. However, there is a single-file corridor in the center of the board that the agents need to pass through and it is not possible for the agents to occupy the same tile, or for them to move through each other. Therefore one agent must learn to reach its target as soon as possible whilst the other agent waits for the first agent to pass through the choke-point, after which it too should rush for its target. The initial configuration for Switch2 is depicted in Figure 18.

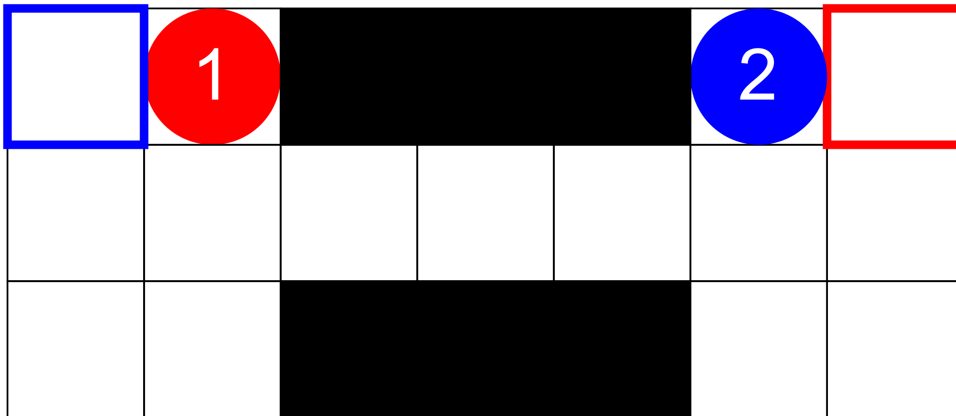

Figure 18: The start position of the default configuration of the Switch2-v0 environment. This environment contains two agents a red agent and a blue agent. The two agents should aim to reach their respective target square in the same color.

Similarly to Checkers, the rewards in Switch2 are shared between the agents, who must learn to cooperate. When an agent reaches its target square it gains a reward of 5. To reward more efficient strategies an agent gets a penalty of -0.1 for every timestep that it is not on its target square. The game ends when both agents have reached their target square, or when a maximum number of 50 timesteps has elapsed.

From the starting square, each agent would need 6 timesteps to reach its target. The starting square is also the square that is closest to the target without blocking the way for the other agent. The second agent can start moving two turns before the first agent reaches its target square. This means that, in total, there are 6 turns in which both agents get a penalty and 5 more turns in which only the second agent incurs a penalty. This means that the total return would be 8.3 in the optimal case.
The observations for the agents are shared: each agent observes its own and the other agent's coordinates. The action space for Switch2 is the same as the one for Checkers: each agent can move up, down, left, or right, or stay on its current tile.

5 Experimental Setup

In this chapter, we will discuss the choices we made for the plots in Section 5.1. We also discuss the search for the best network architectures in Section 5.2. Finally, we discuss the choices for the hyperparameters F, CR, and b in Section 5.3.

5.1 Plot choices

As mentioned in Sections 4.1 we run five replications of each experiment because all three algorithms we run are inherently stochastic. When we plot the results of an experiment we plot the average return in a full line. The standard in most RL papers is to also incorporate the standard deviation into these plots. We have chosen not to do this for our plot. Instead, we plot the maximum and minimum return obtained at each timestep in a dashed line and highlight the area between them in a transparent version of the same color used for the average. We chose this approach over the standard deviation approach because it allows us to better highlight when agents have reached or surpassed specific milestones, more on these milestones later.

We have chosen to color-code the plots as much as possible. In the final plots of the results section, we have colored modDE in yellow. For figures containing more than one version of modDE, we have reserved yellow for the version of modDE that most closely resembles the version used as the baseline for modDE in the results section 6. We have also used two alternative colors that we use when multiple versions of modDE are plotted in the same figure, these are lime and blue. RS and VDN are colored in purple and teal respectively, these two colors are only reserved for RS and VDN, respectively. The choice of these colors is based on the Viridis colormap form matplotlib, by hunter [17].

In dotted lines, we have plotted three different milestones that we have identified for the (Compressed) Checkers environment, two of which are also applicable for the Switch2 environment. The first one: "Single-agent high score" is highlighted in cyan. It represents the highest possible score that can be obtained when only one of the agents moves. Any team of agents that reaches a higher return must have worked together to achieve this, making it an interesting milestone to test if a team suffers from the lazy agent problem or not. The second milestone: "VDN score reported by [30]" is highlighted in teal. It represents the score for VDN on the Checkers environment, as reported by Mu et al. [30]. In our own experiments, we were unable to replicate their results. So for completeness, we have plotted it as a milestone for modDE to reach. Naturally, we only show this line for plots about the Checkers and Compressed Checkers environments, as we did reach an optimal score with VDN on Switch2. The final milestone is the "Theoretically optimal score", which we have highlighted **black**. This is the optimal score we found for the environment, based on the strategy used by the supervised approach, more on this in Section 5.2.1. It provides a good estimate of an upper bound for the return that can be reached in the Checkers environment.

5.2 Network Architectures

Before we can tune any other hyperparameters we need to find out which network architectures can play the environments and which ones can be trained using differential evolution. To test the suitability of the network architectures we will use the compressed version of the checkers environment. We chose this environment because it provides a nice balance between keeping the main challenges of checkers whilst

allowing for smaller networks with a lower dimensionality than the original checkers implementation. This allows for shorter training times because the budget of our modDE experiments is tied to the dimensionality of the problem. Due to time restrictions, we assume that the optimal configuration for the compressed Checkers environment carries over to the Switch2 and Checkers environments.

5.2.1 Hidden Layers

In the early stages of experimentation, we explored various types of NN architectures. Three of these architectures are worth discussing in the context of this thesis, they are highlighted in Figure 19. In this figure, agent type A is the smallest possible network that can still process all the inputs and outputs required for the Compressed Checkers environment. Agent type B has a hidden layer as well as an activation function, which allows it to solve non-linear problems. Finally, Agent type C also has an LSTM layer that allows the agent to "remember" past observations.

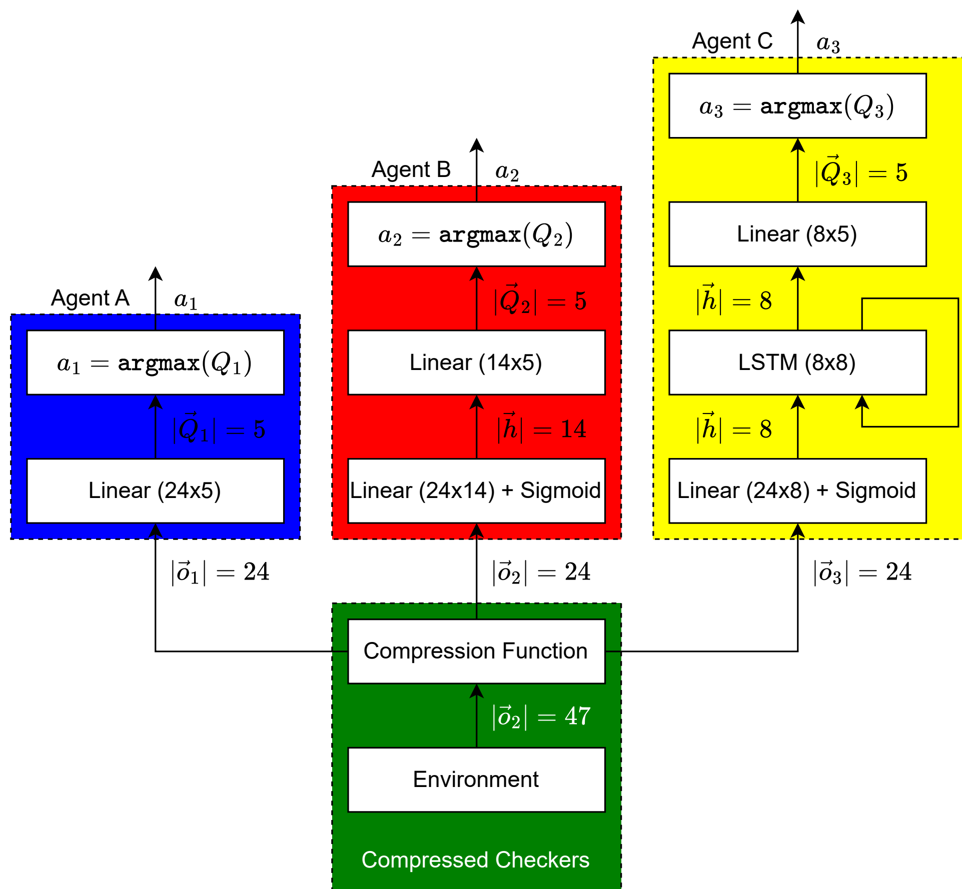

Figure 19: This figure highlights the Compressed Checkers environment and the three types of agents that were evaluated for this thesis. Agent A, on the left, is the smallest and simplest of these agents. It is a simple single-layer NN that directly connects the inputs to the 5 possible actions. Agent B, in the center, is a larger version of Agent A with a sigmoid activation function after the input layer and an extra hidden layer with 14 nodes. Finally, Agent C, on the right, has an input layer with the sigmoid activation followed by an LSTM layer and an output layer.

