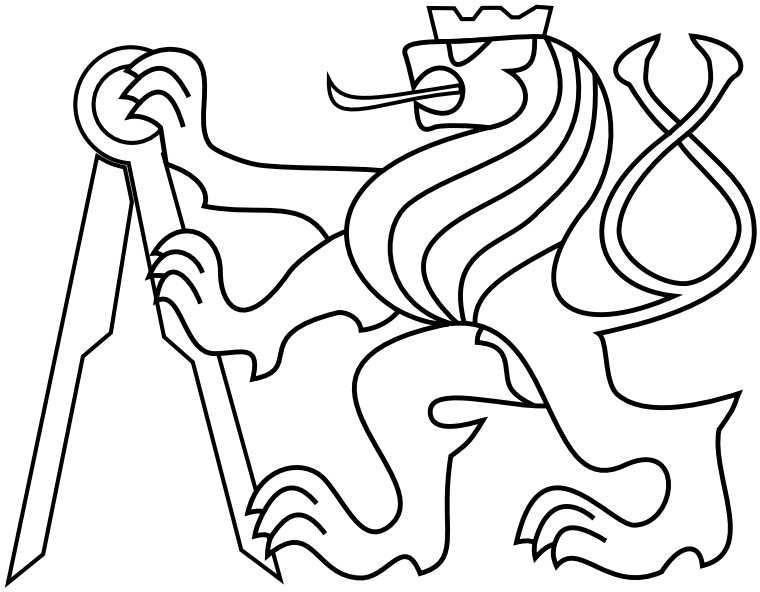
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

MASTER'S THESIS



Kateřina Brejchová

Hybrid Neuroevolution

Department of Computer Science

Thesis supervisor: **Ing. Jiří Kubalík, Ph.D.**
Czech Institute of Informatics, Robotics, and Cybernetics

May 2021

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Brejchová Kateřina** Personal ID number: **465816**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence**

II. Master's thesis details

Master's thesis title in English:

Hybrid neuroevolution

Master's thesis title in Czech:

Hybridní neuroevoluce

Guidelines:

The goal of this thesis is to design and experimentally evaluate a hybrid neuroevolutionary method.
1) Review up-to-date neuroevolutionary methods, focusing on methods using an indirect encoding of neural networks [1,2,3].
2) Choose one neuroevolutionary method and propose its extension using the local search.
3) Implement the proposed method.
4) Choose suitable problems on which the proposed method will be demonstrated (take inspiration from [4]).
5) Experimentally evaluate and analyze the proposed hybrid neuroevolutionary method and compare it with the base neuroevolutionary approach not using the local search.

Bibliography / sources:

[1] Stanley, K. O. et al.: A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks. Artificial Life. 15 (2): 185–212. 2009.
[2] Jaderberg, M. et al.: Population Based Training of Neural Networks, arXiv preprint arXiv:1711.09846, 2017
[3] Lan, G. et al.: Learning Directed Locomotion in Modular Robots with Evolvable Morphologies, arXiv:2001.07804, 2020
[4] OpenAI Gym: https://gym.openai.com/

Name and workplace of master's thesis supervisor:

Ing. Jiří Kubalík, Ph.D., Machine Learning, CIIRC

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **21.02.2021** Deadline for master's thesis submission: **21.05.2021**

Assignment valid until: **19.02.2023**

Ing. Jiří Kubalík, Ph.D. Supervisor's signature	Head of department's signature	prof. Mgr. Petr Páta, Ph.D. Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce her thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

Date of assignment receipt	Student's signature

Author statement for graduate thesis

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 21st May 2021 Signature .....

Acknowledgements

I would like to thank Ing. Jiří Kubalík, Ph.D., for all the provided consultations and his insights on the topic. I would also like to thank my partner Ota for proofreading the text and my family for their never-ending support during my studies.

Abstract

Neuroevolution is an approach for learning artificial neural networks by an evolutionary algorithm. The evolutionary algorithm can evolve both the topology of the network as well as weights and biases. While evolutionary algorithms perform well in exploration, local fine-tuning can be problematic. This thesis proposes a hybrid approach that combines the neuroevolutionary algorithm HyperNEAT with gradient-based algorithm DQN. Firstly, we propose three different options for initialising DQN by a solution found by HyperNEAT. Secondly, we propose a method for fine-tuning HyperNEAT's population by a solution found by DQN. Finally, we combine the two proposed steps into a training loop that iteratively runs the sequence of HyperNEAT and DQN. We test the approach in a reinforcement learning control domain with discrete action space, namely Cart pole, Acrobot and Mountain car environments from OpenAI gym. We conclude that the main challenge in combining the two algorithms is the different interpretability of their outputs. We describe the initialisation strategies that did not work and discuss the possible reasoning behind it. We show promising results for both the DQN and the HyperNEAT initialisation.

Abstrakt

Neuroevoluce je metoda trénování neuronových sítí pomocí evolučních algoritmů. Evoluční algoritmus může vyvíjet jak topologii sítě, tak i váhy a práhy. Zatímco evoluční metody zvládají dobře exploraci prostoru, dělá jim problém lokální dolazení řešení. Tato práce navrhuje hybridizovaný přístup kombinující neuroevoluční algoritmus HyperNEAT s gradientním algoritmem DQN. Nejprve navrhujeme tři různé způsoby inicializace DQN pomocí řešení z HyperNEATu. Poté navrhujeme metodu na dolazení populace HyperNEATu pomocí řešení z DQN. Nakonec kombinujeme oba navržené kroky v trénovací smyčku, která iterativně spouští sekvenci HyperNEAT a DQN. Daný přístup testujeme v doméně posilovaného učení s diskrétním prostorem akcí, jmenovitě na problémech z řízení Cart pole, Acrobot a Mountain car, které jsou definované v OpenAI gym. Docházíme k závěru, že hlavní výzvou pro kombinování obou přístupů je odlišná interpretovatelnost jejich výstupů. Uvádíme i inicializační strategie, které nezafungovaly a diskutujeme možné příčiny. Prezentujeme slibné výsledky navrženého postupu.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Neuroevolution is a subfield of artificial intelligence that combines evolutionary algorithms and neural networks. The traditional approach with neural networks is to train them using gradient-based methods such as stochastic gradient descent. This approach has experienced many successes over the recent years in domains such as image classification [1], natural language processing [2] or recommender systems [3]. However, an often mentioned problem is that it takes time and a lot of experience to design the neural network architecture to be efficient, accurate, and learn quickly. Evolutionary algorithms are inspired by natural evolution and work with a population of highly randomised solutions that are gradually recombined and mutated to be pushed towards the optimum. They show success, especially for black-box problems where exploring a suboptimal solution is very valuable and hard to obtain by conventional algorithms. An example of such domains are combinatorial [4] or control problems [5].

One of the efficient algorithms combining neural networks and evolutionary algorithms is NEAT [6]. This method represents the networks directly as graphs and evolves them similarly as in genetic programming. The algorithm starts with minimal graph structures and gradually complexifies them as the algorithm progresses. While this method worked for smaller networks, with the growth of the typical network and the emergence of deep learning, it was clear that direct encoding of the networks would not be sufficient due to memory and speed requirements.

Hence, indirect encoding is used to represent the neural networks. This idea is used in a method called HyperNEAT [7]. The encoding is a function called compositional pattern-producing network (CPPN) that, given coordinates of two neurons in the network constrained by a specified substrate (network size and connectivity), generates the weight between them. Then, rather than recombining large neural networks, we recombine more compact functions that generate them.

We aim to combine HyperNEAT with a gradient-based Deep Q-learning Networks (DQN) algorithm [8]. The motivation is that the hybridised solution leverages the exploration capabilities of evolutionary algorithms as well as the fine-tuning capabilities of the gradient-based methods. To accomplish this, we investigate the following possibilities of hybridisation:

1. We take the solution found by HyperNEAT and fine-tune it by local search.
2. We take the fine-tuned model and backpropagate its parameters to the population of generating functions from HyperNEAT.
3. We iterate these two approaches to obtain a well-performing solution.

We test the baseline HyperNEAT, DQN and the proposed approaches on three classic control problems in the reinforcement learning domain. We use the state-of-the-art DQN algorithm both for our local search extension as well as for the baseline solution. The goal of the experiments is to show the effect of DQN policy initialisation by HyperNEAT and the effect of fine-tuning CPPN(s) based on the weights and biases of the policy network trained by DQN.

The main contributions of this thesis are:

- the proposed strategies for initialisation of DQN by HyperNEAT policy and vice-versa,
- an experimental evaluation of the hybrid neuroevolution approaches,
- a PyTorch-based implementation of the proposed methods.

The thesis is divided into the following chapters: Chapter 2 describes the basics of neural networks, evolutionary algorithms and neuroevolution, where we review the relevant state-of-the-art literature. Chapter 3 describes the tested reinforcement learning domain and relevant algorithms. Chapter 4 presents the proposed solution, more specifically the combination of HyperNEAT and DQN. Chapter 5 briefly describes the implementation and the used software. Chapter 6 provides results from the experimental evaluation of the proposed approaches and discusses the results. Chapter 7 summarises the outcomes of the thesis.

2 Neuroevolution

This chapter gives a background on neuroevolution. Neuroevolution is a class of methods that evolve neural networks by evolutionary algorithms. First, we describe the basics of neural networks and evolutionary algorithms. Then, we continue with the neuroevolution, and its different variants. We focus on two neuroevolutionary methods, NEAT and HyperNEAT, which are the foundation of this thesis.

2.1 Neural networks

This chapter gives a brief overview of neural networks that have been a widely used AI tool over the past decades [1], [2] and [9]. The information from this chapter was obtained in [10] and [11]. Neural networks are inspired biologically by neurons in the human brain. Each neuron gets input signals from its neighbouring neurons connected by synapses, processes them and sends a signal to its neighbours. The neurons are interconnected into huge networks and together are able to represent a complex behaviour. The human brain has approximately 86 million neurons [12].

The artificial neurons are traditionally formed into layered architectures where each layer contains several neurons that are connected with the neurons in the preceding and successive layers. In feed-forward architectures, the neuron gets input from its preceding neurons weighted by the connecting edges, sums the weighted inputs together, adds a neuron-specific threshold (bias) and performs a non-linear activation function (see Figure 1). In this way, the signal gradually travels from the first input layer to the last output layer (see Figure 2).

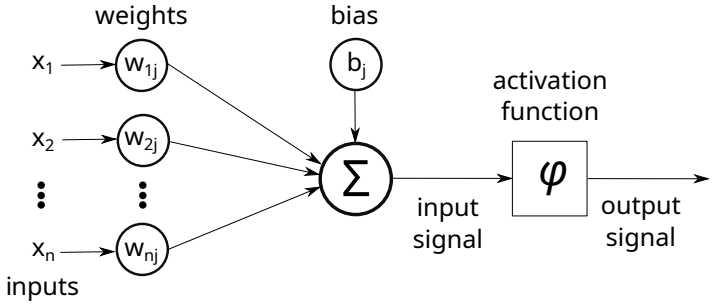


Figure 1: Artificial neuron [13] with input $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ and output $\varphi(b_j + \sum_i w_{i,j} \cdot x_i)$.

The goal is to correctly design the architecture with the parameters being mainly the number and types of layers and the way they are interconnected, the number of neurons in each layer, and the type of activation functions. The learning objective is to find the (sub)optimal parameters (connection weights and neuron biases) such that the performance of the network is high and generalises well to previously unseen data.

A neural network is an approximator of a function that has a feature vector on the input. Such function is either used for classification or regression. In the former, the output vector of the network usually represents probabilities with which the feature vector belongs to a given class. An example of such task is a neural network that gets an image of a single written

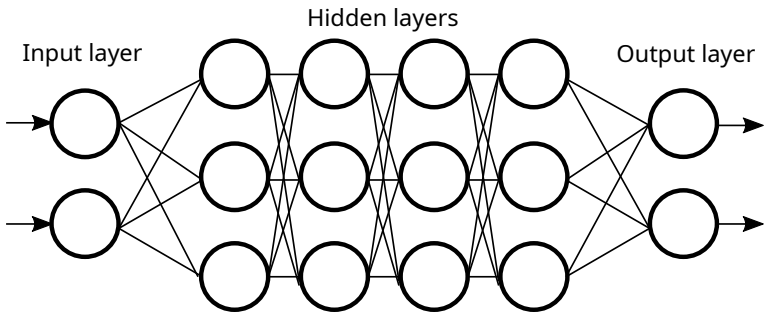


Figure 2: Feed-forward fully connected neural network architecture with four hidden layers.

digit on the input and outputs a probability vector of size 10, saying for each possible digit, what is the probability that it was on the input. In the latter, the output represents a vector of problem-specific values. An example of such task could have a feature vector describing a movie on the input (with features such as the main actors, release year or genre) and a vector describing the percentual success of the movie (values such as revenue or user rating) on the output.

We measure the performance of the trained neural network using a loss function that compares the expected and the approximated outputs and shows how 'different' the outputs are. There are many different loss functions that are suitable for different types of problems, such as absolute error, mean square error or cross entropy. Absolute error calculates the sum of the absolute differences in the output and is suitable for regression problems where having a sparse network is beneficial as it pushes the parameters to zero. Mean square error calculates the sum of the squared differences of the outputs and is suitable for regression problems where being approximately correct in all cases is better than being correct in some cases but very incorrect in a few cases. Cross entropy is used for classification problems and measures the difference in distributions of the real and approximated classes.

When we consider the neural network in terms of linear algebra, the neurons accept a linear combination of the weighted preceding signals, and the activation functions are differentiable (or have predefined behaviour in their non-differentiable parts). Hence, we can optimise the network parameters (weights and biases) by gradient backpropagation that aims to modify the parameters so that the loss is minimised. To do that, we create a computational graph representing the compound derivatives of each parameter in the network, calculate the loss over the input samples, and modify each parameter by the calculated gradient.