For the earliest proof of concept experiments with DE, we used networks that are comparable to that of agent B from Figure 19. This is because we assumed that this was the simplest viable architecture.

However, almost all of these yielded solutions that ended up getting stuck in local optima in which the sensitive agent solved the environment on its own.

At first, we thought that this was because the agents of type B were not sophisticated or expressive enough to learn to cooperate. This led to the development of agent type C, which was inspired by the architecture of Sunehag et al. [37]. Agent type C incorporates an LSTM into its architecture which allows agents to "remember" past states. Unfortunately, agents of type C have a significantly higher number of trainable weights, which is directly proportional to the dimensionality of the search space for DE. Therefore, the runtime for DE is significantly longer when training agents of type C. This means that we did not have enough time to run a full experiment for agents of type C.

A supervised approach To test whether or not a network architecture is expressive enough to represent a policy for Checkers that cooperates with other agents we ran a supervised learning experiment. For this experiment, we first devised the optimal strategy for both agents in Checkers. We wrote this strategy as a list of actions for each agent at each timestep. Then we ran a demo in which the actions were played according to this strategy to record the observations. This allows us to verify that each observation always has the same action if it is repeated because agents of type A and B cannot remember information about previous states. In short, it checks if the Markov property holds if the problem was resolved by a centralized controller. The observations recorded are fed as inputs to a supervised learning model. The actions that the optimal strategy would pick will be used as the ground-truth labels for these networks.

After training various architectures using this supervised learning approach, we concluded that agents of type B and type A were both able to learn the optimal policy in a supervised learning setting. The next step would be to test if agents of type A could be trained using modDE instead of supervised learning.

Architecture results Figure 20 compares the training progression of an agent of type A with that of an agent of type B. To make a fair comparison between the two agent types, we ran a reproduction of the earlier type B agent runs with the same values for b, F, and CR as the final experiments. We also ran both networks without biases for the same reason.

From Figure 20 we conclude that the smaller type A networks train significantly faster than the larger type B networks. The return of the type A networks is also significantly higher than that of the type B networks. In fact, the type B networks gets stuck in a locally optimal solution in which only the sensitive agent moves; it clears a path for itself by eating the minimal amount of lemons required to eat all the apples, in the fewest amount of timesteps. This solution is a local optimum because any small change from one agent would lead to a worse score; both agents need to change their behavior to improve this solution. However, there is one run with the type B networks that escaped this locally optimal solution near the end of its training time. This means that it is plausible that the type B networks could eventually reach the same score as the type A networks, albeit with a significantly longer training time. Even if the type B networks could match the type A networks given enough training time, the type A networks still train significantly faster and do not seem to get stuck in local optima as easily. Therefore, we have decided to optimize the type A networks further and compare it against the other methods.

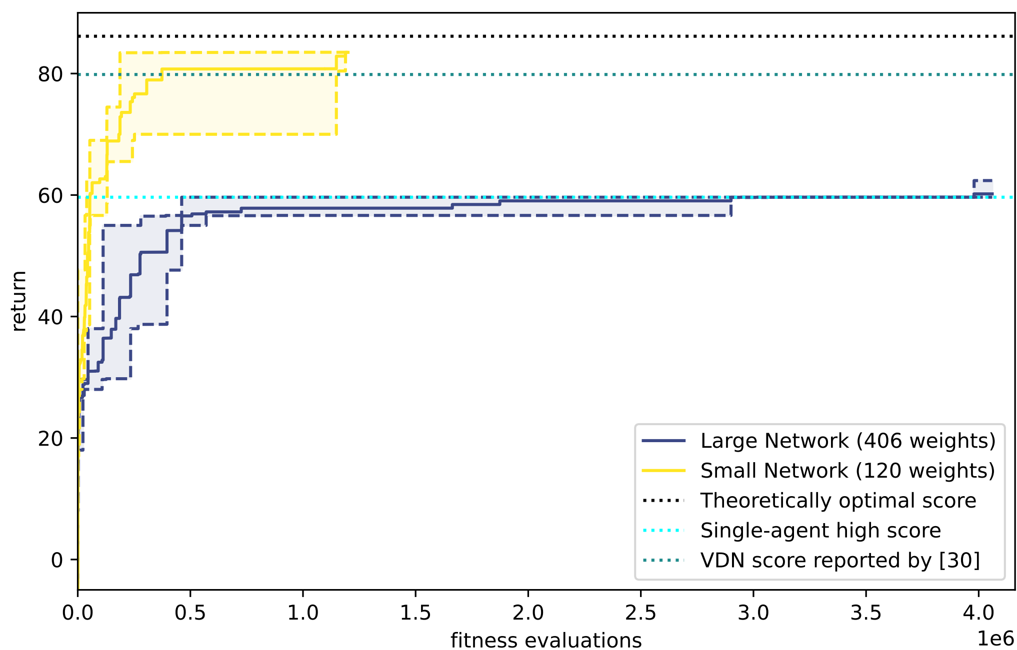

Figure 20: This figure shows the average return plotted against the training time in fitness evaluations for two different configurations of modDE, run on the Compressed Checkers environment. **The architectures** of the networks were varied for these two experiments. The larger network with a hidden layer is highlighted in blue. Whereas the smaller network without hidden layer is highlighted in yellow. The replications with the highest and lowest returns are also plotted in dashed lines. The dotted lines represent the milestones that can be reached on this environment.

5.2.2 Bias

The first set of agents that could successfully solve the Compressed Checkers environment had two agents of type A, with both of them still having bias nodes. Sticking with the theme of reducing dimensionality as much as possible we wanted to see if two agents of type A without bias nodes could still solve this environment. To evaluate this, we compared the training progression of a run with agents with bias nodes to one with agents without bias nodes. The resulting training progressions are displayed in Figure 21.

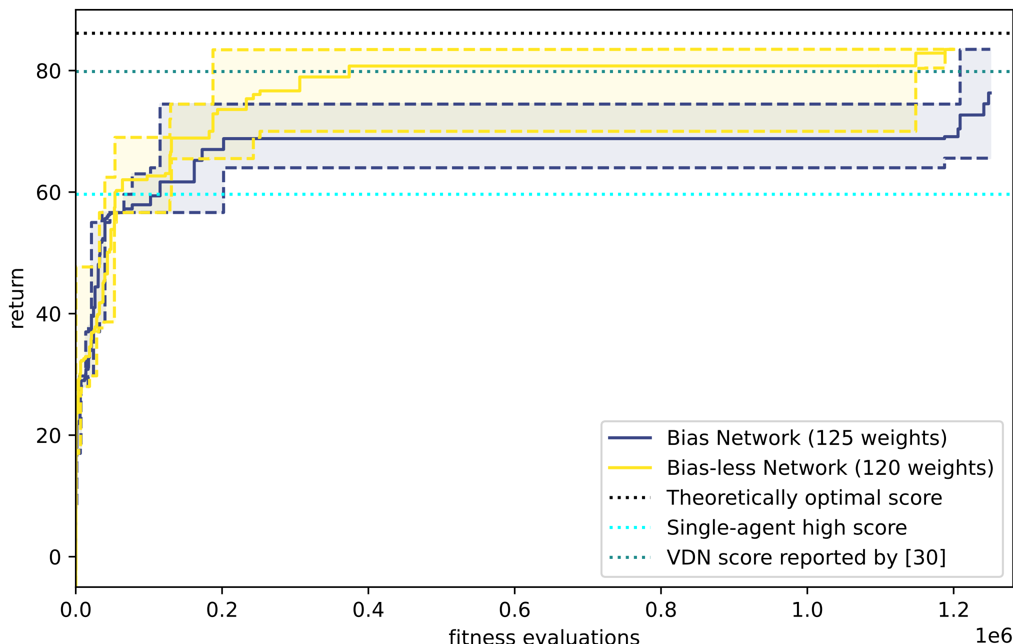

Figure 21: This figure shows the average return plotted against the training time in fitness evaluations for two different configurations of modDE, run on the Compressed Checkers environment. **The use of bias nodes** in these networks is differnet for these two experiments. The network with bias is highlighted in blue. Whereas the network without bias is highlighted in yellow. The replications with the highest and lowest returns are also plotted in dashed lines. The dotted lines represent the milestones that can be reached on this environment.