There are three main practical issues connected to that. Firstly, the search can get stuck in the local optimum. Secondly, due to high memory and time requirements, we cannot possibly input all possible samples to the network and make the gradient update in one step. Lastly, we probably do not have all the possible input samples that exist, or we might not have the perfect architecture design. Ignoring these issues could result in technical infeasibility to perform the optimisation or optimising different criteria than is required. To partially resolve that, stochastic gradient descent is used (see Algorithm 1). This method iteratively samples a small part (batch) from the dataset and performs an update over this small sample only. The relative size of the updates is controlled by a learning rate that says how much we want

to change the parameters in the given iteration. It has been proved [14] that the learning converges if certain conditions such as properly selected dataset, learning parameters, and loss function are met. The optimisation is sensitive to getting stuck in the local optima. To mitigate this, more complex optimisation methods than SGD such as momentum [15] or ADAM [16] are used. These methods are taking into account not only the current gradients but the combination of the previous ones as well and adaptively change the learning rate.

Algorithm 1 Stochastic gradient descent pseudocode for dataset D with samples x^i, targets y^i, the number of episodes E, learning rate α and a function h parametrised by θ

$\theta \leftarrow$ random init
for *epoch* $\in [1, \ldots, E]$ **do**
    shuffle D
    for minibatch of size M containing $\mathbf{x^i}, y^i \in D$ **do**
        get prediction $\hat{y}^i \leftarrow h(\mathbf{x^i})$
        get loss $L \leftarrow \frac{1}{M} \sum_{i=1,\ldots M} loss(\hat{y}^i, y^i)$
        get gradients $\Delta\theta \leftarrow -\nabla_L \theta$
        make gradient step $\theta \leftarrow \theta + \alpha \cdot \Delta\theta$

Many problems can arise while training the network where most of them are connected to deep learning [17], [9]. One of them being overfitting to the given dataset, which results in poor performance on unseen data. We can use a separate validation dataset to stop the training when the loss would still decrease on the training data, but it would increase on the validation data. Moreover, we can use dropout layers [18] that randomly turn off some of the connections, making the network more stable. Another problems are related to the size of the gradient. With the growing size of the network, it can happen that the gradient vanishes [19], which results in updating only the parameters towards the end of the network. For example, this can easily happen with ReLU activation functions [20] that have zero gradient for negative input, so they are not propagating gradient any further. We can carefully design the architecture or introduce skip connections [21] that interconnect nodes in non-neighbouring layers to mitigate this. On the other hand, too large gradients cause a gradient explosion [22] resulting in the parameters being updated from one extreme to another and not learning anything useful. Techniques such as regularisation or data normalisation [17] help to avoid this. The first one pushes the parameters to stay in a reasonable range of values. The second one helps the network so that it does not have to learn how to normalise the data itself.

In the scope of the thesis, we use neural networks as simple function approximators and avoid using advanced techniques. We work with shallow networks with a moderate number of neurons only (typically 4 hidden layers, 32 neurons each). The neural networks we use are feed-forward and fully connected, meaning that there are no cycles in the computational graph, and each neuron in layer i is connected to each neuron in layer $i + 1$. We solve a regression task using SGD optimisation in a reinforcement learning domain. We focus mainly on combining the two algorithms (HyperNEAT and DQN) rather than finding the perfect hyperparameters for the neural networks themselves.

2.2 Evolutionary algorithms

Evolutionary algorithms are based on Darwin's principle of natural selection. The principle says that in a population of varied individuals, the strongest ones are preferred for reproduction. This helps the whole population to conserve the most substantial features and survive. The information from this chapter was obtained in [23] and [24]. Evolutionary algorithms mimic this behaviour by evolving a population of individuals. Each individual in the population represents a possible solution in a predefined domain (e.g. one individual can be one neural network with fixed parameters). The individual is represented by a *genotype* which is encoded information defining the individual, and *phenotype* which is an interpretation of the genotype. For example, if our individual was a real value, the phenotype would be the specific value, and the genotype could be its binary encoding. The individuals are evaluated by a *fitness* function that is to be maximised. Fitness says how good the individual is in the given environment.

Each individual can be modified by a *mutation* operator. The operator slightly changes the genotype so that part of the genotype is kept while the other part is randomly modified. Mutation pushes towards diversity in the population and helps to explore the search space by performing a local search around the current genotype. The advantage against random initialisation of a new genotype is that some of the beneficial genes are kept in the individual. An example of mutation is a bit-flip for binary representation or adding Gaussian noise for real representation.

The individuals reproduce using a *crossover* operator where the typical arity of the operator is two. The idea behind the operator is that by mixing two well-performing solutions, we can get a new solution (called child or offspring) that takes the best out of its parents. One-point crossover finds a crossing point in both of the parents and combines the first part of one of the parents with the second part of the other parent to create a new offspring. Similar is a two-point crossover with the difference that there are two crossing points, and we create the offspring by combining three parts selected from the parents. Uniform crossover selects each gene of the offspring randomly from the parents.

The population of individuals evolve in generations. During each generation, a new population is created from the current population by selecting parents and applying crossover and mutation operators to create new individuals (see Algorithm 2). The new population is then evaluated by the fitness function and replaces the old population.

There are different *replacement and selection strategies*. We can either replace the whole population (generational replacement) or replace just some individuals in the population with the new offsprings (steady-state replacement). These two represent the tradeoff between exploration (we search for as many candidate solutions as possible) and elitism (we keep the well-performing candidate solutions in the population). Selection determines which individuals enter the crossover as parents. The goal of selection is to prefer the well-performing parents to push towards optimality but also to select some of the weaker individuals to push towards diversity. Roulette wheel selection chooses parents randomly proportionate to their fitness. Tournament selection gradually randomly samples a small batch of individuals and adds the best of them to the pool of parents until it's full.

Algorithm 2 Evolutionary algorithm pseudocode

```
1: initialise(population)
2: evaluate(population)
3: while not termination condition do
4:     parents ← select(population)
5:     offsprings ← crossover(parents)
6:     mutate(offsprings)
7:     evaluate(offsprings)
8:     population ← offsprings
9: return bestof(population)
```

Different problems can arise that primarily result in a stagnating population. The population stagnates if the best or mean fitness does not change and there is no diversity among the individuals causing the crossover happening between two same parents outputting the identical offspring. This can be partially solved by using a sufficiently explorative mutation that would introduce new individuals to the population. However, this might not help when the fitness of the stagnating population is much higher than the fitness of the newly introduced individual. As the selection procedures are fitness proportionate, it could easily happen that the new offspring would stay in the population just for one generation and would not help it to escape from the local optimum. Methods such as fitness sharing or speciation [25] help to mitigate that by modifying the fitness of each individual so that it competes mainly with the individuals similar to it.

2.2.1 Genetic programming

Genetic programming is a subfield of evolutionary algorithms concentrating on genotypes that are trees representing a hierarchical program or function [26]. The tree accepts input using its leaves and gradually propagates the signal to its root while applying a function or operator in each of the nodes that it is passing through. See Figure 3 for examples of functions represented using a tree data structure. A typical application of GP is symbolic regression, where the task is to find an analytic expression that fits the training data the best [27].

Genetic programming follows the same evolution scheme as a classical evolutionary algorithm (see Algorithm 2) but differs in the mutation and crossover operators. Subtree crossover selects a node (crossover point) in each of the parents and replaces the selected subtree from the first parent with a subtree from the second parent (see Figure 3). Subtree mutation selects a node (mutation point) in the individual and replaces the subtree with a randomly generated tree. Point mutation selects a mutation node and changes its operator to a different operator of the same arity.

The main challenge of genetic programming is handling code bloat. Code bloat refers to the excessive growth of the tree containing inviable code (a never reached branch) or unoptimised code (branch representing a formula reducible to a shorter expression). Bloat happens by gradually replacing subtrees with deeper subtrees using crossover or mutation, emerging from the fact that there are more deeper-level nodes than shallow-level nodes chosen

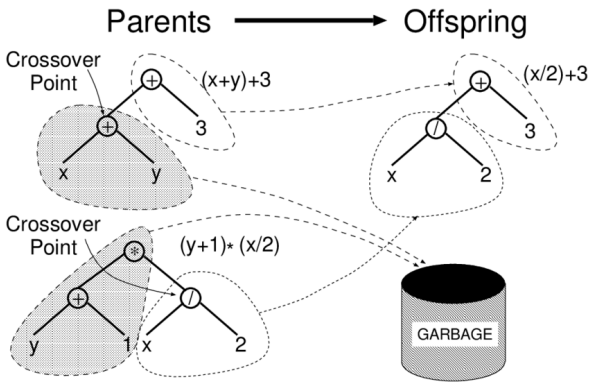


Figure 3: Subtree crossover by [26], CC BY-NC-ND 2.0 UK

for mutation/crossover. Growing the size of the tree makes it harder to create well-performing offsprings. To resolve that, we can introduce a reduction function that would compress the tree's redundant parts or limit the maximal depth of the tree.

2.3 Neuroevolution background

This chapter describes the basics of neuroevolution, which is the core part of the thesis and reviews the relevant literature. We also review relevant tasks that are commonly solved by neuroevolution and available HyperNEAT software. This section is composed mainly of the information contained in survey/review articles on neuroevolution [28], [29], [30] and [31].

Neuroevolution is a subfield of artificial intelligence that interconnects the main concepts of evolutionary algorithms and deep learning. A typical goal of neuroevolution is to train a neural network using evolutionary optimisation algorithms. The optimisation focus ranges from the network architecture [32], network parameters [33] or network hyperparameters such as learning scheme [34].

The basic unit over which the evolutionary algorithm operates is a population of individuals. Each individual is a genotype representing a neural network. There are two ways of encoding being used. *Direct encoding*, where the neural network is specified explicitly by one to one mapping, and *Indirect encoding*, where we specify how the network should be generated. The advantage of the direct encoding is its completeness and that we do not have to work with another level of abstraction. On the other hand, the indirect encoding allows more variability and a compact representation, which is crucial, especially in the case of neural networks. The neural network is referred to as *phenotype*, while its encoded form is called a *genotype*.

The population of individuals (genotypes) is evolved by a classical evolutionary algorithm scheme. The mutation and crossover strategies directly depend on the chosen encoding strategy. As the crossover of neural networks is rather complicated [6], some of the authors [33] work with the mutation operator only.

A recent survey has shown that there is a big potential in neuroevolution [31] and pointed out possible directions where the biologically inspired algorithms could go.

In 2018, Uber researchers released experiments [33] showing that even a simple genetic algorithm is able to outperform gradient-based (Q-learning), and gradient approximation based (Evolution strategies) methods in the reinforcement learning domain. Namely, they tested their implementation on Atari games, humanoid locomotion, and image maze domains. The same authors propose an indirect encoding method that recreates the neural network by applying mutations, specified by a vector of random mutation seeds, to the original network. The implementation of A. Ecoffet [35] shows the reproducibility of the results and points out problems related to the robustness of the solution that could be improved if better datasets were used for training.

In 2020, Google researchers released a study [36] on indirect encoding of vision-based reinforcement learning tasks. This was achieved by introducing the concept of self-attention that could be intuitively interpreted as intentional blindness, which leads to simplification of the visual space. They used neuroevolution, more specifically CMA-ES algorithm, to train their network. The experimental results showed the potential to generalise with a compact encoding.

In the following text, we will describe the core neuroevolution approaches with a main focus on NEAT based methods, that are often referred to in the community.

2.3.1 NEAT

Neuroevolution of Augmenting Topologies (NEAT) addresses how to evolve both network topology and the parameters simultaneously. The authors [6] introduce the following innovations:

- crossover of different topologies
- specification for protecting different topology types
- initialisation with minimal structures

Specification protects the fitness diversity in the population which prevents stagnation. Minimal initialisation ensures that the solution complexity grows from the smallest structures to the most complex. This increases the probability that the best-performing individual is as simple as possible. If the individual gets too complex, it gets harder to recombine or mutate it into a meaningful solution.

The crossover innovation addresses the issue of competing conventions (permutation problem). Traditionally, crossover is problematic in subgraph swapping methods (see Figure 3) as it is challenging to recognise homology between different networks. If the permutation problem is not handled, crossovers of two homological individuals, as shown in Figure 4, degrades the performance of the network.

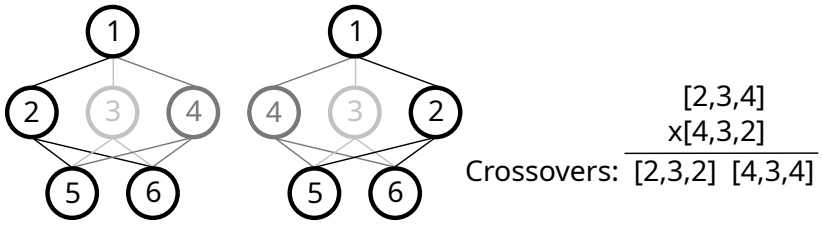


Figure 4: Competing conventions problem. Crossover is performed over two same networks, and the resulting solution loses a key feature as it was not identified that the networks are in fact the same. Both of the networks contain features 2,3,4; but when we recombine the two homological networks, the resulting network is either missing information from feature 2 or feature 4, while no new feature is added. Image adapted from [6].

In NEAT, this problem is solved by a special kind of direct encoding (see Figure 5) that tracks the history of modifications applied to the network and, therefore, can recognise when two networks were created in the same way. Two networks with the same history have the same topology, but not necessarily the same weights. To perform such tracking and to keep the networks minimal, the algorithm starts by evolving simple neural networks with no hidden nodes, rather than starting with a population of randomly generated networks.

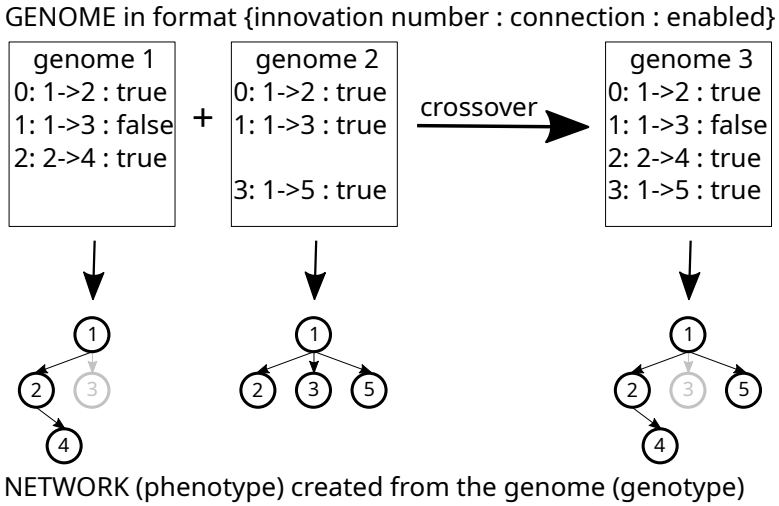


Figure 5: NEAT crossover of two genomes and their network representation. Each genome consists of list of connections, their unique innovation number and whether they are enabled or not.