From Figure 21 we conclude that including bias nodes in our networks reduces performance slightly, this is likely because adding the bias nodes increases the dimensionality slightly. Therefore, we have decided to omit bias nodes from all networks in our future experiments. Besides the higher return, this allows for slightly shorter training times, as the budget and population size are also tied to the dimensionality.

5.3 Hyper parameter optimization

We ran our first baseline experiments with the default hyperparameters for modDE $F = CR = 0.5$, with all other hyperparameter settings matching those in Appendix A. However, it was unlikely that these parameters were the optimal setting for the given environment. Therefore, we chose to run a small hyperparameter sweep on the Checkers environment to find the optimal settings for the F and CR

hyperparameters as well as the boundaries of the problem.

5.3.1 Boundaries

As mentioned in Section 4.1.1 the weights of all neural networks need to be flattened into a single vector. This means that the search space for an N agent game with d weights per agent is am $N \cdot d$ dimensional vector-space $\mathbb{R}^{N \cdot d}$. To limit this infinite space to a more reasonably sized search space we introduce the boundary b. We constraint the weights of the network to be in the range $[-b, b]$. To find out which boundaries are optimal we ran three experiments in which we only varied b. F and CR were both set to 0.5 for this experiment since that was the default configuration of modDE.

The results of this experiment can be seen in Figure 22. The values of $b = 1$, $b = 7$, and $b = 10$ were chosen for this experiment. $b = 7$ was chosen based on the supervised experiment which yielded a network with weights $\theta_i \in [-7, 7]$. $b = 10$ was chosen because the resulting modDE run had several weights which were exactly ± 7, which suggested that there could be a better solution if the bounds were increased. Finally, $b = 1$ was chosen because the result from the $b = 10$ experiment showed no significant improvement over the $b = 7$ experiment. Thus we wanted to know if there was a solution to the problem within a smaller search space.

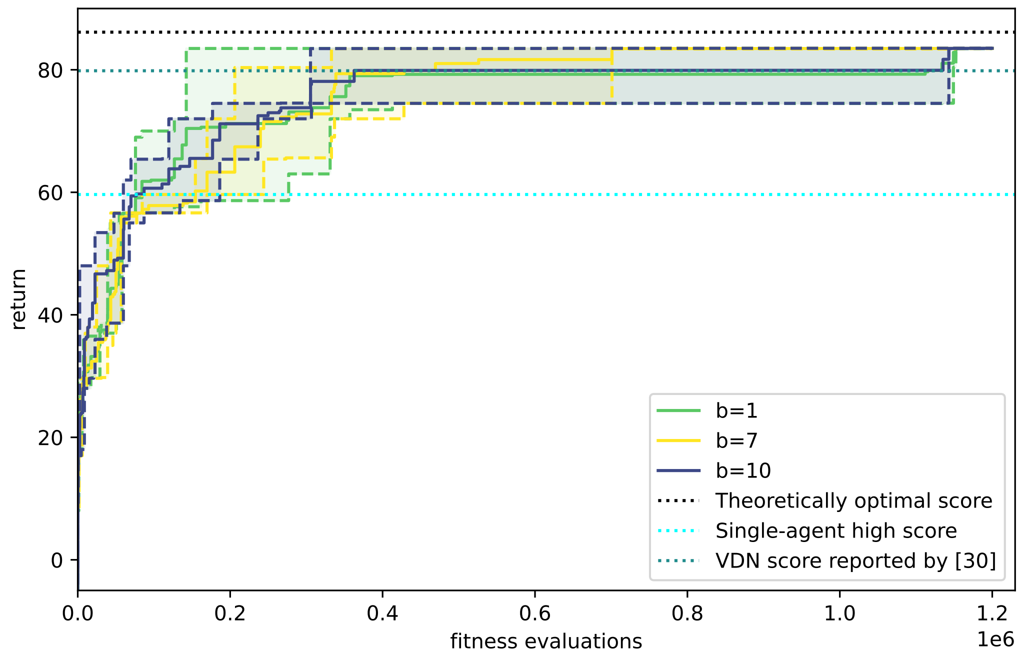

Figure 22: This figure shows the average return plotted against the training time in fitness evaluations for three different configurations of modDE, run on the Compressed Checkers environment. **The boundary size** b is varied for these three different experiments. The replications with the highest and lowest returns are also plotted in dashed lines. The dotted lines represent the milestones that can be reached on this environment.

From Figure 22 we can see that there does not seem to be a significant difference in the return for the

various settings of the boundaries. This is likely because the search space is very rugged, meaning there are also many different (locally) optimal solutions spread across this search space. Because the return does not seem to depend much on the size of the bounds, we have decided to keep $b = 7$ constant for the remaining experiments.

5.3.2 Hyperparameters for modDE

To find the best values for F and CR we wanted to test values for F and CR in the ranges [0.5, 0.9] and [0.8, 1], as suggested by Storn and Rainer [35]. However, due to time constraints only the combinations only the combinations of $F = 0.7, CR = 0.9$ and $F = 0.9, CR = 1$ have been tested in this range, as well as the default run with $F = 0.5, CR = 0.5$. The results of these experiments can be seen in Figure 23.

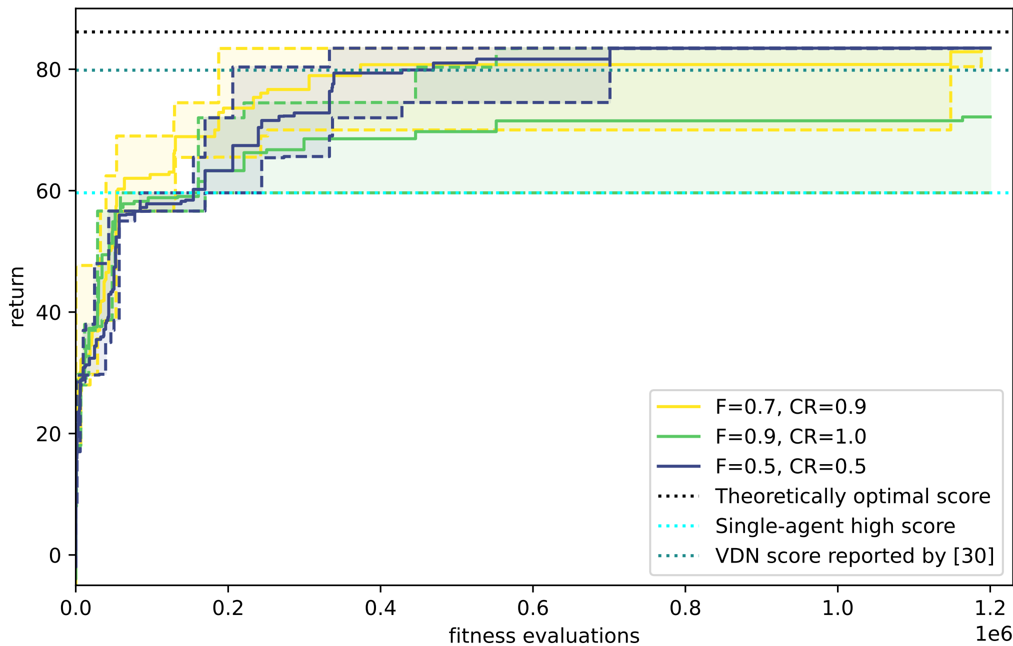

Figure 23: This figure shows the average return plotted against the training time in fitness evaluations for three different configurations of modDE, run on the Compressed Checkers environment. **The hyperparameters** F **and** CR were varied for these three different experiments. The replications with the highest and lowest returns are also plotted in dashed lines. The dotted lines represent the milestones that can be reached on this environment.

From Figure 23 we can see that there is some variation in performance between the different hyperparameter configurations. However, there did not seem to be enough difference between the performance of the $(F, CR) = (0.7, 0.9)$ and $(F, CR) = (0.5, 0.5)$ configurations. Especially considering that the only difference in average return is caused by a single run that converged later. We did not perform an extensive search for F and CR in the ranges [0.5, 0.9] and [0.8, 1], respectively. However, these experiments are quite time-consuming, as can be seen in Appendix B. Therefore we have decided not to continue the search for the optimal hyperparameter configuration, as the expected improvement is minimal compared to the time required to find the optimal setting. For all further experiments, we have decided to use $F = 0.7$ and $CR = 0.9$, as suggested by Storn and Rainer [35].