During the crossover phase, the genomes are compared gene-vice, and matching genes are selected randomly from any parent, while disjoint genes are selected from the better parent. The concept of disjoint/compatible genes is also used in the fitness sharing function, ensuring that the diversity in the population is kept.

Even though NEAT experiences scalability problems induced by direct encoding, it is key research on which the more recent research builds. One of the recent papers, that still feature NEAT (though only partially) is [37]. The authors solve a RL task by strictly separating policy

and feature learning (algorithm NEAT+PGS). While the policy evolves by Policy Gradient Search, the features are evolved by NEAT algorithm.

2.3.2 Population Based Training

Another direction of neuroevolution in recent years is Population Based Training (PBT) by Jaderberg, et al. [34]. PBT jointly optimises weights and hyperparameters (learning rate, and entropy cost) of the network to reduce the cost of ML model deployment. An asynchronous optimisation model is used to utilise computational resources effectively. The final output of the algorithm is a network with fixed architecture, and a schedule of hyperparameters (i.e., if we want to retrain the network later, the hyperparameters dynamically change during the training).

Firstly, for each individual in the initial population, the parameter vector (network weights) is updated using a gradient descent method. Then, there is an exploit phase, where given the fitness of the individual and the rest of the population, the algorithm either keeps the current solution or accepts the weights of the best performing model. The individual then continues to an explore phase, where the hyperparameters are changed to suit the new parameter vector better. These three steps (SGD update, exploit, explore) are repeated until the training ends. The best performing network is selected.

In Li, et al. [38], PBT is further extended to perform black-box optimisation, which means that no assumptions on model architecture, loss function, and optimisation scheme have to be made. The algorithm is based on Vizier, a hyperparameter optimisation service [39] and the PBT optimisation algorithm, the difference is that the mutation is not performed inside the worker, but in the supervising controller which allows more variability.

2.4 HyperNEAT

HyperNEAT addresses the scalability problems of approaches with direct representation by introducing a concept of Compositional Pattern Producing Networks (CPPN) to manage indirect encoding of the neural networks. In HyperNEAT, one individual is not the network itself but a CPPN that generates it. These CPPNs are then optimised using the original NEAT algorithm (see Algorithm 3).

CPPN works over a predefined grid called a substrate. Such substrate could be a cube with regularly spaced nodes where each node is connected with its direct neighbours; or a cube divided into layers where the connections are directed from layer i to layer $i + 1$. CPPN is a function that accepts coordinates of two nodes in the substrate and returns the weight of their connection. The whole network is built by querying all the allowed connections. While the network structure is limited by the structure of the underlying substrate, it can still learn to produce many different architectures if we consider that the connections with low weights are not created in the network. See the process of converting CPPN phenotype into the neural network with initialised weights in Figure 6.

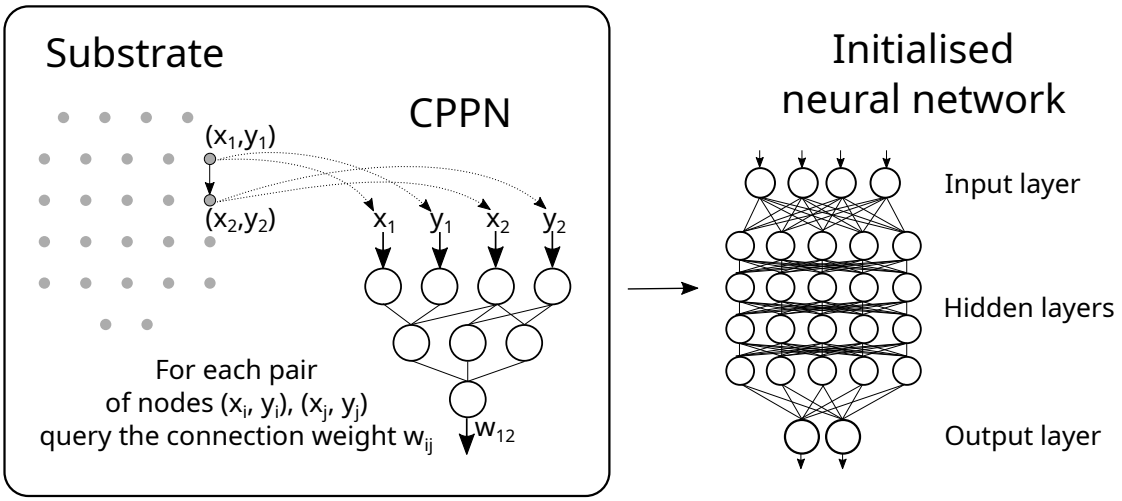


Figure 6: Genotype (CPPN) to phenotype (neural network) mapping. Firstly, every possible connection of a pair of nodes in the substrate is queried. Secondly, the coordinates of the pair of nodes are the input of the CPPN network. Lastly, CPPN outputs the weight of the connection between the pair of nodes.

Algorithm 3 HyperNEAT pseudocode adapted from [7]

```
choose substrate configuration (node layout)
initialise population with randomised minimal CPPNs
while not termination condition do
    for each individual in the population do
        query its CPPN for connection weight of each possible connection in the substrate
        run the created neural network to obtain fitness
    reproduce CPPNs according to NEAT method to get updated population
return bestof(population)
```

There are two main advantages of the indirect representation. Firstly, it is able to generalise to different substrate sizes, which can be especially useful if the size of the input (data sample) or output is variable. Secondly, the representation is compact, allowing to represent thousands of neural network parameters only by dozens of CPPN nodes.

While HyperNEAT has been criticised by van den Berg and Whiteson [40] for not performing well on irregular tasks, the authors of HyperNEAT have shown [41] counterexamples to disprove it. Several years later, a survey [28] on the progress of HyperNEAT and its extensions and applications was published. The findings are presented in the next paragraphs.

Adaptive HyperNEAT (Risi & Stanley [42]) has focused on the plasticity of the networks, i.e. the ability to evolve weights. This improvement helps to circumvent the weight limitations imposed by the nature of the CPPN.

HybrID (Clune, et al. [43]) starts with indirect encoding and then continues with direct encoding to allow individual weight fine-tuning.

HyperNEAT-LEO (Verbancsics & Stanley [44]) addresses the issue of disappearing connections by adding an output to CPPN that determines whether the connection should be present or not. This improvement helps to distinguish which connections are in the network, and which connections are below a threshold.

ES-HyperNEAT (Risi & Stanley [45]) adds the possibility to evolve position and number of hidden layers in the substrate. This improvement allows the substrate to evolve with the CPPN, and not to remain prefixed by the user.

HyperGP (Buk [46]) uses genetic programming instead of NEAT to evolve the CPPN.

Deep HyperNEAT (Sosa & Stanley [32]) is an extension of NEAT where one node in a genotype does not represent a neural network node but a neural network layer and its specifications. Fitness is based on how well the network can be trained in a few generations.

CoDeepNEAT (Miikkulainen, et al. [47], and Bohrer, et al. [48]) is similar to Deep HyperNEAT, but instead of having neural network architecture as a population, we evolve two different populations – blueprints and modules. Modules are small parts of an architecture, and blueprints are the recipes that assemble them.

2.4.1 Available HyperNEAT implementations

As the goal of this thesis is to combine NEAT-based algorithms with SGD, we implement the code in Python as it is currently the most used open-source machine learning language with major deep learning libraries (Keras, PyTorch), data handling libraries (Pandas, Seaborn) and RL experimental platforms (OpenAI gym). Therefore, I focus on the libraries implemented in Python. However, there are also other (Hyper)NEAT libraries, especially for C# and C++. These implementations can be found in the software list maintained by the EPLEX group at UCF (`http://eplex.cs.ucf.edu/software-list`).

While there is plenty of available implementations of NEAT-based algorithms, it is challenging to distinguish between them, and choose the right one to use it 'as a tool' since most of the published implementations are paper-/project- related, and not maintained or not general enough.

NEAT is implemented by McIntyre, et all. in [49] and indexed in PyPI as `neat-python`. This implementation looks stable, tested and quite well documented. It is not further developed, but it is still maintained. This is also the NEAT implementation that is often built upon in the research papers.

PyTorch based implementation is provided by Uber research [50]. This implementation covers NEAT, HyperNEAT, and Adaptive HyperNEAT. The interface looks general enough. However, it seems that it is not further maintained as the reported issues are not resolved.

Tensorflow based implementation is made by Bodnar [51]. However, this implementation is based on Tensorflow 1.x, while the current standard is Tensorflow 2.x, and as the author states in his technical report, the implementation is significantly slower than the PyTorch version.

Another *Tensorflow based* implementation is done by Uber research [52]. This implementation supports GPU evaluation and parallelisation. However, only a simple genetic algorithm and algorithm for evolution strategies is implemented. The project is labelled as work in progress, but the last update was done one and a half years ago. Even though the code might not be applicable as is at the moment, it could be worth it to improve it as GPU supported operations, and a good parallelisation could significantly speed up the runtime of the experiments.

2.5 Use cases of NEAT based methods

Generally, HyperNEAT-based methods are used mainly for tasks with significant geometrical features (e.g. symmetry, repetition) or for tasks where varying input/output size handling is required (i.e. image resolution). Such features are often found in robot control, image recognition, and reinforcement learning tasks.

2.5.1 Vision

The power of CPPNs to create geometrical objects is shown in Clune & Lipson [53] where 3D objects are designed and evolved. The paper is accompanied by a popularisation website `http://endlessforms.com/`.

Calimeri, et al. [54] show the application of HyperNEAT in the biomedical domain for blood vessel segmentation and compare the method with other state-of-the-art methods for image segmentation. Verbancsics & Harguess [55] use HyperNEAT to classify maritime vessels from satellite images. The main presented advantage of using HyperNeat is the ability to work with varying scales of images. CoDeepNEAT is used in Miikkulainen, et al. [47] to create image captioning of a major online magazine. The authors use image and text representation of the items to produce captions for blind people.

Neurogram is an online tool (`https://otoro.net/neurogram/`) by Ha [56] that implements the NEAT algorithm to allow the user to experiment with NEAT operators on random images. Picbreeder is an online community-based tool (`http://www.picbreeder.org/`) by Secretan, et al., [57] which evolves art using the NEAT algorithm based on user experience. The users can create new art pieces by evolving existing pictures or combining them with their own art.

2.5.2 Games

There has been a lot of research applying neuroevolutionary algorithms in Atari gaming. This domain is of particular interest as it is defined in a very convenient open-source simulator OpenAI Gym [58], and provides several complex environments corresponding to different Atari games. To play Atari, one has to be able to map a quickly changing screen capture to actions. As the different environments are similar but still different enough, the domain is often used to test the ability to generalise.

Hausknecht, et al. [59] use HyperNEAT; Such, et al. [33] use a simple version of a genetic algorithm without mutation; and Peng, et al. [37] use NEAT+PGS to create a potentially general Atari player. This list is not exhaustive as this domain is very popular in Reinforcement learning research as well. Recently, Atari ZOO [60] was published to allow easier comparison between the implemented algorithms.

Schrum [61] presents a HyperNEAT-based algorithm specialised to generate CNN-like architectures. The author shows the results using the Tetris game puzzle.

2.5.3 Robot control

Cheney, et al. [62] evolve soft robots to create new morphologies. They utilise HyperNEAT to create morphologies of varying density. Lee, et al. evolve robot gait [63] using HyperNEAT for a 4-legged robot. Drchal, et al. [64] use HyperNEAT for a line following robot reacting to variable size sensor inputs.

Buk [46] uses HyperGP to control robots using their sensoric inputs. The author employs the indirect encoding concept of CPPN while disregarding the complexities of the NEAT algorithm, and using genetic programming instead. Dvorský [65] then uses HyperGP in his master thesis to control multi-legged robots.

Haasdijk, et al. [66] control a multi-robot organism by HyperNEAT. The authors leverage the geometric aspects of HyperNEAT to create an algorithm that allows each of the robots to operate autonomously.

2.5.4 Other

Bahçeci & Miikkulainen [67] explore the possibilities to transfer heuristics created for simple board games and use them as a hot start for more complex games. Didi [68] extends HyperNEAT for a policy transfer task with experiments completed in keep-away RoboCup soccer domain.

Boyles [69] evolves scouts for military simulations using the NEAT algorithm in his master thesis. Kroos & Plumbley [70] extends NEAT to use it for sound event detection where the main goal is to develop a competent network that is as minimal as possible.

3 Reinforcement learning

Reinforcement learning aims to design an agent that maps states to actions to maximise the agent's reward in the environment. The information from this chapter was obtained in [71] and [72].

3.1 Background

Reinforcement learning model is defined as

- S – set of states, can be discrete or continuous
- A – set of actions the agent can perform, can be discrete or continuous
- $R : S \times A \to \mathbb{R}$ – reward function determining reward $r \in \mathbb{R}$ that the agent gets after performing action $a \in A$ in state $s \in S$
- t – discrete time steps, $t \in [0, \ldots, T]$
- $P : S \times A \times S \to [0, 1]$ – probability function $P(s'|s, a)$ defining the probability that agent applying action a in state s gets to state s'

The model of the agent and the environment is shown in Figure 7. The agent has a policy $\pi : S \to A$ that, given a state s, outputs action a that the agent should take. The goal of the reinforcement learning methods is to design π^* that maximises the reward of the agent in the environment.

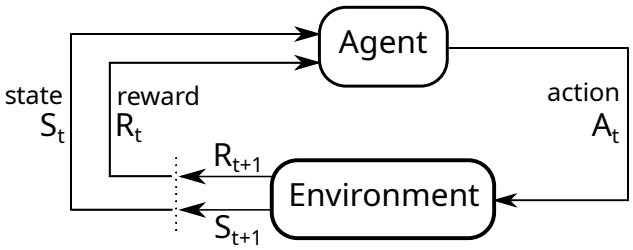


Figure 7: Reinforcement learning as Markov decission process

A value function $V_\pi : S \to \mathbb{R}$ assigns each state s the expected mean reward obtained by following policy π from state s. A action-value function $Q_\pi : S \times A \to \mathbb{R}$ assigns each pair s, a the expected mean reward obtained by following policy π from state s.

The V- and Q-values are tied by the Bellman optimality equation. The equation states for Q^* that

$$Q^*(s, a) = \sum_{s', r} P(s'|s, a)[r + \gamma \max_{a'} Q^*(s', a')] \tag{1}$$

where $\gamma \in (0, 1]$ is a discount factor.

To find the optimal policy, we would need to evaluate all possible states and actions in iterations to find a converging solution. This would require knowledge of reward function R and probability distribution P. Apart from that, the action and state-space would need to be finite and reasonably small as keeping the Q-value table would require $|S| \times |A|$ entries. If these conditions were met, we could find the optimal solution using Value- and Policy-iteration algorithms that iteratively update the Q-table and policy to obtain the optimal solution in the Bellman sense.

There are different algorithms used for unknown MDPs that learn Q-values by sampling the search space.

These methods are based on the temporal difference, which stands for a difference in estimated Q-values in consequent time steps. The methods are either on-policy, meaning that the current policy is used to select an action for the next state s', or off-policy, meaning that action maximising the value of the next state is selected, ignoring the current policy.

Q-learning is an example of an off-policy algorithm with the one-step update

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)] \tag{2}$$

SARSA is an example of an on-policy algorithm with the one-step update

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)] \tag{3}$$

where a' is an action chosen by policy π.

To learn the Q-values properly, it is necessary to follow an exploration policy that satisfies GLIE properties [73]. GLIE stands for greedy in the limit (in the limit, the policy should choose the maximising action) and infinite exploration (state-action pairs are visited an infinite number of times). In practice, this is ensured by using ϵ-greedy policy that chooses the maximising action with probability $1 - \epsilon$ and random action with probability ϵ where ϵ decreases with the number of iterations.

In this thesis, we work with a deterministic environment with continuous state space, discrete action space and we use a method based on Q-value estimation. The presented approach should work for stochastic environments and discrete state space as well, whereas using continuous action space would require a modification of the local search part of the algorithm.

3.2 Deep Q-learning Networks

Deep Q-learning Networks is an algorithm [8] designed to be able to leverage Q-learning capabilities in huge state space. To do so, the authors approximate the Q-table by a Q-function that is represented by a neural network.

The algorithm (see Algorithm 4) utilises experience replay where one experience is a tuple (s_t, a_t, r_t, s_{t+1}). Each experience is stored in a replay buffer D, which serves as a dataset to be fed to the neural network by random sampling in mini-batches. The real value y is set to the Q-value estimated by the neural network. The target value y' corresponds to the one-step

lookahead taking into account the next state Q-value and the obtained reward. The gradient step is performed based on the mean square error of y and y'. To make the algorithm more stable, two different networks Q-network Q and Q-target-network $\hat{Q}$ are used. To simplify the notation, in the sequel, we use Q-target for Q-network-target. Q represents the current solution, is used to generate y and is being updated by the gradient step. $\hat{Q}$ is an older version of Q that is used to generate y'. $\hat{Q}$ is updated every C steps by the current version of Q.

Algorithm 4 DQN pseudocode adapted from [74]

Initialize replay buffer D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with random weights $\theta^- = \theta$
for episode $= 1, M$ **do**
    Initialize sequence s_1
    for $t = 1, T$ **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$
        Execute action a_t in emulator and observe reward r_t and state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in D
        Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
        Set $y_j = r_j$ if episode terminates at step $j + 1$
        Set $y_j = r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-)$ otherwise
        Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to θ
        Every C steps reset $\hat{Q} = Q$

The method leverages the following:

- Data efficiency: The classical Q-learning generates a data sample from the environment for each Q-value update (as 1:1 mapping of sample to update). On the contrary, DQN uses a replay buffer D that is gradually being filled by data samples. When performing an update, a random batch of data is sampled from the replay buffer. This allows to use each data sample more times (as 1:N mapping of sample to update) and is especially beneficial for cases when the environment evaluation is costly.
- Sample randomisation: Learning from consecutive samples is inefficient because the samples are correlated. By sampling from the replay buffer D, we can get data from several different episodes, making the training faster as the sample distribution is more representative.
- Training stability: DQN uses Q as a policy, and $\hat{Q}$ as a target policy. $\hat{Q}$ is a fixed target that is updated every C time steps by the current parameters from Q. In the meantime, only Q is being modified by the calculated gradients, which improves convergence as $\hat{Q}$ is a fixed, stable target. If Q was used for target calculation instead of $\hat{Q}$, the target value would change for each update, and the algorithm could start oscillating.

The authors have tested the algorithm on various Atari 2600 games and achieved a human-level performance. The agent was receiving high-dimensional preprocessed frame pixels and game scores on the input. Moreover, the hyperparameters and architecture were fixed for all

the different game types. A disadvantage of DQN is that it produces a finite-length vector where each feature in the vector corresponds to one action in the action space. This limits the action space to be a discrete finite set.

3.3 Testing platforms

The algorithm is to be evaluated in the OpenAI gym environments or its alternatives. OpenAI gym [58] is an open-source catalogue of RL environments. The advantage of this library is that it has a very clean API, is well-tested and used by many researchers, which is convenient for performance benchmarking. There are five main categories of environments in OpenAI gym (some of them are dependent on the MuJoCo simulator described below):

- Atari – 59 different Atari environments for Atari 2600 video game console
- Box2D – continuous control tasks in the Box2d simulator
- Classic control – control theory problems from the classic RL literature like the cart pole or inverted pendulum
- MuJoCo – continuous control tasks such as humanoid or 4-legged robot
- Robotics – goal-based tasks for the Fetch and ShadowHand robots in MuJoCo simulator

MuJoCo [75] is a physics engine for fast and accurate simulation, mainly used in robotics and biomechanics domain. The simulator allows defining custom environments. The humanoid environment (also defined in OpenAI gym) is often used to test RL tasks; see, for example, this video `https://youtu.be/iJlEbHsgM7Q`. However, the usability of the simulator is limited as it is licensed, with a 30-day free trial or a free license for students.

Robogym [76] is an open-source library that provides a wrapper Python API that is the same as in OpenAI gym to run several different environments in MuJoCo. There are two types of environments. Firstly, Dactyl environments where the goal is a robotic hand that has 20 actuated degrees of freedom to manipulate a Rubik's cube. Different complexity levels are utilised by restricting the degrees of freedom of the Rubik's cube. Secondly, Rearrange environments where the goal is to use a robotic arm with a gripper to set up items on a table in a requested way.

An interesting alternative to OpenAI gym that heavily depends on the licensed MuJoCo simulator is PyBullet Gymperium [77]. It is an open-source reimplementation of the OpenAI Gym MuJoCo environments, and Roboschool environments [78] based on the Bullet Physics simulator. An advantage of this package is that the environments are defined using the OpenAI gym API. There are also plenty of other RL environments one could use for experiments. I found the list at `https://github.com/clvrai/awesome-rl-envs` to be the most comprehensive.

In this thesis, we use the classic control environments with discrete action space from OpenAI gym package for experiments to keep the thesis open-source while considering the fact that the usage of the DQN algorithm implies the action space to be discrete.

4 Proposed Approach

This section describes the proposed approach that combines the evolutionary algorithm HyperNEAT with the gradient-based algorithm DQN. The objective is to develop an algorithm that is able to train an agent that performs well in a given reinforcement learning environment. Firstly, we describe the modifications we made in the baseline HyperNEAT algorithm [7]. Secondly, we describe the specifics of the used DQN implementation. Lastly, we demonstrate how we combine the two algorithms by showing how to initialise DQN by HyperNEAT and vice-versa. We discuss the effects of each modification in Chapter 6. The high-level overview of the algorithm workflow is shown in Figure 8.

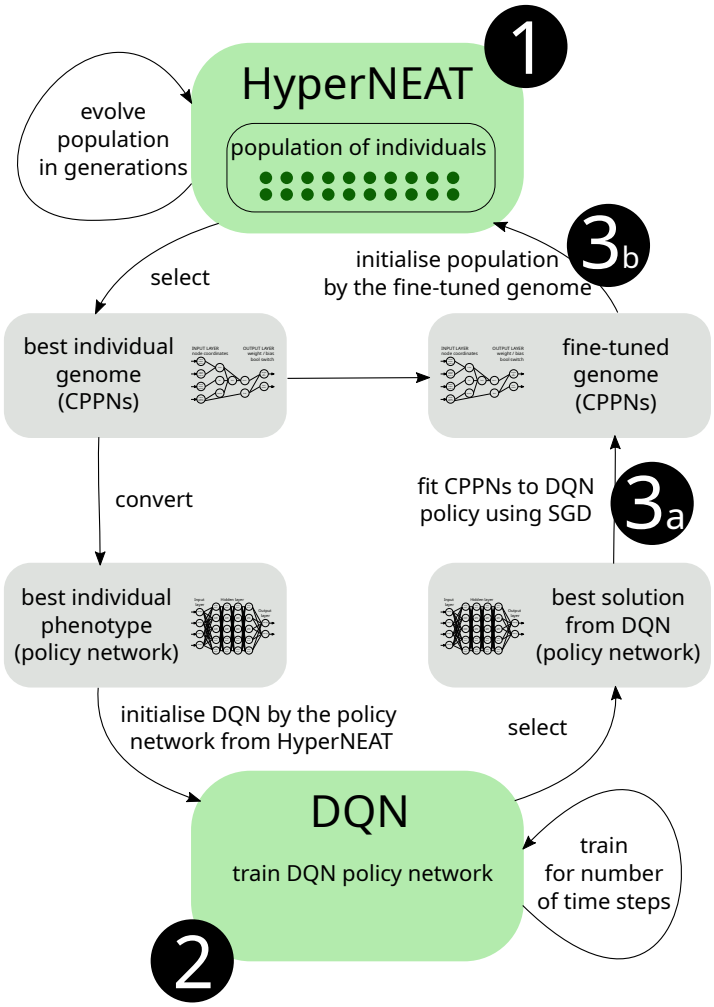


Figure 8: Structure of the algorithm in 3 steps. Firstly, CPPNs generating HyperNEAT policy network are produced by the HyperNEAT algorithm. Secondly, the HyperNEAT policy network initialises the DQN algorithm. Lastly, the DQN policy network is used to fine-tune the original HyperNEAT CPPNs by sampling training data from the DQN policy and fitting it to the CPPNs using SGD. The fine-tuned CPPNs are converted to a genome that is then recombined with genomes of the original HyperNEAT population. The algorithm iterates until the terminal condition holds.

In the sequel, we use the following terminology:

- *Genome* stands for one individual of the HyperNEAT population.
- *CPPN* is a neural network that produces weights and/or biases of the final neural network.
- *HyperNEAT policy network* is the final neural network that is generated by possibly multiple CPPNs for a given fixed-size substrate (coordinate grid); the policy network accepts a state and outputs a vector of values for each action.
- *DQN policy network* is a neural network that is trained by DQN; the network accepts a state and outputs a vector of values for each action.
- *Fine-tuned CPPN* is a CPPN trained on a data set extracted from the DQN policy network.
- *Fine-tuned genome* is a genome that encodes the fine-tuned CPPN(s).

4.1 HyperNEAT realisation

HyperNEAT is an evolutionary algorithm that trains a population of individuals where each of the individuals is a genome representing CPPN(s) that generates neural network parameters. The neural network accepts the current state of the environment and outputs a value for each available action. The agent selects the action with the highest value, performs it in the simulator and receives a reward and its next state. The individuals in the population are evaluated based on how well on average the agent performs in the simulation.

We use the original version of the algorithm [7] with the following assumptions – the size of the substrate is fixed to 4 hidden layers and 32 neurons in each. The size was set experimentally so that the number of neurons in one layer is a factor of two, and both HyperNEAT and DQN can train the policy network. The size of the substrate could be further optimised, but that is out of the scope of this thesis. Another assumption is made on the connectivity of the underlying substrate – we work with feed-forward fully connected networks. This is motivated by speeding up the implementation as we can group the nodes into layers and leverage matrix multiplication to evaluate the network. We use Leaky ReLU [79] as the activation function in the hidden nodes and linear activation (i.e., no activation) in the output nodes. See the used policy network architecture in Figure 9.

We diverge from the original implementation by testing different genome types. In the original implementation, each individual is represented by one CPPN that generates weights for the network. Each weight is accepted if it lies above a specified threshold and set to zero otherwise. No bias is used in the policy network. Our genome types differ in the following:

1. We disregard the weight threshold to use fewer hyperparameters.
2. We modify the genome structure to be able to generate both weights and biases.

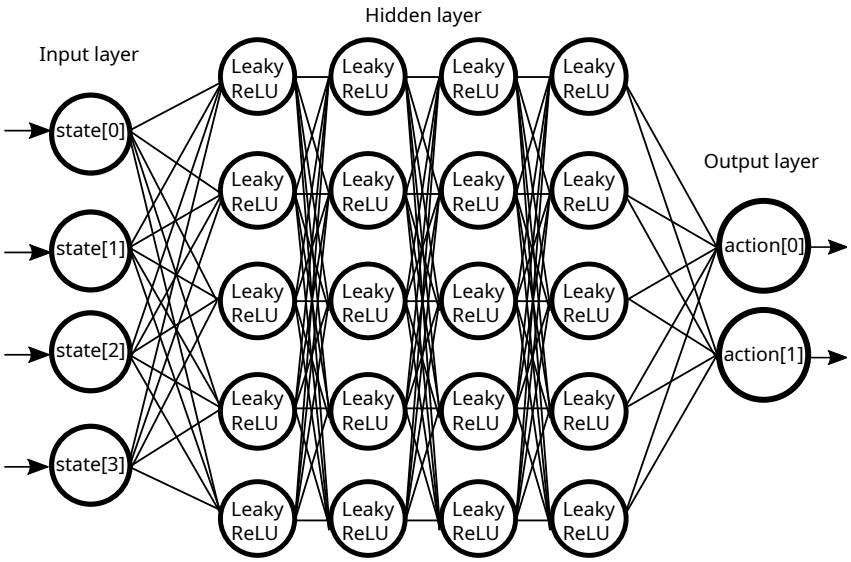


Figure 9: Policy network architecture used for HyperNEAT as well as for DQN. The parameters of the network (weights and biases) are generated by evolvable CPPN in HyperNEAT or trained by stochastic gradient descent in DQN. This example of the policy network accepts a state with four features and outputs a value for a discrete action space of size two. The input/output size can vary for different environments.