6 Results

In this chapter, we will discuss the results of our experiments. We will evaluate the agents with the smallest possible architectures, as discussed in Section 5.2. We will search for the optimal policies for them using modDE and RS, as discussed in Sections 4.1.1 and 4.1.3, respectively. We will compare the results of modDE against that of RS and that of VDN, as discussed in Section 4.1.2. We will evaluate these algorithms on three environments: Checkers in Section 6.1, Compressed Checkers in Section 6.2, and Switch2 in Section 6.3. The full list of hyperparameters used for modDE can be found in Appendix A, with the justification for the choices of bounds, F, and CR explained in Section 5.3.

As discussed in Section 5.1, we plot the average over five replications in a solid line, as well as the maximum and minimum return over those replications in a dashed line. The dotted lines in our figures represent the milestones that can be reached by our algorithms.

6.1 Checkers

Figure 24 shows the progression of the highest obtained return plotted against the number of fitness evaluations it took to achieve this return for the first time. The performance of modDE is plotted in yellow, RS is plotted in purple, and VDN is plotted in teal. The dotted lines for VDN were omitted from this figure, because they would clutter the figure too much to be legible. However, they are still highlighted by the lightly colored areas.

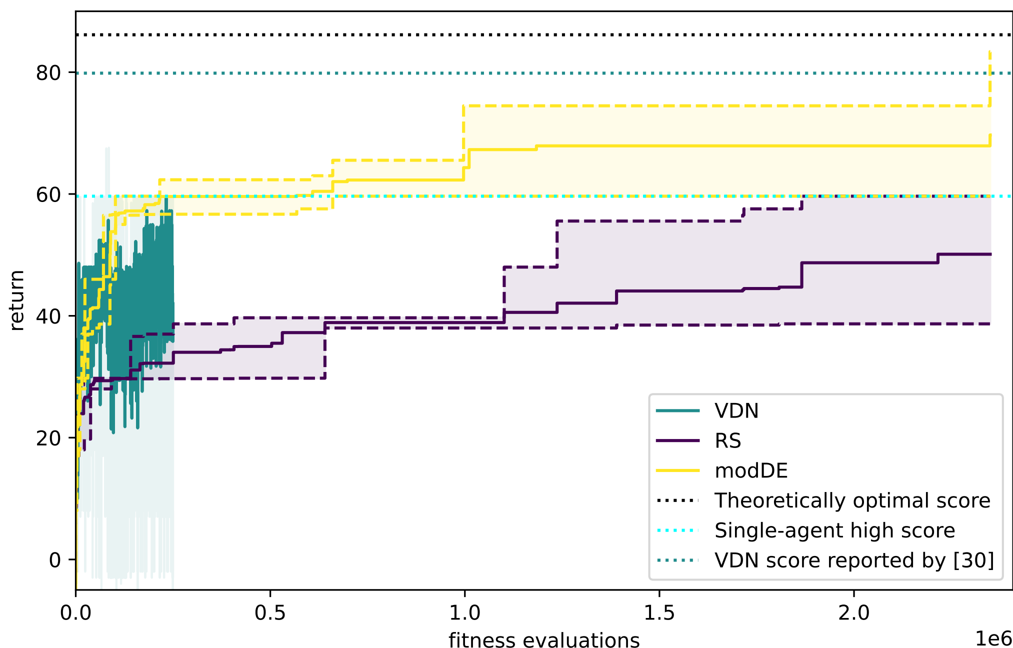

Figure 24: This figure shows the average return plotted against the training time in fitness evaluations modDE, VDN, and RS run on the Checkers environment. The replications with the highest and lowest returns are also plotted in dashed lines. The dotted lines represent the milestones that can be reached on this environment.

From Figure 24 we can see that the average modDE runs can learn to beat the single agent high scores after about 700 000 fitness evaluations. We can also see that there are still one or two runs that got stuck in the local optimum of the best single-agent solution. Near the end of the training, there was even one run that outperformed the VDN score as reported by Mu et al. [30]. We can also observe that the highest RS runs had an identical score to the worst modDE runs after approximately 1 800 000 fitness evaluations. In this figure, it is hard to see the results of our replication of VDN due to its significantly smaller budget at 100 000 fitness evaluations. To get a better overview of this run, as well as how it compares to modDE and RS, we have zoomed in to the combined training time in Figure 25.

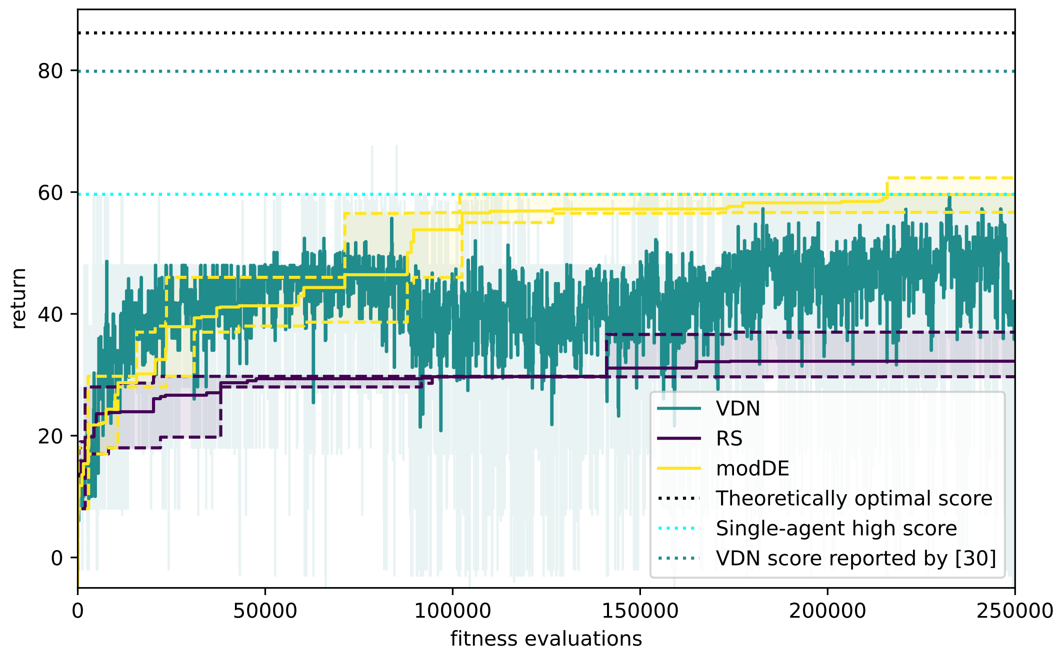

Figure 25: This figure shows the average return plotted against the training time in fitness evaluations modDE, VDN, and RS run on the Checkers environment. The replications with the highest and lowest returns are also plotted in dashed lines. The dotted lines represent the milestones that can be reached on this environment. This figure is a zoomed-in version of Figure 25.

From Figure 25 we can see that VDN performs similarly to modDE, there are even a few instances of VDN slightly outperforming the single-agent high score, albeit very briefly. Both instances occurred in the same replication, and in both cases, the return decreased to its level below the single agent optimal solution after a single TD update. This also highlights a difference in approach between VDN and modDE. Namely, the fact that survivor selection in modDE is elitist. This means that the best return modDE can achieve will never decrease in future generations, as the best solution is always kept. We can trivially implement a form of elitism in VDN as well, by simply keeping track of the best networks found so far. We have plotted the results for VDN if it was an elitist algorithm in Figure 30 in Appendix C.

6.2 Compressed Checkers

Compressed Checkers is functionally the same environment as Checkers, though with a simplified observation function. Therefore, we will still highlight the same milestones as for Checkers. Those being the single-agent high score, VDN score as reported by [30], and the **Theoretically optimal score**. We have not run VDN on this version of Checkers.And it would not be fair to compare VDN as trained on the normal Checkers environment to modDE trained on Compressed Checkers. Figure 26 shows the training progression of modDE and RS on the Compressed Checkers environment.