The first modification adds one more output to the CPPN network. This additional output is a boolean value that says whether we should use the weight or set it to zero and is implemented as $\frac{1}{1+e^{(-x)}} > 0.5$ where x is the input signal of the newly created node. This modification is inspired by [44]. An example of the CPPN network is shown in Figure 10.

The second modification changes the genome so that it generates bias as well and comes in two versions:

- The genome structure stays the same, but we use the represented CPPN to generate biases. To do so, we use the same CPPN as for weights and input the queried node twice. Considering the original usage of CPPN to be $w_{i,j} = CPPN(node_i, node_j)$, we use the same CPPN as $b_i = CPPN(node_i, node_i)$, where $w_{i,j}$ stands for the weight between the i-th and j-th node, and b_i stands for the bias of the i-th node.
- We change the genome to represent two CPPNs instead of one. The first CPPN corresponds to the original CPPN and generates weights only. The second CPPN generates biases only. This modification is inspired by [46]. The genetic operators working over the new genome are equivalent to a nested version of the original operators, i.e. when performing crossover of genomes $g_1 = [s_1^1, s_1^2]$ and $g_2 = [s_2^1, s_2^2]$, we perform the crossover on the corresponding subgenomes (a subgenome s_i^j represents the j-th CPPN of genome i) to get:

  $$cross(g_1, g_2) = [cross(s_1^1, s_1^2), cross(s_2^1, s_2^2)] = [s_3^1, s_3^2] = g_3 \quad (4)$$

  Other operations over the genome, such as mutation or initialisation, are performed analogously.

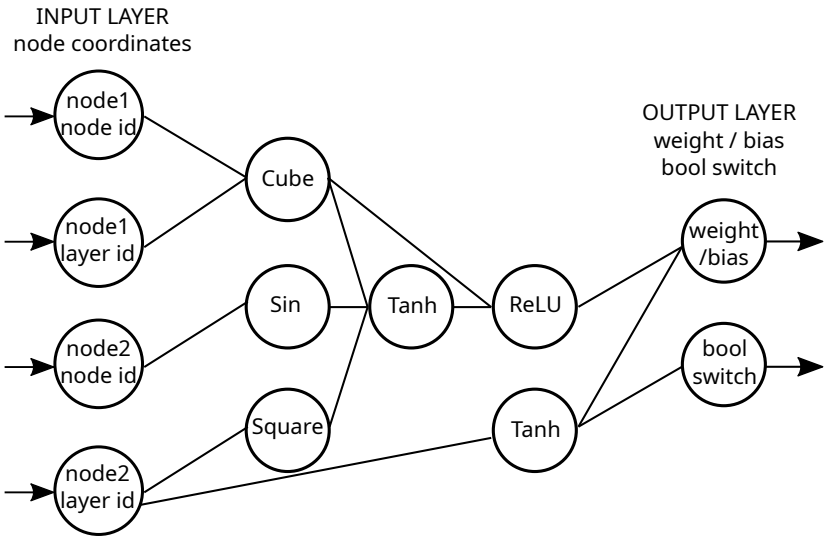


Figure 10: An example of a CPPN. The inputs of the CPPN are coordinates of two nodes in the underlying substrate. The output of the CPPN is the weight (or bias) and the boolean switch saying whether we should use the weight/bias or not. The hidden nodes of the CPPN network use an activation function selected from a predefined set of functions. The structure of the CPPN is evolved by the NEAT algorithm.

4.2 Initialising DQN by HyperNEAT

We have used a DQN implementation from Stable Baselines 3 [80] that differs from the original pseudocode (Algorithm 4) in the following:

- The two nested for-cycles are reduced into one while loop that runs over the total number of time steps performed in the algorithm, instead of running for M episodes, each with T time steps.
- Instead of collecting one sample (time step) each iteration, we can collect up to R samples, where R stands for rollout size. Each collected sample is added to the replay buffer and updates the total counter of the performed steps.

The pseudocode containing these changes is provided together with the other described modifications in Algorithm 5.

Combining HyperNEAT and DQN is a challenging task as the workflow of the algorithms is very different, and more importantly, the outputs of the policy networks of the two algorithms have a different interpretation:

- HyperNEAT outputs a vector of values, the action with the maximal value is selected.
- DQN outputs a vector of Q-values, the action maximising the Q-value is selected.

Additionally, the Q-values are interpretable as "expected reward if we select given action in the state we are currently in". HyperNEATs output values do not have such an interpretation and they cannot be considered Q-values that satisfy the Bellman equation 1. We examine three options on how to initialise DQN using a (suboptimal) policy network from HyperNEAT.

Algorithm 5 Deep Q-learning pseudocode with all of the proposed modifications. The initialisation options are not necessarily used at the same time. If initialisation option 1) or 2) is not used, the initialisation is random instead.

Initialize action-value function Q with policy H from HyperNEAT ▷ Init 1)
Initialize target action-value function $\hat{Q}$ with policy H from HyperNEAT ▷ Init 1)
Initialize replay buffer D to capacity N by the ϵ_1-policy H from HyperNEAT ▷ Init 2)
Initialize sequence s_1
while timestep $< T$ **do**
    for rollout $= 1, R$ **do**
        With probability ϵ_2 select a random action a_t
        otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$
        Execute action a_t in emulator and observe reward r_t and state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in D
    timestep $\leftarrow$ timestep $+ R$
    Sample random minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
    if timestep $< T_H$ **then**
        Set $y_j = r_j$ if episode terminates at step $j + 1$
        Set $y_j = r_j + \gamma Q(s_{j+1}, \arg\max_{a'} H(s_{t+1}, a'); \theta)$ otherwise ▷ Init 3)
    else
        Set $y_j = r_j$ if episode terminates at step $j + 1$
        Set $y_j = r_j + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-)$ otherwise
    Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$ with respect to θ
    Every C steps reset $\hat{Q} = Q$

Option 1: Use policy network from HyperNEAT as the initial policy network for DQN

We run the HyperNEAT algorithm (step 1 in Figure 8) and take the best performing individual (genome). We use the CPPNs encoded by the genome to generate a HyperNEAT policy network. We copy the weights and biases from the HyperNEAT policy network and set them as initial parameters to DQN policy network (step 2 in Figure 8). We can do that because the DQN policy network has the same architecture as the HyperNEAT policy network (shown in Figure 9).

Firstly, we set both Q-network (Q in Algorithm 4) and Q-target ($\hat{Q}$ in Algorithm 4) to the initial solution. Q-network is the DQN policy network. The hypothesis is that the HyperNEAT policy network already performs well, and DQN would help to fine-tune it by evolving it so that it produces the true Q-values.

Secondly, we relax the initialisation and set Q-target only, while Q-network stays initialised by the standard PyTorch random initialisation $\mathcal{U}(-\sqrt{k}, \sqrt{k}) : k = 1/in\_feat$. Starting with the random initialisation has already proven to work well. By initialising the target network,

during the first C iterations, the DQN policy network (Q-network Q) is pushed towards the HyperNEAT policy network. The advantage against the previously mentioned full initialisation is that if the HyperNEAT policy network is very off with the produced values that are not the true Q-values, the Q-network (DQN policy network) would not be skewed correspondingly.

Option 2: Use HyperNEAT policy network to prefill replay buffer of DQN

Replay buffer is the database of samples (D in Algorithm 4). We use the suboptimal policy network found by HyperNEAT (step 1 in Figure 8) to prefill the replay buffer (step 2 in Figure 8). This aims to speed up the training by providing more educative samples as the suboptimal HyperNEAT policy network explored more advanced states than the randomly initialised DQN policy network. A disadvantage of this approach is that by after learning only from the "nice" samples (i.e. samples on the path taken by the suboptimal HyperNEAT policy network), the DQN policy network would not know how to react in the border cases. Hence, when filling the buffer, we use an ϵ-policy that with probability ϵ uses random action and with probability $(1-\epsilon)$ uses action selected by HyperNEAT policy network. This is a typical problem of balancing between exploration and exploitation.

Option 3: Use HyperNEAT policy network as an external policy for DQN training

DQN algorithm is an off-policy algorithm. This means that when we calculate the loss, the target Q-value is not calculated using the current DQN policy represented by Q-network Q, instead it is determined by the maximising action using the Q-target network $\hat{Q}$, see (6). Based on Algorithm 4, the Q-values given sample (s_t, a_t, r_t, s_{t+1}) are as follows:

$$y = Q(s_t, a_t) \tag{5}$$

$$y' = \begin{cases} r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'), & \text{if } s_{t+1} \text{ non-terminal} \\ r_t, & \text{otherwise} \end{cases} \tag{6}$$

where y is the curent Q-value (left side of the Bellman equation (1)) and y' is the target Q-value (right side of the Bellman equation (1)). We use the suboptimal policy network from HyperNEAT (step 1 in Figure 8) as external policy H (step 2 in Figure 8) to select next action a_{t+1} for the y' calculation as follows:

$$y = Q(s_t, a_t) \tag{7}$$

$$y' = \begin{cases} r_t + \gamma Q(s_{t+1}, \arg\max_{a'} H(s_{t+1}, a')), & \text{if } s_{t+1} \text{ non-terminal} \\ r_t, & \text{otherwise} \end{cases} \tag{8}$$

To sum it up, for the first T_H timesteps, which is a user-defined parameter of this strategy, we disregard Q-target $\hat{Q}$ and use Q-network only. We do that because we would not use the $\hat{Q}$ to select the maximising action, which was its intended usage in the first place (see 3.2 for more details). Instead of selecting the action maximising the Q-value, we select the action maximising the HyperNEAT policy network output. After T_H iterations, we stop using the external policy H and return to the original y and y' calculation to allow the DQN policy network to evolve beyond the capabilities of the original HyperNEAT policy network.

All of the options are summarised in Algorithm 5. The initialisation options are not necessarily used simultaneously; in that case, the default random initialisation is used instead of the skipped modification.

4.3 Initialising HyperNEAT by DQN

We use the pre-trained HyperNEAT genome G with the final HyperNEAT population P, the trained DQN policy network Q and propagate parameters of Q into G. This corresponds to steps 3a and 3b in Figure 8.

Firstly, we describe step 3a (Figure 8). To recapitulate, G is a genome representing two CPPNs. The CPPNs are two neural networks, g_w parametrised by θ_w and g_b parametrised by θ_b that generate weights, and biases for the HyperNEAT policy network H. DQN policy network Q has the same architecture as the HyperNEAT policy network H. Hence, we can use the DQN policy network Q as a training set $T_w = \{(x_i, y_i)\}$ where x_i are coordinates of two connected neural network nodes, and y_i is the weight of the connection. The size of $|T_w|$ corresponds to the number of connections in the network. Similarly, we create the bias training set T_b.

We use the training sets T_w and T_b to train parameters θ_w and θ_b by stochastic gradient descent (see Algorithm 6). After each epoch, a new genome G' is created based on the updated parameters θ_w and θ_b. During the training, we monitor the fitness of G' and we stop the training if the fitness $f_{G'}$ decreases below a threshold defined as $q \cdot f_G$, where the backpropagation quality $q \in [0, 1]$ and f_G is the fitness of G.

Algorithm 6 SGD pseudocode for backpropagation of DQN solution Q into HyperNEAT genome G. **IN:** genome G, DQN policy network Q. **OUT:** fine-tuned genome G'

set θ_w, θ_b to parameters of weight and bias CPPN of genome G
generate datasets T_w, T_b from the DQN policy network Q
calculate fitness f_G of genome G
for *epoch* $\in [1, \ldots, E]$ **do**
    for T, $\theta, g \in \{(T_w, \theta_w, g_w), (T_b, \theta_b, g_b)\}$ **do**
        shuffle T
        for minibatch of size M containing $\mathbf{x^i}, y^i \in T$ **do**
            get predictions $\hat{y}^i \leftarrow g(\mathbf{x^i})$
            get loss $L \leftarrow loss(\hat{Y}^m, Y^m)$
            get gradients $\Delta\theta \leftarrow -\nabla_L \theta$
            make gradient step $\theta \leftarrow \theta + \alpha \cdot \Delta\theta$
    calculate fitness $f_{G'}$ of genome G' recreated from θ_w and θ_b
    if $f_{G'}/f_G < q$ **then**
        break
return fine-tuned genome G'

Secondly, we describe step 3b (Figure 8). We take the newly created genome G' and recombine it with each member of the original HyperNEAT population P. We chose to use the original population P instead of a randomly initialised population to preserve the historical markings that guide the reproduction in the NEAT algorithm. We continue the training by iterating HyperNEAT and DQN as shown in Figure 8.

5 Implementation

The proposed approach is implemented in Python 3.8 and PyTorch 1.7. The project dependencies are managed by the Conda environment management system. The Conda environment specifications can be found in `configs/environment.yml`. To install the environment run `conda env create --name env-name --file=environment.yml` and then `conda activate env-name`.

We have used an external library for NEAT algorithm [49] which we extended by new classes `MultiGenome` and `MultiReproduction` that allow one genome to represent more than one CPPN and we implemented the corresponding operators such as crossover. We built our custom implementation of HyperNEAT on top of the NEAT library. Our HyperNEAT implementation has been partially inspired by [50]. However, the HyperNEAT library mentioned above did not contain all features that we wanted to use (e.g. CPPN backpropagation), so we decided for a custom implementation instead. See the implementation in module `algorithms/hyperneat_classes` and `hyperneat.py`. Our implementation of both the HyperNEAT policy network and CPPN network is PyTorch-based.