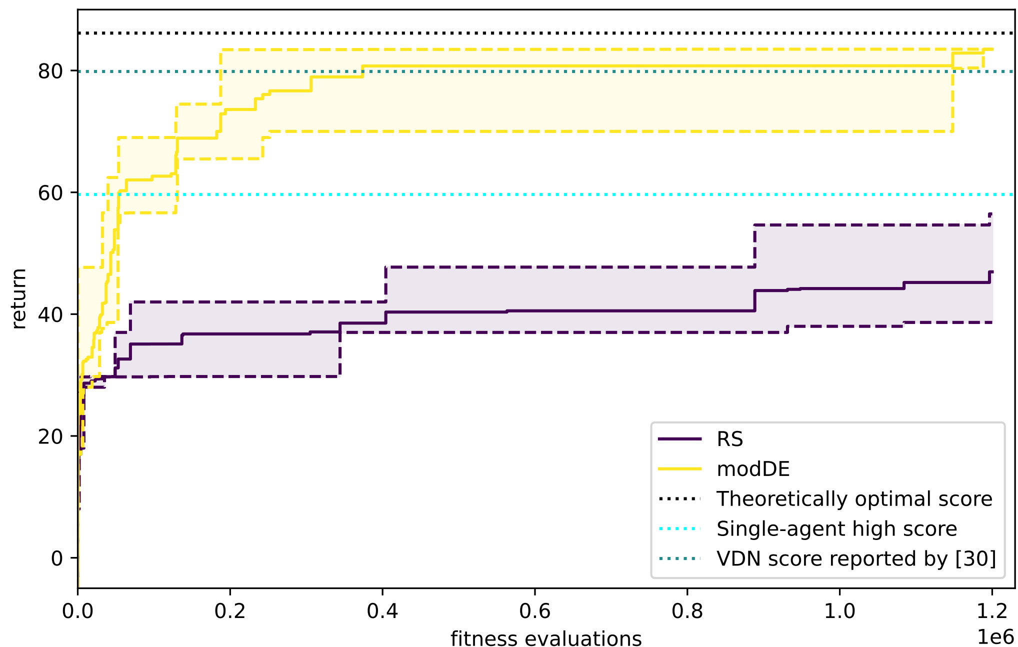

Figure 26: This figure shows the average return plotted against the training time in fitness evaluations modDE and RS run on the Compressed Checkers environment. The replications with the highest and lowest returns are also plotted in dashed lines. The dotted lines represent the milestones that can be reached on this environment.

From Figure 26 we can see that modDE can outperform VDN on the Compressed Checkers. It even reaches close to the optimal score. However, this VDN was trained on the regular Checkers environment, which means that this is not entirely a fair comparison. Furthermore, we do not know how many fitness evaluations it took for VDN to reach this reward, making it even harder to make a fair comparison. We can also note that it seems like the agents training with modDE do not get stuck in the single-agent local optimum for very long. Compared to RS, modDE has a strictly better performance, as there is no overlap between the best and worst solutions, as was the case for Checkers. Interestingly, RS was not able to reach the single-agent high score for the Compressed Checkers environment, highlighting the difference in performance between RS and modDE even further.

6.3 Switch2

Figure 27 shows the progression of the highest obtained return, plotted against the number of fitness evaluations it took to achieve this return for the first time, on the Switch2 environment. VDN, modDE, and RS were evaluated on this environment. Unlike the plots for Checkers and Compressed Checkers, there were only two milestones plotted for in this figure: the Single-agent high score and the **Theoretically optimal score**. Even though the total budget for modDE and RS on the Switch2 environment was 200 000 fitness evaluations, there were no changes in the return for any set of agents over the last 90% of the training time. Therefore, we have decided to cut this last of the plot in order to make the earlier part of the plot more readable.

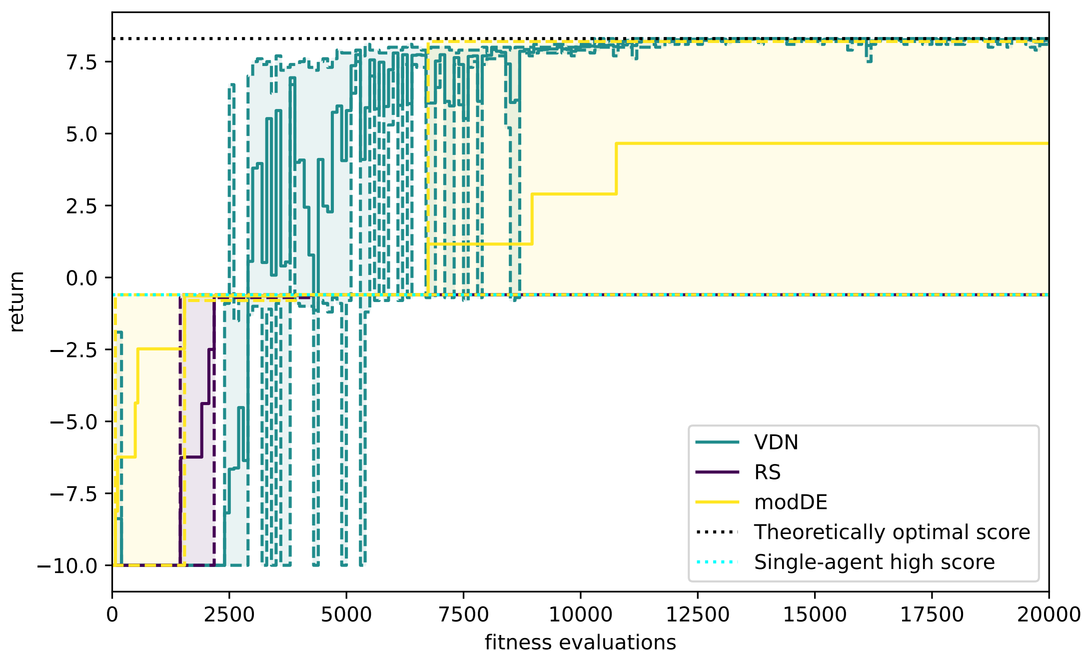

Figure 27: This figure shows the average return plotted against the training time in fitness evaluations modDE, VDN, and RS run on the Switch2 environment. The replications with the highest and lowest returns are also plotted in dashed lines. The dotted lines represent the milestones that can be reached on this environment.

From Figure 27 we can see that VDN outperforms modDE on the Switch environment. In this environment, VDN converges to the optimal solution after approximately 8000 fitness evaluations. Interestingly, all RS runs converged to the best single-agent solution, in which one agent reached its target and the other one did not, after approximately 2000 fitness evaluations. ModDE has the potential to be as good as VDN on the switch2 environment, as three out of five runs of the modDE algorithms were able to reach the same score as VDN. However, two out of five runs were unable to escape the single-agent local optimum, and thus performed as poorly as RS. Analogously to training VDN on the Checkers environment, we have plotted the results for an elitist version of VDN, which keeps track of the best networks found so far, in Figure 29 in Appendix C.

7 Discussion

In this Chapter, we will interpret the results of our experiments and highlight the strengths and weaknesses of our method. In Section 7.1 we discuss the overall performance of modDE on the Checkers and Switch2 environments. In Section 7.2 we discuss the impact of dimensionality on the performance of modDE. In Section 7.3 we highlight locally optimal solutions that our method found and what role credit assignment and the lazy agent problem played in those local optima.

7.1 Overal performance

Based on the results in Chapter 6 we can conclude that modDE can find solutions that can solve the Switch2 and (Compressed) Checkers environments, although not all runs did solve their respective environment. This means that modDE can overcome the issues of non-stationarity and partial observability in MARL problems. One key observation is that the size of the networks used when training via modDE is significantly smaller than the size of VDNs, while still resulting in comparable performance.

7.1.1 Switch2

Based on the results in Section 6.3, the case for the Switch2 environment is clear, VDN performs better on this environment than modDE. One interesting observation about the performance of modDE and RS on Switch2, which is not immediately obvious from the plot in Figure 27, is that the solutions recorded by these methods are all part of three categories:

- Neither agent reaches its target.
- One agent reaches its target in the shortest number of timesteps, whilst the other does not.
- Both agents reach their target in the shortest number of timesteps.[9]

It is interesting to note that the first solution is one of many global minima in the search space, the second one is one of many local maxima, and the third one is one of several global optima in the search space, more about this in Section 7.3.

This also means that the lower average return for modDE compared to VDN on switch2 is caused by the fact that only three out of five agents were able to match the score for VDN, while the other two were stuck in the single agent local optimum. Interestingly, all five RS runs were also able to reach this return. One way to deal with modDE getting stuck in a local optimum like this is to simply restart the algorithm with a fresh initial population. As explained by Luke [25], it may in this case be better to perform several shorter runs, rather than one long run. In terms of sample efficiency, we can also see that VDN requires slightly fewer evaluations to reach the return it did. However, the run time required for VDN is longer than that for modDE, as can be seen in Appendix B.

7.1.2 Checkers and Compressed Checkers

ModDE can reach higher scores on Checkers than those reported by Mu et al. [30]. However, this only happened in one of the five replications, near the very end of the training cycle (when the budget had nearly expired). Moreover, one of modDE's replications failed to escape the single-agent local optimum. This means that, on the Checkers environment, the average return for modDE is lower than that of

[9] Depending on which agent moves first this can vary by 1 timestep.

VDN as reported by Mu et al. [30].

When training modDE on the Compressed Checkers environment we see that it performs even better than VDN. All of the five replications managed to reach a better return than VDN as reported by Mu et al. [30]. However, it must be stated that this environment removes a lot of redundancies in the observations. This does make the comparison somewhat unfair, as VDN was not run on Compressed Checkers but on Checkers.