Regarding the DQN algorithm, we used the implementation from Stable Baselines 3 [80]. We have extended the implementation to be able to use external policy as described in Section 4.2. The implemented extensions can be found in module `dqn_extensions`.

All of the tested approaches are implemented with a unified interface in files `dqn.py`, `hyperneat.py`, `hyperneat_dqn_joined.py` and `hyperneat_dqn_loop.py` respectively. The code contributions of the thesis are the following: We have designed and implemented the genome representation of CPPN, we implemented converter from the genome to CPPN where CPPN is a fully working PyTorch module. We implemented a converter that creates a feed-forward policy network from the given CPPN and substrate; this policy network is also implemented as a PyTorch module. Implementation of the policy network as a PyTorch module allows easy transfer of the HyperNEAT policy parameters to the DQN policy network. We have implemented buffer initialisation for DQN and subclassed DQN class from Stable Baselines 3 to implement the training loop that uses external policy to guide the search. We have implemented the backpropagation of weights and biases from DQN policy to CPPN network, conversion of the CPPN network back to genome representation and initialisation procedure reproducing the fine-tuned genome into the new population.

The algorithm hyperparameters are passed via a command line or a `yaml` configuration file. The hyperparameters are handled by a dataclass defined in `utils/experiment_config.py`. An example interface to run experiments is provided in `experiment.py`. All of the four algorithm scripts also have a main method for local testing, which is a good starting point to test the project.

The implementation can be found at `https://gitlab.com/katerinab/diploma-thesis`.

6 Experiments

This chapter describes the tested reinforcement learning problems and scenarios, the performance measures and hyperparameters of the algorithms. We present the experimental evaluation and discuss the results. We test the performance of the following algorithms:

1. HyperNEAT – custom PyTorch implementation build upon NEAT [49]
2. DQN – Stable Baselines 3 implementation [80]
3. HyperNEAT→DQN – evolving policy by HyperNEAT and fine-tuning it by DQN (steps 1 to 2 in Figure 8)
4. HyperNEAT→DQN loop – evolving policy by HyperNEAT, fine-tuning it by DQN, backpropagating the DQN policy to CPPNs and initialising a new HyperNEAT population (iterated steps 1, 2, 3 in Figure 8)

6.1 Test problems and tested scenarios

We used three classic control problems defined in OpenAI gym [58]. Cart pole and Acrobot have been used for testing the initialisation options of HyperNEAT and DQN (see Sections 4.2 and 4.3). Mountain car has only been used for the final evaluation.

6.1.1 Cart pole

Cart pole problem was defined in [81], and it has been used in the reinforcement learning literature ever since as a simple problem to test the algorithm [72], [82]. The system consists of a cart with an attached vertically placed pole. The agent's goal in the environment is to balance a moving cart so that the pole does not fall.

The state is defined as x-position of the cart, velocity of the cart, pole angle relative to the upright position and pole angular velocity. Available actions are: push the cart to the left and push the cart to the right. The agent gets a positive reward 1 for each step that the pole stays upright. The episode terminates if the pole angle is more than 12 degrees, the cart reaches the edge of the display or the episode length is larger than 200. The environment is considered to be solved if the average return reward over 100 independent runs is 195 or above. The return reward of the j-th independent run R_j is defined as a sum of rewards r_t collected in one episode (i.e., during at most 200 time steps t).

6.1.2 Acrobot

Acrobot is a two-link inverted pendulum with an actuator at the elbow, but no actuator at the shoulder. The state corresponds to $[\cos(\theta_1), \sin(\theta_1), \cos(\theta_2), \sin(\theta_2), \overrightarrow{\theta_1}, \overrightarrow{\theta_2}]$ where θ_i are

the angles of the Acrobot joint and $\overrightarrow{\theta_i}$ are the corresponding angular velocities. Available actions are to apply torque -1, 0 or 1 to the second joint (the joint between the two links).

The agent gets a negative reward of -1 for each time step it stays in the simulation until reaching the goal state. The goal is to swing the lower link to a specified height. The environment does not have a defined reward threshold where it is considered to be solved. Based on the OpenAI gym leaderboard [83], we set the threshold to be -60, while considering the reward of -80 to be a good performance. The problem is defined in [84].

6.1.3 Mountain car

Mountain car problem [85] is another standard testing domain where a car starts on a one-dimensional track placed in a valley between two hills. The goal is to drive the car up to the right hill. The problem is that the engine of the car is not strong enough to push the car up the hill in a single pass. Instead, the car has to drive back up the left hill and use its potential energy to help it driving up the right hill.

The state is defined as the car's x-position and velocity. The available actions are: accelerate to the left, accelerate to the right or do not accelerate. The agent gets a negative reward -1 if the car has not reached the goal state and reward 0 otherwise. The episode terminates when the car has reached the goal state (position on the top of the right hill), or the length of the episode is more than 200. The environment is considered solved if the average return reward over 100 independent runs is above -110 (i.e. the car reaches the goal state in at most 110 time steps).

6.2 Performance evaluation

When evaluating the performance of the HyperNEAT- and DQN-based approaches, we can track:

- the number of data samples used until convergence,
- the number of performed episodes until convergence,
- the quality of the solution when the algorithm converges,
- the runtime until convergence,
- the number of different solutions that the algorithm explores before convergence,
- the probability the algorithm converges to a given performance level in a given time,

What makes the evaluation difficult is the fundamental difference between the two algorithms; see Table 1. We do not compare the algorithms based on the runtime because it is implementation-dependent. Though, the general observation is that HyperNEAT is faster as

it is easily parallelisable. In HyperNEAT we can evaluate the individuals (i.e. different candidate solutions) in parallel while in DQN the candidate solution is updated sequentially. Due to this, we can observe that when given the same limit of time steps, the runtime of HyperNEAT is lower than the runtime of DQN. We also do not compare the algorithms based on the number of episodes as the DQN implementation that we are using does not utilise it as a relevant time unit.

Feature	**HyperNEAT**	**DQN**
Training loop	Evolves a population in generations	Trains the solution in time steps
Evaluation	Evaluates each individual by fitness function averaging the return reward over a given number of episodes	Tracks the solution progress based on the loss calculated from actual and desired Q-value
Data samples	Uses a lot of data samples	Uses only a few data samples (from the replay buffer)
Solution changes	Makes less updates (one solution for each individual in each generation)	Makes more updates (one solution for each gradient step)
Exploration	Larger difference between two consecutive populations as the algorithm recombines different solutions	Smaller difference between the two consecutive solutions, based on the gradient

Table 1: Main differences in the workflow of DQN and HyperNEAT that make the algorithms difficult to compare

When comparing the algorithms we use *time steps* as a time unit. One time step corresponds to one data sample queried from the environment. For HyperNEAT, time steps are 'consumed' each time a fitness function of an individual is evaluated, where fitness function is:

$$fitness(ind_i) = \frac{1}{V} \sum_{j=\{1,\dots,V\}} R_j \tag{9}$$

where V is the number of different episodes and R_j is the return reward of the j-th independent episode that is limited to T_E timesteps. The HyperNEAT algorithm evolves population of P individuals in G generations. In total, it consumes T_{HN} timesteps, where

$$T_{HN} \leq G \cdot P \cdot T_E \cdot V \tag{10}$$

The inequality covers the cases where the episode terminates before reaching its maximal T_E timesteps. This happens in the Cart pole environment when the pole falls and in the Acrobot and Mountain car environments when the goal position is reached.

For DQN, the time steps are consumed each time new samples are generated into the replay buffer. This happens in two situations. Firstly, when randomly generating T_I initial samples into the replay buffer. Secondly, in each of the I iterations, when the algorithm performs a rollout to collect T_R new samples to the replay buffer and samples a batch of B samples from the replay buffer to perform the gradient step. In total, DQN consumes T_{DQN} timesteps, where

$$T_{DQN} = T_I + I \cdot T_R \tag{11}$$

We can notice that if we set the total number of time steps to T_{DQN} and fix the initial replay buffer size T_I, the number of iterations I depends on the rollout size T_R. This is an

important observation as a gradient update is performed based on B samples in each of the training iterations. Therefore, by modifying T_R, we are indirectly influencing the total number of effective (repetitive) samples used for the training $I \cdot B$. Since DQN does not internally track the current fitness (episodal return reward), we evaluate it externally by evaluating the solution for every $V \cdot T_E$ time steps. This kind of evaluation corresponds to the frequency of HyperNEAT individual evaluation. However, since DQN is guided by the training loss and not by the fitness, we do not include the DQN evaluation time steps to calculate T_{DQN}.

We measure the performance of the algorithms by fixing the total number of samples T_{max} that can be used for the training. We evaluate the solutions found after T_{max} time steps and measure their final return reward over V episodes. We stop the training if the logged average (over V episodes) return reward exceeds the threshold where the environment is considered to be solved (195 for Cart pole, -110 for Mountain car, undefined for Acrobot). In case the algorithm solves the environment, we also measure the number of time steps it consumed. Note, that we relax the 'solve' condition to be defined as the average over V independent evaluations, not the original 100 independent evaluations as that would to be too time consuming. Finally, we analyse the average return reward throughout the training and compare the speed of convergence.

Even though we tried to design the evaluation to be as fair as possible, it is not perfect. For example, while keeping the same T_{max} for DQN, we could modify the rollout size T_R, and the algorithm would effectively perform more gradient updates which would probably result in a better solution. Similarly, for HyperNEAT, while fixing the same T_{max}, we could say that we do not mind inaccurate fitness evaluation and decrease V, the number of different episodes for fitness evaluation. This would effectively increase the number of training generations and probably result in a better solution.

Hence, we do not spend too much time optimising the hyperparameters for our specific reinforcement learning problems, and when comparing the approaches, we focus on comparing the relative performance instead of the absolute performance. For example, we run HyperNEAT→DQN with one of the proposed initialisation options, see Section 4.2. Both the DQN and the HyperNEAT parts of the HyperNEAT→DQN approach get a budget of T_{max} time steps. We evaluate the final solution. For absolute comparison, we compare the results with the baseline DQN that runs for T_{max} time steps. For relative comparison, we compare the results with the baseline DQN that runs for $T_{max}/2$ time steps, as then we can observe whether the initialisation had a positive effect on the performance of DQN.

When considering the absolute comparison, limiting the number of total time steps seems to be more beneficial for the DQN algorithm as it is designed to be sample-efficient and is able to use the whole budget of T_{max}. On the contrary, HyperNEAT is not sample-efficient and is often not able to use the whole budget of T_{max} as the episodes can terminate earlier when it fails. Though, using time steps as a time unit makes sense as querying the update from the environment is often the most expensive operation, especially if considering more complex environments than we are using in this thesis.

We have to consider that while the environments are deterministic, the starting state of the environment is randomised. Hence, if our agent is not stable enough, it may obtain a high return reward when evaluated in an episode with starting state s_1 and a low return

reward when evaluated in an episode with starting state s_2. Additionally, both of the algorithms are randomised in the initial initialisation. To compensate for that, we always evaluate the algorithms over several independent runs with the same hyperparameters, just a different random seed of the algorithms. More specifically for one algorithm run, we consider two different types of random generators: random generator g_A is used by the algorithm and is initialised by a given seed $seed_A$ (that is passed as an argument), random generator g_E is used by the RL environment, and is initialised at the beginning of each fitness evaluation by setting seed $seed_B \leftarrow next(g_A)$. By making g_E dependent on g_A, we ensure both the environment is different in each fitness evaluation and that the experimental algorithm evaluation is reproducible.

6.3 Configurations

Both DQN and HyperNEAT have several hyperparameters that need to be set. Moreover, a few additional hyperparameters arise from the usage of the combined methods with different initialisation techniques. We summarise the hyperparameters in Table 2 for HyperNEAT and in Table 3 for DQN. In these tables, we name each hyperparameter, briefly describe it, and we either state its value or present its default value and refer to the experiments where this value is modified.

In the following text, we use the following notation for different genome types: prefix CPPN(w+b) for genome using only one CPPN generating both weights and biases; prefix CPPN(w,b) for genome using two different CPPNs for weights and biases; prefix CPPN(w) for genome with one CPPN generating only weights; suffix _1 says that the CPPN has only one output, _2 says that the CPPN has two outputs as described in Section 4.1.

Parameter	**Value**	**Description**
Total generations G	200	The total number of training iterations for evolving the population of individuals
Population size P	20	The number of individuals in a population
Validation episods V	10	The number of independent episodes to average in the fitness evaluation
Max. episode steps T_E	200	The max. number of time steps in one episode
Layers in substrate	5	The number of layers in the substrate
Nodes in one layer	32	The number of neurons in a substrate layer
Output activation	Linear (identity)	Activation on the output layer
Hidden activation	Leaky ReLU	Activation on the hidden layers
Genome types	CPPN(w,b)_2 changes in 6.4.1	Genome type corresponding to one of the 6 options presented in Section 4.1

Table 2: The most important hyperparameters of HyperNEAT, NEAT-specific hyperparameters such as mutation probability or weight restrictions, can be found in Appendix E with the description at `https://neat-python.readthedocs.io/en/latest/config_file.html`.

Parameter	Value	Description
Train iterations T_{max}	8000000	The total number of data samples the algorithm can use
Rollout size T_R	20	The number of data samples collected in one iteration
Batch size B	64	The number of data samples used for the gradient update in one iteration
Discount factor γ	0.9	Discount factor from Bellman equation 1
Target update freq. C	10000	Every C steps policy network Q is copied to the target policy network $\hat{Q}$
Initial buffer size T_I	50000, changes in 6.4.3	The size of the replay buffer that is initialised before the training starts
Replay buffer epsilon ϵ	0.2, changes in 6.4.3	If buffer initialisation is used, ϵ is the ratio of random samples
Ext. policy threshold T_H	500000, changes in 6.4.4	If external policy is used, T_H is how many initial time steps it should be used
CPPN backprop quality q	0.9, changes in 6.4.6	If the loop version is used, q is how much the solution quality can decrease until the backpropagation is terminated
CPPN backprop epochs E	20	If the loop version is used, E is how many epochs the backpropagation runs
Loop repetitions	2,5,10 or 20, changes in 6.4.6	If the loop version is used, it says how many iterations is performed

Table 3: The most important hyperparameters of DQN (the first 6 parameters for the baseline DQN) and the proposed approaches. Hyperparameters such as max. replay buffer size can be found at `https://stable-baselines3.readthedocs.io/en/master/modules/dqn.html`.