We have not been able to replicate the score for VDN as reported by Mu et al. [30]. This makes it difficult to make a fair comparison of the sample efficiency for VDN compared to modDE. What is interesting to note is that VDN used approximately twice as much time to sample only a fraction of the samples that modDE sampled. This means that we can conclude that modDE is more efficient in terms of total run time.

7.2 Dimensionality

Based on our experiments in Sections 5 we have found that reducing the size of the network had the biggest impact on the performance of modDE. In fact, all networks that could be successfully trained by modDE were linear. This is likely because the dimensionality of the search space for modDE is directly proportional to the size of the network. Namely, it is equal to the sum of the sizes of the neural networks of all agents it has to train. This is consistent with the results found by Wong et al. [42], who found that evolutionary strategies are more sample-efficient than gradient-based methods on low-dimensional, single-agent RL problems, with the reverse being true on high-dimensional, single-agent RL problems. They also found that evolutionary strategies (ES) could find linear solutions to single-agent RL problems, unlike gradient-based methods.

We found that modDE can train significantly smaller networks than have previously been found. For reference, the networks we trained using modDE had 20, 235, and 120 trainable parameters for the Switch2, Checkers, and Compressed Checkers environments, respectively. Compared to the 8 773 trainable parameters VDN had for Switch2 and the 11 635 for Checkers, VDN had approximately 440 times as many trainable parameters for Switch2 and 50 times as many for Checkers. This comparison should come with the disclaimer that we did not try to run VDN, or a gradient-based algorithm comparable to VDN, with a smaller architecture. Therefore, we can not rule out that VDN can solve the problem with a smaller network size. However, as stated by Wong et al. [42], it can be difficult for gradient-based methods to find linear policies in the multimodal search space of RL problems. This challenge for gradient-based methods likely transfers to MARL settings as well.

We can highlight the problem of dimensionality even further by plotting the results on Compressed Checkers, in the same figure as those on regular Checkers, as we do in Figure 28. Remember that the agents learning Compressed Checkers have 120 trainable parameters each, and those learning Checkers have 235 trainable parameters. This means that the search space for Compressed Checkers has 240 dimensions and the search space for Checkers has 470 dimensions. This comparison between Checkers and Compressed Checkers is not fair in terms of algorithm performance because the environments are different. However, the only key difference in this environment is the complexity and dimensionality of the observations. This allows us to reduce the size of the networks of the training agents. The smaller observations of the Compressed Checkers environment should be easier for an agent to analyze, but the key challenges of cooperation and navigation should remain unchanged. Therefore, compar-

ing these two results should give a good indication of the effect the dimensionality has on the performance.

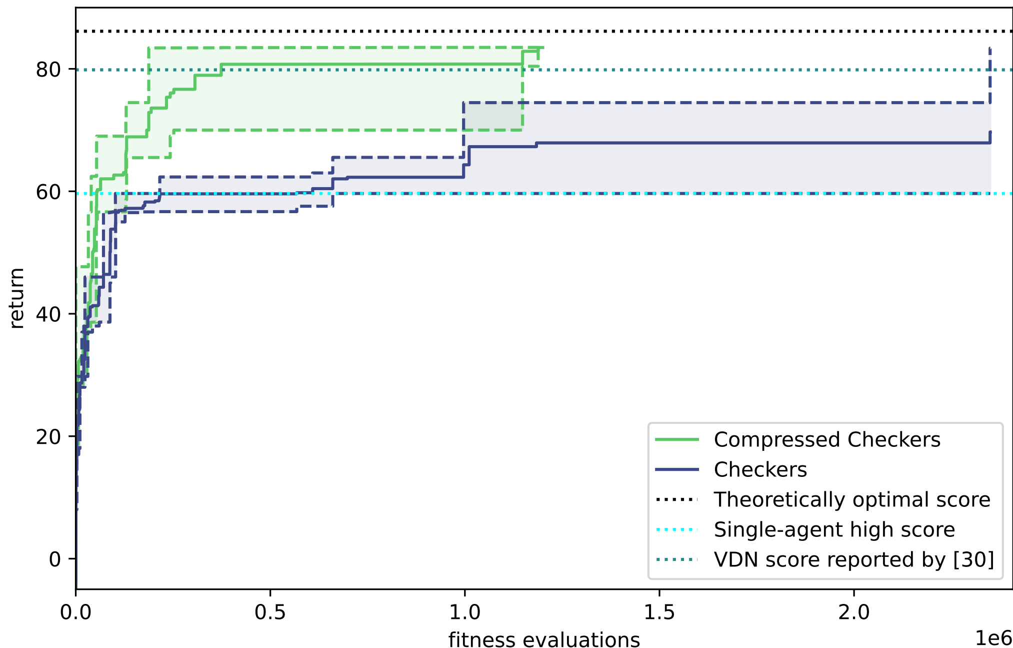

Figure 28: This figure shows the average return plotted against the training time in fitness evaluations for modDE. One replication was run on the lime environment while the other was run on the blue environment. The replications with the highest and lowest returns are also plotted in dashed lines. The dotted lines represent the milestones that can be reached on this environment.

From Figure 28 we can see that the agent trained on the Compressed Checkers environment learns significantly faster than the one trained on the normal Checkers environment. The return obtained is also lower for four out of five runs on the Checkers environment. The remaining run did have a return matching the agents trained on Checkers. So it seems reasonable to assume that agents training on Checkers can reach a similar as agents training on Compressed Checkers if given significantly more training time.

It seems fair to conclude that DE scales poorly with the dimensionality of the problem. This also means that modDE scales poorly with the number and complexity of agents. As the dimensionality of the search space scales linearly with the number of agents. This can be a serious issue for real-world applications, which can often have many agents. Meaningful real-world MARL problems could also have more complicated environments than Checkers and Switch2. For example, the convolutional neural networks (CNNs) used by Krizhevsky et al. [23] to classify images from the ImageNet dataset by Deng et al. [5] have approximately 60 million trainable parameters in their network. The input size for these networks is $224 \times 224 \times 3 = 150,528$ dimensional. This would be too large to handle directly for modDE, even if the networks were linear. Especially since we found that training two networks with 406 parameters each, in an 812-dimensional search space was already a challenge for modDE, as can be seen in Figure 20 in Section 5.2.

Therefore it can be critical to reduce the dimensionality as much as possible. For example, it could

be beneficial to first reduce the dimensionality of observations or input data by using a Variational Auto-encoder by Kingma and Welling [19]. ModDE can then be applied to NNs which take the lower-dimensional latent representation of the observations as input, rather than the full observations. Model-based RL methods can also be used to simplify the observation space for EA methods like modDE.

7.3 Local Optima

One of the problems that we encountered was the lazy agent problem, in which one of the agents does not contribute to the solution. Both agents trained on Switch2 and (Compressed) Checkers suffered from this issue. The optimal single-agent approach does differ between Switch2 and Checkers.
For the (Compressed) Checkers environment this means that the sensitive agent completes the environment on its own. Compared to the two-agent solution this means that the total penalty that needs to be taken increased from 3 to 30. Lowering the maximum total reward from 87 to 60, before factoring in the idle penalties. The fastest route for the sensitive agent to complete this game takes 18 timesteps to complete. Since both agents get a penalty of -0.01 at every timestep we need to subtract 0.36 from the maximal reward to reach the highest possible score that can be obtained by only moving the sensitive agent. This is why we have plotted the milestone for the single-agent high score at 59.64 in all of our plots for the (Compressed) Checkers environments.

This solution to the problem still has a decent reward. However, it is a clear example of a solution that is in a local optimum since a slight change in behavior from one agent will result in a lower score. Only by changing the policy of both agents can the score be increased.

For the Switch2 environment, this meant that one of the agents found its target whilst the other one did not. The Switch2 environment does not terminate if only one agent reaches its target. Therefore, the game will end after the maximum number of timesteps has elapsed; 50 in this case. An agent can reach its target in 6 timesteps from its starting position. Therefore, a total of 56 idle penalties of -0.1 are incurred by both agents in the optimal single-agent strategy. Adding the one-time reward of +5 to this yields a single-agent high score of -0.6.

8 Conclusions and Future Research

In this thesis, we hypothesized that EAs can provide an alternative to gradient-based methods in MARL settings. We identified four key challenges in MARL settings: non-stationarity (1), credit assignment (2), partial observability (3), and computational complexity (4). To test our hypothesis we have evaluated a gradient-based method: VDN and an EA: modDE against a baseline: RS on three environments. These environments are Switch2 and Checkers from ma-gym by Koul [20], and a modified version of the same Checkers environment we called Compressed Checkers.