6.4 Results

All of the experiments were run on the CIIRC computational cluster. All of the experiments were run with the maximal number of time steps T_{max} limited to 8000000, see details for HyperNEAT and DQN algorithms in Tables 2 and 3. In the case of the combined methods, we split the timestep budget between all algorithm parts uniformly so that the total number of timesteps summed over all algorithm parts is at most T_{max}. Each experimental settings is run several times, and we measure the mean or median performance.

Generally, the higher the number of independent runs is, the more accurate performance metrics we get. As the experiments are computationally heavy, we set the number of runs to 10 only for the initial experiments. For the final experiment, we select the best-performing methods, baseline DQN and HyperNEAT, and we evaluate the performance of these methods over 30 independent runs. As mentioned in the previous chapter, we use Cart pole and Acrobot environments for the initial experiments and for the final experiments, we additionally use Mountain Car environment.

6.4.1 HyperNEAT: Different genome types

In this experiment, we test six different genome types described in Section 4.1. We run the HyperNEAT algorithm with the default parameters as described in Table 2, i.e. 200 generations and population size of 20. The results are presented in Table 4.

genome type	succ	T_{avg}	$\frac{T_{avg}}{T_{max}}$%	T_{std}	T_{med}	R_{avg}	R_{std}	R_{med}	R_{max}	R_{min}
Cart pole										
CPPN(w)_1	10	174114	2.18	287980	34257	198	2	200	200	195
CPPN(w)_2	10	149652	**1.87**	353926	25278	**199**	1	200	200	195
CPPN(w+b)_1	10	413005	5.16	562120	221642	198	1	200	200	195
CPPN(w+b)_2	7	294081	3.68	189482	283067	156	74	197	200	9
CPPN(w,b)_1	8	490926	6.14	678514	75540	186	25	200	200	131
CPPN(w,b)_2	10	239543	2.99	273194	34598	**199**	1	200	200	195
Acrobot										
CPPN(w)_1	0	6892270	86.15	825378	7012095	-114	43	-99	-72	-200
CPPN(w)_2	0	6409798	**80.12**	398216	6412031	**-79**	4	**-77**	-74	-87
CPPN(w+b)_1	0	7673258	95.92	450079	7939093	-155	44	-159	-101	-200
CPPN(w+b)_2	0	7944667	99.31	182159	7999626	-178	40	-200	-90	-200
CPPN(w,b)_1	0	6851390	85.64	645649	6685811	-115	29	-106	-95	-200
CPPN(w,b)_2	0	6500385	81.25	1036491	6106833	-113	56	**-77**	-74	-200

Table 4: Comparing different genome types based on 10 independent runs on Cart pole and Acrobot environments

We can see that the Cart pole environment is not hard to solve as most of the genome types solved the environment in all 10 runs (the number of solved runs is in column *succ*). Hence, instead of reward, we compare T_{avg}, which is the average number of time steps that the algorithm used before solving the environment. We can see that the genomes without bias CPPN(w)_1 and CPPN(w)_2 converge the fastest as they used only 1-2 % of the provided budget of T_{max}. When comparing the genomes that generate bias, CPPN(w,b)_2 shows the best results as it used only 3 % of T_{max}.

Acrobot environment is more challenging to solve, and as it does not have a predefined reward threshold, we cannot say if the environment was solved. Therefore, we focus on the obtained return reward when comparing the algorithms. CPPN(w)_2 has the highest average return reward, and we can also see that the standard deviation of the rewards is low, suggesting that the learning algorithm is more stable than the other presented options. On the other hand, we can compare the median reward and see that CPPN(w,b)_2 has the same median performance as CPPN(w)_2, but did not manage to find a working solution in some of the runs as R_{min} is -200.

Although the genome type CPPN(w)_2 shows the best performance, we choose genome type CPPN(w,b)_2 for further experiments as it generates biases. This is an important feature of the genome type as later we combine HyperNEAT with DQN, and as DQN uses policy network with biases, we need to be able to generate them for compatibility reasons. While it would be possible not to use bias in the DQN policy networks, we choose not to do it as bias makes the network more general.

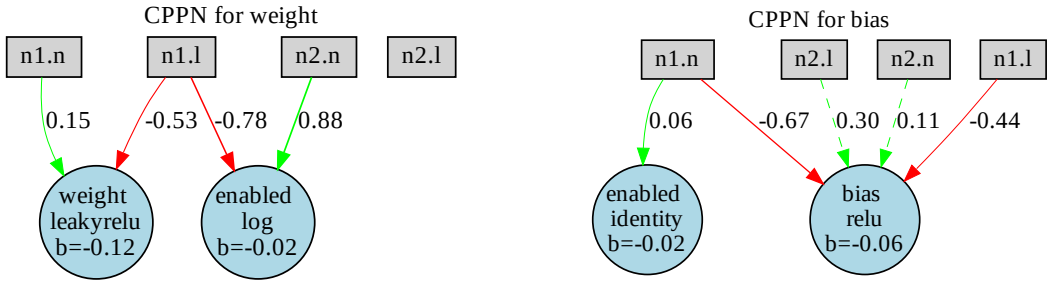


Figure 11: Example of an evolved genome with a simple topology that is converted to the corresponding CPPNs. The red connections have a negative weight, the green ones have a positive weight.

We can see an example of the best-performing genome of type CPPN(w,b)_2 on Acrobot in Figure 11. The grey nodes correspond to input keys where $n1$ is the first node, l is its layer index, and n is its node index. The blue nodes are output nodes, one for weight/bias and the other one for the boolean switch. The presented CPPNs are compact and do not have any hidden nodes. One of the more complex evolved genomes can be found in Appendix A. Each hidden node of the CPPN performs the specified activation function over the sum of its input signals and the node bias.

6.4.2 HyperNEAT→DQN: Initialisation of Q-networks (option 1)

In this experiment, we evaluate the effect of the DQN policy network and the target policy network initialisation by the policy network found by HyperNEAT. The initialisation is described in Section 4.2, Option 1. Firstly, we run HyperNEAT for $T_{max}/2$ time steps. Then, we copy the weights and biases of the found HyperNEAT policy network to the initial DQN (target) policy network. Finally, we train DQN for $T_{max}/2$ time steps and report the found DQN policy network performance.

Table 5 shows the performance of DQN for different initialisation of policy network Q and the target policy network $\hat{Q}$. If Q init is True, both the DQN policy network Q and the DQN target policy network $\hat{Q}$ are initialised by the HyperNEAT policy. Otherwise, only $\hat{Q}$ is initialised by the HyperNEAT policy network parameters and Q is initialised randomly as described in Section 4.2, Option 2. C is the target update frequency and determines the time step frequency for synchronising $\hat{Q}$ and Q.

For the Cart pole problem, we get the best results (both reward and used time steps) for $C = 10000$, which corresponds to the default settings (see Table 3) and no Q initialisation. For the Acrobot problem, we get the best average reward for the same settings. However, the best median reward is obtained for $C = 100000$ and no Q initialisation. In Appendix B, we

Q init	C	succ	T_{avg}	$\frac{T_{avg}}{T_{max}}$%	T_{std}	T_{med}	R_{avg}	R_{std}	R_{med}	R_{max}	R_{min}
Cart pole											
False	10000	10	205000	**5.12**	40469	215000	**198**	1	200	200	195
False	25000	10	1494400	37.36	824018	1704000	**198**	1	198	200	195
False	50000	3	3161000	79.03	1532577	4000000	92	86	52	200	10
False	100000	3	3218000	80.45	1507740	4000000	66	86	10	200	10
True	10000	7	1663000	41.58	1761162	1149000	178	56	199	200	10
True	25000	7	1213800	30.34	1824178	21000	143	85	198	200	10
True	50000	8	1118200	27.96	1681031	**16000**	161	75	200	200	10
True	100000	8	1032000	25.8	1599962	**16000**	161	75	200	200	10
baseline	10000	10	198200	4.96	14817	197000	198	1	199	200	195
Acrobot											
False	10000	0	4000000	100.0	0	4000000	**-90**	17	-87	-75	-137
False	25000	0	4000000	100.0	0	4000000	-92	18	-94	-72	-132
False	50000	0	4000000	100.0	0	4000000	-102	34	-88	-79	-200
False	100000	0	4000000	100.0	0	4000000	-105	38	**-84**	-69	-200
True	10000	0	4000000	100.0	0	4000000	-131	45	-105	-83	-200
True	25000	0	4000000	100.0	0	4000000	-125	48	-94	-79	-200
True	50000	0	4000000	100.0	0	4000000	-141	50	-128	-79	-200
True	100000	0	4000000	100.0	0	4000000	-142	45	-155	-79	-200
baseline	10000	0	4000000	100.0	0	4000000	-81	6	-81	-72	-93

Table 5: Testing Q (target) policy initialisation with different values of target update frequency C on 10 independent runs on Cart pole and Acrobot environments. Comparison with randomly initialised DQN.

can see the training performance of the first algorithms part (HyperNEAT) and the second algorithms part (DQN, $\hat{Q}$ init, C=10000).

Additionally, we compare the results with the baseline DQN that run for $T_{max}/2$ time steps, see rows *baseline* in Table 5. Especially in the case of Acrobot, we can see that it is better not to initialise the DQN (target) policy network when considering both the mean and the median performance.

6.4.3 HyperNEAT→DQN: Initialisation of the replay buffer (option 2)

In this experiment, we test the effect of buffer initialisation on the performance of DQN. The initialisation strategy is described in 4.2, Option 2. Firstly, we run HyperNEAT for $T_{max}/2$ time steps. Then, we take the final HyperNEAT policy network H and use it to initialise the replay buffer D up to the size T_I. Finally, we train DQN for $T_{max}/2 - T_I$ time steps and report the found DQN policy network performance. Additionally, we test the H_ϵ policy, that in 20% cases ($\epsilon = 0.2$) chooses random action and in the rest of the cases chooses the action based on H.

Table 6 shows the performance of DQN for varying initial replay buffer sizes T_I. Note that for larger T_I, we decrease the rollout size T_R. The rollout size T_R says how many samples we collect in each iteration. By decreasing it, we keep the total number of iterations $I = \frac{T_{max}-T_I}{T_R}$ similar among different settings of T_I. That means that the total number of effectively used samples $I \cdot B$ stays similar among the different settings.

We can see that performance on the Cart pole problem of baseline DQN and initialised DQN is comparable with respect to the mean number of performed iterations before the

T_I	T_R	ϵ	succ	T_{avg}	$\frac{T_{avg}}{T_{max}}\%$	T_{std}	T_{med}	R_{avg}	R_{std}	R_{med}	R_{max}	R_{min}
Cart pole												
50000	20	0.0	10	214400	5.36	38312	218000	**199**	1	**200**	200	196
100000	20	0.0	10	321400	8.04	83066	293000	197	2	196	200	195
500000	18	0.0	10	783400	19.58	53874	766000	198	1	198	200	195
1000000	15	0.0	10	1284000	32.1	59524	1288000	198	1	199	200	195
50000	20	0.2	10	209200	**5.23**	18765	**203000**	198	1	197	200	195
100000	20	0.2	10	275400	6.88	22163	272000	198	1	198	200	195
500000	18	0.2	10	735200	18.38	34850	737000	198	1	199	200	195
1000000	15	0.2	10	1199000	29.98	81688	1222000	**199**	0	**200**	200	197
baseline			10	198200	4.96	14817	197000	198	1	199	200	195
Acrobot												
50000	20	0.0	0	4000000	100.0	0	4000000	-92	13	-88	-76	-119
100000	20	0.0	0	4000000	100.0	0	4000000	-102	27	-96	-78	-176
500000	18	0.0	0	4000000	100.0	0	4000000	-124	35	-121	-82	-200
1000000	15	0.0	0	4000000	100.0	0	4000000	-131	41	-126	-76	-200
50000	20	0.2	0	4000000	100.0	0	4000000	-89	13	-85	-75	-125
100000	20	0.2	0	4000000	100.0	0	4000000	**-88**	12	**-84**	-79	-119
500000	18	0.2	0	4000000	100.0	0	4000000	-90	16	**-84**	-73	-127
1000000	15	0.2	0	4000000	100.0	0	4000000	-94	19	-87	-73	-136
baseline			0	4000000	100.0	0	4000000	-81	6	-81	-72	-93

Table 6: Results obtained for replay buffer initialisation on 10 independent runs on Cart pole and Acrobot environments. Comparison with randomly initialised DQN.

environment is found. Note that for the first T_I time steps, the DQN does not perform any training but uses these time steps to prefill the replay buffer D. For example, for $T_I = 1000000$, $\epsilon = 0.2$ we have $\frac{T_{avg}}{T_{max}}\%$ is 29.98%. However, when we subtract the initial number of time steps and calculate $\frac{T_{avg}-T_I}{T_{max}}\%$, we get 4.98%.

The performance on the Acrobot problem shows more interesting results. We can see that it is always better to use the ϵ-policy that acts randomly in 20% of cases. The median performance is slightly better for $T_I = 50000$, while the mean performance is slightly better for $T_I = 100000$. None of the initialised settings is substantially better than the baseline DQN with random initialisation.

6.4.4 HyperNEAT→DQN: Initialisation of the external policy (option 3)

In this experiment, we evaluate the effect of using an external HyperNEAT policy for the target predictions during the first T_H steps of the training. The initialisation is described in Section 4.2, Option 3. Firstly, we run HyperNEAT for $T_{max}/2$ time steps. Then, we pass the found HyperNEAT policy network to the DQN training loop. Finally, we train DQN for $T_{max}/2$ time steps, where for the first T_H time steps, we use the external HyperNEAT policy H to guide the target predictions. We report the trained DQN policy network performance. Table 7 shows the performance of DQN when the training is guided by H for varying initial time steps T_H.