To answer our research question: **Is differential evolution (DE) better suited to deal with the challenges of MARL problems than gradient-based RL methods?** Yes, we found that modDE was able to find near-optimal solutions to all three environments. However, convergence is not guaranteed for a single run on any of the environments. Because of this, we conclude that non-stationarity (1) and partial observability (3) are challenges that can be overcome by using modDE. Credit assignment (2) can still be an issue for modDE. In all three environments, we found that there were runs of modDE

that (temporarily) got stuck in locally optimal solutions, in which only one of the agents contributed to the group reward. However, some runs managed to escape or avoid such local optima, which means 2: credit assignment is still a challenge, but it can be overcome by modDE.

Computational complexity (4) is still a major challenge for modDE. Based on the experiments on the Switch2 environment we can see that it takes modDE slightly more samples to converge to the optimal solution (if it does not get stuck). However, the run time required to solve Switch2 is lower for modDE than it is for VDN. Due to our inability to reproduce the results by Mu et al. [30], it is hard to compare the sample efficiency of VDN to that of modDE on the Checkers environment. However, when given a similar run time, the results for modDE are higher on average.

We have noticed that modDE scales poorly with the dimensionality of the problem. The run time and budget requirements for high dimensional problems make the computational complexity of modDE even worse. This might pose a challenge to using modDE in real-world MARL implementations, for which the number of agents and the complexity of the problems might require a larger dimensionality. Especially when these problems can no longer be solved by single-layer NNs.

We found that modDE can find solutions to the Switch2 and Checkers environments in which the agents use significantly smaller network architectures than have previously been found using VDN or other gradient-based methods. Moreover, solutions with similarly complex network architectures will likely not be found by modDE due to its difficulty scaling with dimensionality.

For future research, we would like to run modDE on other MARL environments. Preferably ones with more than two agents as these would provide a bigger challenge in terms of non-stationarity. The dimensionality of the search space scales linearly in the number of agents involved, because each agent has its own network, which is flattened into a vector and concatenated with all other agents. It would be interesting to see how modDE handles this increase in dimensionality.

The dimensionality of the search space is a crucial factor in the performance of modDE in MARL problems. Therefore it could be promising to combine modDE with some form of model-based RL, which can make a lower dimensional representation of the observation space, which can allow for smaller networks for all agents. Pairing modDE with VAEs could also be useful because of this.

References

[1] Richard Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences*, 38(8):716–719, 1952.

[2] Rick Boks, Anna V. Kononova, and Hao Wang. Quantifying the impact of boundary constraint handling methods on differential evolution. *CoRR*, abs/2105.06757, 2021.

[3] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1:1–23, 03 1993.

[4] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[6] Zihan Ding and Hao Dong. *Challenges of Reinforcement Learning*, pages 249–272. Springer Singapore, Singapore, 2020.

[7] Carola Doerr, Hao Wang, Furong Ye, Sander van Rijn, and Thomas Bäck. Iohprofiler: A benchmarking and profiling tool for iterative optimization heuristics. *arXiv e-prints:1810.05281*, oct 2018.

[8] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[9] M. Ganger, E. Duryea, and W Hu. Double sarsa and double expected sarsa with shallow and deep learning. *Journal of Data Analysis and Information Processing*, 4, 2016.

[10] MW Gardner and SR Dorling. Artificial neural networks (the multilayer perceptron)—A review of applications in the atmospheric sciences. *Atmospheric Environment*, 32(14-15):2627–2636, 1998.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[12] John J. Grefenstette, David E. Moriarty, and Alan C. Schultz. Evolutionary algorithms for reinforcement learning. *CoRR*, abs/1106.0221, 2011.

[13] Jean Harb and Doina Precup. Investigating recurrence and eligibility traces in deep q-networks. *CoRR*, abs/1704.05495, 2017.

[14] Hado Hasselt. Double q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

[15] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[17] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[18] Sibi Ittiyavirah, S. Jones, and P. Siddarth. Analysis of different activation functions using backpropagation neural networks. *Journal of Theoretical and Applied Information Technology*, 47:1344–1348, 01 2013.

[19] Diederik Kingma and Max Welling. Auto-encoding variational bayes. *ICLR*, 12 2013.

[20] Anurag Koul. ma-gym: Collection of multi-agent environments based on openai gym. https://github.com/koulanurag/ma-gym, 2019.

[21] Anurag Koul. minimal-marl. https://github.com/koulanurag/minimal-marl, 2021.

[22] Landon Kraemer and Bikramjit Banerjee. Multi-agent reinforcement learning as a rehearsal for decentralized planning. *Neurocomputing*, 190:82–94, 2016.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[24] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Mach. Learn.*, 8(3–4):293–321, may 1992.

[25] Sean Luke. When short runs beat long runs. *GECCO'01: Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, 06 2001.

[26] Warren Mcculloch and Walter Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:127–147, 1943.

[27] Tom Mitchell. *Machine Learning*. McGraw-Hill Education, 1997.

[28] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[29] Thomas M. Moerland, Joost Broekens, and Catholijn M. Jonker. Model-based reinforcement learning: A survey. *CoRR*, abs/2006.16712, 2020.

[30] Ronghui Mu, Wenjie Ruan, Leandro Marcolino, Gaojie Jin, and Qiang Ni. Certified policy smoothing for cooperative multi-agent reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37:15046–15054, 06 2023.

[31] Christopher Olah. Understanding lstm networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015.

[32] F. A. Oliehoek, M. T. J. Spaan, and N. Vlassis. Optimal and approximate q-value functions for decentralized pomdps. *Journal of Artificial Intelligence Research*, 32:289–353, May 2008.

[33] Andreas Ostermeier, Andreas Gawelczyk, and Nikolaus Hansen. A Derandomized Approach to Self-Adaptation of Evolution Strategies. *Evolutionary Computation*, 2(4):369–380, 12 1994.

[34] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[35] Rainer Storn. On the usage of differential evolution for function optimization. *Proceedings of North American Fuzzy Information Processing*, pages 519–523, 1996.

[36] Rainer Storn and Kenneth Price. Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11:341–359, 01 1997.

[37] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinícius Flores Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, Karl Tuyls, and Thore Graepel. Value-decomposition networks for cooperative multi-agent learning. *CoRR*, abs/1706.05296, 2017.

[38] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[39] Diederick Vermetten, Fabio Caraffini, Anna Kononova, and Thomas Bäck. Modular differential evolution. *Conference: GECCO '23: Genetic and Evolutionary Computation Conference*, pages 864–872, 07 2023.

[40] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.

[41] Annie Wong, Thomas Bäck, Anna V. Kononova, and Aske Plaat. Deep multiagent reinforcement learning: Challenges and directions, 2022.

[42] Annie Wong, Jacob de Nobel, Thomas Bäck, Aske Plaat, and Anna V. Kononova. Solving deep reinforcement learning benchmarks with linear policy networks, 2024.

[43] Ka-Chun Wong. Evolutionary algorithms: Concepts, designs, and applications in bioinformatics. *Nature-Inspired Computing: Concepts, Methodologies, Tools, and Applications*, 08 2015.

A modDE hyperparameters

This section contains the parameter settings for most experiments using modDE in Table 1. There are two exceptions to this, as described in Section 5.3. Firstly, the plots in Figure 22 use a different value for F and CR, setting both to 0.5. It also shows three plots with varying boundaries b. Secondly, the plots in Figure 23 show three plots that vary in F and CR. All experiments involving modDE use the rand1bin version of modDE, which randomly selects one vector for the base and reference vectors and uses binary crossover, as visible in Table 1

Table 1: This table contains the value or setting of all hyperparameters for modular differential evolution. The first column shows the name of the parameter as we use it in the text for the rest of this thesis. The second column show the name of the parameter as it is implemented in the modDE framework. The third and last column shows the value or setting that we have chosen for that parameter.

Hyper-parameter name	**name in modDE**	**Value**
Selection strategy for base vector	mutation_base	rand
Selection strategy for base vector	mutation_reference	rand
Random sampler	base_sampler	uniform
Crossover type	crossover	bin
Mutation factor (F)	F	0.7
Crossover rate (CR)	CR	0.9
Population size (Π)	lambda_	$5 \cdot d$
budget (B)	budget	$1000 \cdot \Pi = 5000 \cdot d$
Domain boundaries (b)	ub, lb	± 7
boundary correction	bound_correction	saturate
	lpsr	True
	memory_size	6
	use_archive	True
	init_stats	True
	adaption_method_F	None
	adaption_method_CR	None

B Run time and dimensionality

Table 2 shows the dimensionality and the approximate run time of the algorithms we ran. The methods VDN, RS, and modDE should be self-explanatory. However, in the final two rows, we introduce two variations on the standard modDE method, using larger architectures, which increase the dimensionality of the search space for modDE and therefore require a larger budget. These are "modDE with bias"; the networks used for this method also have bias nodes, and "modDE with hidden"; the networks used for this method also have a hidden layer with 14 nodes.

All experiments were run on the LIACS mithril server. This is a shared server that all computer science students can use. As such, the run time of algorithms can be impacted significantly by the amount of people running algorithms on this server. If many people run their code in parallel, each program will take more time to finish because the resources of this server are finite. Because of this, the run times in this table should only be used to get a rough estimate of the run times involved, not as an exact statistic. Take for example the case of modDE versus modDE with bias on the Compressed Checkers environment. ModDE with bias has a significantly lower run time than modDE, despite having a slightly higher budget and dimensionality.

Even though the run times are only rough estimates, we can still observe that VDN can evaluate the environment approximately 2200 times per hour on Switch2 and approximately 2700 times per hour on Checkers. In contrast to modDE which can evaluate Switch2 approximately 400 000-200 000 times per hour and Checkers 40 500 times per hour. This is a difference of two orders of magnitude, which is still significant despite the inaccuracy of the run time measurements.

The methods RS and modDE were run with the exact same network architectures. Except for the runs where modDE was run with a hidden layer or with bias nodes. All environments we used had two agents. Therefore, the dimensionality of the search space is twice the number of trainable weights per agent. A key takeaway from Table 2 is that the number of trainable parameters per agent is significantly larger for VDN than it is for modDE. For the Checkers environment, the difference is approximately a factor 50, and for the Switch2 environment, the difference is approximately a factor 400.

Table 2: This table contains the approximate average runtime of all algorithms, rounded to the nearest full hour. The first column shows the environment on which the method in the second column is evaluated. The third column highlights in which figures the results of these methods are plotted. The fourth column highlights the total budget in episodes or fitness evaluations of the method on the environment. The fifth column displays the total runtime of each method took to use the full budget on the environment. The sixth column shows the number of trainable parameters per agent for the given method on the given environment. The seventh and final column reports the total dimensionality of the search space. Note that this is twice the number of weights per agent in this case, because all environments have two agents.

Environment	**Method**	**Figure(s)**	**budget**	**run time (h)**	**weights per agent**	**dimensionality**
Switch2	VDN	27	20 000	9	8 773	17 546
Switch2	RS	27	200 000	1*	20	40
Switch2	modDE	27	200 000	1*	20	40
Checkers	VDN	24, 25	250 000	93	11 653	23 306
Checkers	RS	24, 25	2 350 000	20	235	470
Checkers	modDE	24, 25, 28	2 350 000	58	235	470
Compressed Checkers	RS	26	1 200 000	14	120	240
Compressed Checkers	modDE	20, 21, 22, 23, 26, 28	1 200 000	42	120	240
Compressed Checkers	modDE with bias	21	1 250 000	29	125	250
Compressed Checkers	modDE with hidden	20	4 060 000	194	406	812

C Elitism for VDN

The value decomposition networks (VDN), as implemented by Koul [21], are temporal difference (TD) methods, which update at every timestep based on the samples in the replay buffer. Because of this, a VDN can lose performance after a TD update. Unlike modular differential evolution (modDE), which always selects the fittest individual between a parent and its offspring. In this sense, the survivor selection of modDE is elitist. It would be trivial to implement a form of elitism for VDN as well; simply keep track of the best networks found so far and update them if a better combination is found. Because of this, we have modified the plotting algorithm used to plot the results in Figures 25 and 27 to treat the VDN data as if VDN was an elitist algorithm. The "elitist" results for the Switch2 can be seen in Figure 29 and the "elitist" results for Checkers can be seen in Figure 30.

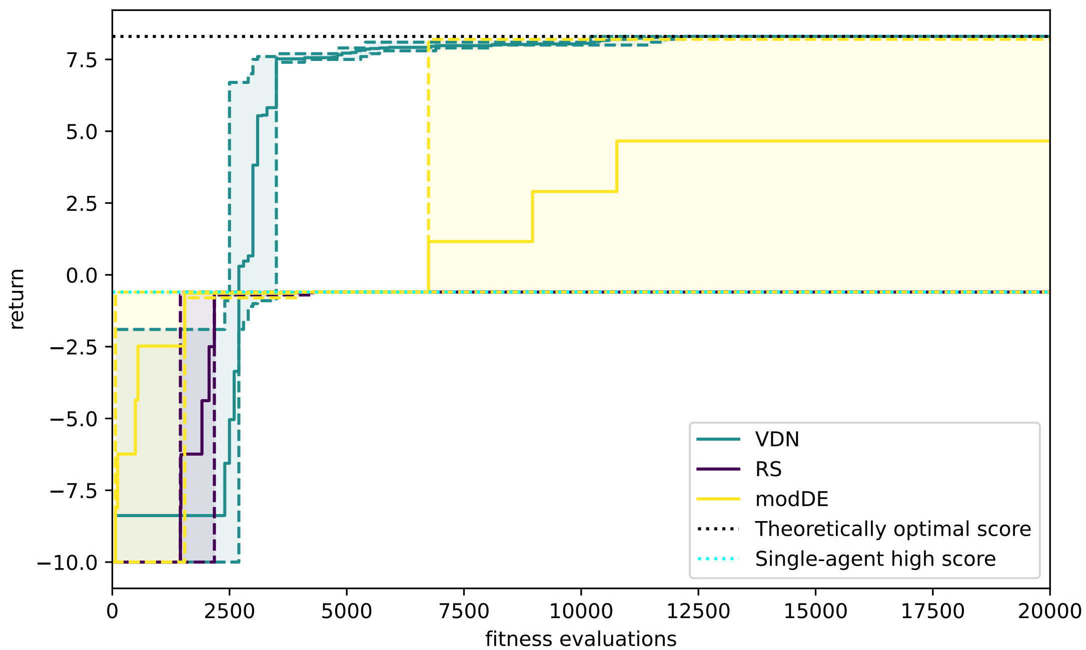

Figure 29: This figure shows the average return plotted against the training time in fitness evaluations modDE, VDN, and RS run on the Switch2 environment. The replications with the highest and lowest returns are also plotted in dashed lines. The dotted lines represent the milestones that can be reached on this environment. VDN is plotted as if it uses elitism like modDE.

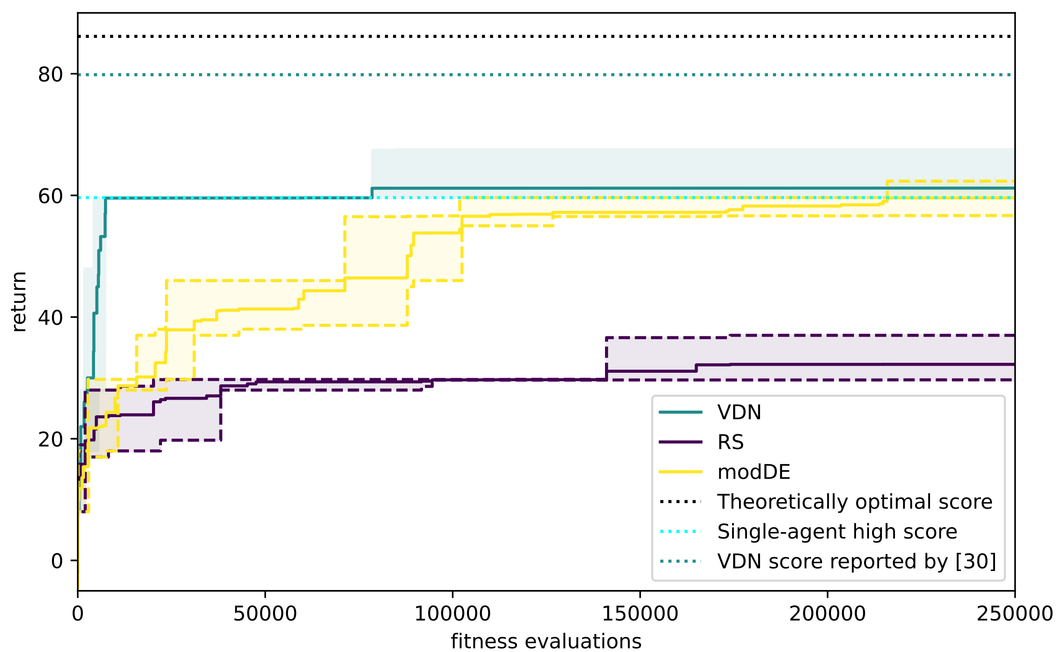

Figure 30: This figure shows the average return plotted against the training time in fitness evaluations modDE, VDN, and RS run on the Checkers environment. The replications with the highest and lowest returns are also plotted in dashed lines. The dotted lines represent the milestones that can be reached on this environment. VDN is plotted as if it uses elitism like modDE.

D Code

The code used to run all experiments in this thesis is available in the following GitHub repository: https://github.com/Koen-AI/Neural-Differential-Evolution