Results on the Cart pole environment are comparable for small T_H. We can notice that the median number of time steps T_{med} is similar for different values of T_H and it happens that $T_{med} < T_H$, which means that environment was solved even though the selection of the target Q-value for the Bellman update was guided strictly by the policy found by HyperNEAT.

T_H	succ	T_{avg}	$\frac{T_{avg}}{T_{max}}\%$	T_{std}	T_{med}	R_{avg}	R_{std}	R_{med}	R_{max}	R_{min}
Cart pole										
100000	10	218400	**5.46**	33260	**213000**	198	2	**200**	200	195
250000	10	268200	6.7	125950	214000	198	2	199	200	195
500000	10	423600	10.59	351233	214000	198	1	**200**	200	195
1000000	10	687600	17.19	903360	214000	198	1	199	200	195
2000000	10	498200	12.46	459373	214000	**199**	1	**200**	200	195
baseline	10	198200	4.96	14817	197000	198	1	199	200	195
Acrobot										
100000	0	4000000	100.0	0	4000000	-82	6	-80	-73	-95
250000	0	4000000	100.0	0	4000000	-83	10	-79	-73	-109
500000	0	4000000	100.0	0	4000000	**-79**	5	-79	-72	-92
1000000	0	4000000	100.0	0	4000000	-86	8	-88	-73	-101
2000000	0	4000000	100.0	0	4000000	-82	8	**-77**	-73	-102
baseline	0	4000000	100.0	0	4000000	-81	6	-81	-72	-93

Table 7: Results obtained for external policy initialisation on 10 independent runs on Cart pole and Acrobot environments. Comparison with randomly initialised DQN.

Though, we have to note that the initialising policy H was optimal (in the sense that it was able to solve the environment) in all of the 10 independent runs.

The Acrobot environment results show the best average performance for $T_H = 500000$ and the best median performance for $T_H = 2000000$. Moreover, the initialised DQN performs slightly better than the baseline DQN with random initialisation in both the mean and the median reward. We can see the training performance for both $T_H = 500000$ and $T_H = 2000000$ in Appendix C.

6.4.5 HyperNEAT→DQN: Initialisation of the replay buffer and the external policy

In this experiment, we combine the two previous experiments and test the combined initialisation of the replay buffer and external policy. We test a subset of combinations of different T_H and different T_I used in the previous experiments. We test only the cases where $T_I < T_H$ because the external policy H is effectively used for the first $T_H - T_I$ time steps from the point when the training starts. We also do not test the non-randomised buffer initialisation as the experiment described in Section 6.4.3 showed better performance for $\epsilon = 0.2$ in all of the tested cases. The results are presented in Table 8.

We can see that some of the Cart pole runs did not converge. Additionally, the training is not very stable as for fixed T_I, the average performance varies a lot. However, the median performance stays similar to what we have seen in the previous experiments.

The Acrobot results are the best for $T_I = 100000$ and $T_H = 500000$, where we get an average reward of -85 and a median reward of -74. In comparison, we take the results from buffer initialisation (Table 6), where the average reward for $T_I = 100000$ was -88, and the median reward was -84. For the external policy initialisation (Table 7) with $T_H = 500000$ the average reward was -79 and the median reward was -79.

T_I	T_H	succ	T_{avg}	$\frac{T_{avg}}{T_{max}}\%$	T_{std}	T_{med}	R_{avg}	R_{std}	R_{med}	R_{max}	R_{min}
Cart pole											
50000	100000	10	217400	**5.44**	24739	**213000**	197	1	197	200	195
50000	250000	9	622400	15.56	1127072	241000	198	2	199	200	193
50000	500000	10	627000	15.68	832398	220000	198	1	199	200	195
50000	1000000	10	371800	9.3	319408	219000	198	1	199	200	195
50000	2000000	10	637600	15.94	710155	228000	**199**	1	**200**	200	195
100000	250000	10	309400	7.74	96718	276000	197	1	198	200	195
100000	500000	9	731600	18.29	1116940	270000	198	1	199	200	194
100000	1000000	10	493000	12.32	452196	275000	198	1	199	200	195
100000	2000000	10	527200	13.18	501481	278000	**199**	1	**200**	200	196
500000	1000000	10	753200	18.83	47127	761000	198	1	198	200	195
500000	2000000	10	739200	18.48	49389	750000	197	1	197	200	195
baseline		10	198200	4.96	14817	197000	198	1	199	200	195
Acrobot											
50000	100000	0	4000000	100.0	0	4000000	-90	14	-88	-74	-123
50000	250000	0	4000000	100.0	0	4000000	-86	8	-85	-76	-107
50000	500000	0	4000000	100.0	0	4000000	-88	13	-82	-74	-120
50000	1000000	0	4000000	100.0	0	4000000	-86	10	-82	-76	-109
50000	2000000	0	4000000	100.0	0	4000000	-97	14	-95	-82	-119
100000	250000	0	4000000	100.0	0	4000000	-89	11	-86	-74	-110
100000	500000	0	4000000	100.0	0	4000000	**-85**	11	**-81**	-74	-112
100000	1000000	0	4000000	100.0	0	4000000	-93	19	-85	-73	-148
100000	2000000	0	4000000	100.0	0	4000000	-95	15	-92	-75	-124
500000	1000000	0	4000000	100.0	0	4000000	-96	15	-90	-75	-124
500000	2000000	0	4000000	100.0	0	4000000	-89	14	-87	-73	-126
baseline		0	4000000	100.0	0	4000000	-81	6	-81	-72	-93

Table 8: Different hyperparameters for simultaneous replay buffer and external policy initialisation on 10 independent runs on Cart pole and Acrobot environments. Comparison with randomly initialised DQN.

6.4.6 HyperNEAT→DQN loop: Fine-tuning of CPPNs by a DQN policy network

In this experiment, we tested the whole loop as shown in Figure 8 and the SGD fine-tuning of CPPNs with recombination into the previous HyperNEAT population to initialise HyperNEAT. We used the DQN initialisation option 3 to initialise DQN.

All of the experiments have 16000000 allowed time steps, i.e. two times more than in the previous experiments. We tested a different number of iterations in the loop with different ratios of the time steps in each of the iterations. An example is that for the current $T_{max} = 16000000$, we take 20 iterations of both of the algorithms, which is 40 iterations in total. Each of them is allowed $T^i_{max} = 16000000/40 = 400000$ time steps. We first run HyperNEAT with the limit of $T^1_{max} = 400000$ time steps, then we use its policy network to initialise DQN, then we run DQN with the limit of $T^2_{max} = 400000$ time steps, then we use DQN policy network to initialise HyperNEAT, then we again run HyperNEAT and repeat this loop until both HyperNEAT and DQN run 20 times.

The reported results were created in the following way: For each of the independent runs, we took all logs for HyperNEAT/DQN, sorted them based on iteration id and merged those logs together into one. We did this separately for DQN and HyperNEAT. One selected run of HyperNEAT→DQN loop can be seen in Appendix D, Figure 20 and its corresponding merged version can be seen in Appendix D, Figure 19.

Since the local number of allowed time steps T^i_{max} varies among different settings, we set the external policy threshold as $T_H = T^i_{max}/2$. Additionally, we test various backpropagation qualities $q \in \{0.01, 0.5, 0.75, 0.9\}$. The backpropagation quality q says how much can the fitness of the HyperNEAT decrease in the fine-tuning step, see Section 4.3.

The full results can be found in Appendix D, Table 12. We found out that the parameter q does not significantly affect the results. Hence, we present results with fixed $q = 0.01$ in Table 9. We run the experiment on the Acrobot environment only as we could see in the previous experiments that the Cart pole environment is too easy to run several iterations on it.

T^i_{max}	T_H	succ	T_{avg}	$\frac{T_{avg}}{T_{max}}\%$	T_{std}	T_{med}	R_{avg}	R_{std}	R_{med}	R_{max}	R_{min}
DQN											
400000	200000	0	8000000	100.0	0	8000000	**-78**	4	**-77**	-73	-86
800000	400000	0	8000000	100.0	0	8000000	-84	7	-85	-71	-99
1600000	800000	0	8000000	100.0	0	8000000	-81	3	-81	-75	-89
4000000	2000000	0	8000000	100.0	0	8000000	-82	8	-79	-73	-101
HyperNEAT											
400000	200000	0	6474555	80.93	696695	6395349	-78	10	**-70**	-69	-100
800000	400000	0	6444218	80.55	697855	6451861	-77	9	**-70**	-69	-95
1600000	800000	0	6215327	**77.69**	518156	**6057495**	**-72**	6	**-70**	-68	-85
4000000	2000000	0	6556365	81.95	861981	6433503	-91	38	-73	-69	-200

Table 9: HyperNEAT→DQN loop with different number of iterations and T^i_{max} ratios with fixed $T_{max} = 16000000$, $q = 0.01$, and Acrobot environment.

To provide results comparable with the previous experiments, we also show results for $T_{max} = 8000000$ in Table 10 (resp. Appendix D, Table 13). These results were obtained from the same experiment by considering only the first half of the logged results.

T^i_{max}	T_H	succ	T_{avg}	$\frac{T_{avg}}{T_{max}}\%$	T_{std}	T_{med}	R_{avg}	R_{std}	R_{med}	R_{max}	R_{min}
DQN											
400000	200000	0	4000000	100.0	0	4000000	-89	11	-86	-76	-112
800000	400000	0	4000000	100.0	0	4000000	-90	12	-85	-78	-118
1600000	800000	0	4000000	100.0	0	4000000	**-87**	7	**-84**	-77	-100
4000000	2000000	0	4000000	100.0	0	4000000	-97	32	-86	-73	-173
HyperNEAT											
400000	200000	0	3388710	84.72	346031	3459711	**-78**	10	**-70**	-69	-100
800000	400000	0	3432460	85.81	441206	3363858	-104	48	-84	-69	-200
1600000	800000	0	3386959	**84.67**	363561	**3317717**	-99	50	-71	-68	-200
4000000	2000000	0	3446845	86.17	420841	3387507	-108	47	-88	-70	-200

Table 10: HyperNEAT→DQN loop with different number of iterations and T^i_{max} ratios with fixed $T_{max} = 8000000$, $q = 0.01$, and Acrobot environment.

We can see that the DQN part of the algorithm produces more stable results (standard deviation is lower), while the HyperNEAT part of the algorithm produces a higher median reward. Since we will later compare the algorithms with $T_{max} = 8000000$, we focus on the results presented in Table 10. We can see that the HyperNEAT part of the algorithm performs the best for more iterations with a lower number of allowed time steps $T^i_{max} = 400000$ where both the average and median reward are the lowest. DQN part of the algorithm performs similarly for different values of T^i_{max}.

6.4.7 Comparison of all approaches

To summarise the previous experiments, we select three hybrid neuroevolution approaches that performed the best. We compare the selected approaches with baseline DQN and HyperNEAT. We run 30 independent runs for each of the algorithms with fixed hyperparameters. Each of the algorithms as allowed $T_{max} = 8000000$ time steps. The tested methods are summarised below:

1. Baseline HyperNEAT
   HyperNEAT with genome type CPPN(w,b)_2 as tested in 6.4.1.
2. Baseline DQN
   DQN with hyperparameters described in Table 3.
3. HyperNEAT→DQN (Option 3)
   DQN initialised by external policy from HyperNEAT with $T_H = 500000$ as tested in Section 6.4.4.
4. HyperNEAT→DQN (Options 2, 3)
   DQN initialised by external policy and preloaded replay buffer from HyperNEAT with $T_H = 500000$ and $T_I = 100000$ as tested in Section 6.4.5.
5. HyperNEAT→DQN loop
   The full HyperNEAT→DQN loop as proposed in Figure 8 and tested in Section 6.4.6. We use the version with 10 iterations, $T_H = 100000$ and $T^i_{max} = 400000$.

The results are summarised in Table 11. Each of the proposed approaches is composed of two (or more for the HyperNEAT→DQN loop) algorithm parts. Contrary to the previous experiments, in this section, we do not report the results of DQN and HyperNEAT parts separately but join them together.

Note that both DQN and HyperNEAT terminate when the environment is solved (reward threshold 195 for Cart pole, -110 for Mountain car, -60 for Acrobot). However, we still run the full round of the algorithm. For example, when we run HyperNEAT→DQN loop, and HyperNEAT solves the environment in iteration i and time step $T_c < T^i_{max}$, we stop HyperNEAT, initialise DQN and run it for another at most T^i_{max} time steps. Then the loop does not continue with the next iteration $i + 1$ because the solution was found. Additionally, HyperNEAT effectively never uses the full budget of T_{max}. Hence, the algorithms are not fairly comparable based on the number of timesteps, and we should rather compare them based on reward progress, the final solution's quality and the stability of the training over a set of independent runs.

We performed Mann–Whitney U test to compare the median rewards R_{med} of the proposed approaches with the median reward of the baseline DQN. The null hypothesis H_0 is that the distributions of rewards R_j for the proposed approach and the baseline DQN do not differ significantly with the two-sided confidence interval $\alpha = 0.05$. The corresponding median rewards can be found in Table 11.

algorithm	succ	T_{avg}	$\frac{T_{avg}}{T_{max}}\%$	T_{std}	T_{med}	R_{avg}	R_{std}	R_{med}	R_{max}	R_{min}
Cart pole										
HN	30	182773	2.28	296084	**19899**	**199**	0	200	200	195
DQN	30	181133	**2.26**	26273	182000	198	1	200	200	195
HN→DQN (3)	30	844610	10.56	625691	775976	**199**	0	200	200	198
HN→DQN (2, 3)	30	870144	10.88	863978	663719	**199**	0	200	200	197
Acrobot										
HN	0	6656073	83.2	899777	6376284	-110	50	-79	-72	-200
DQN	0	8000000	100.0	0	8000000	-80	6	-79	-72	-97
HN→DQN (3)	0	7430118	92.88	460858	7352425	-81	10	-77	-70	-110
HN→DQN (2, 3)	0	7430118	92.88	460858	7352425	-81	9	-78	-70	-119
HN→DQN loop	0	7407774	92.6	362820	7372038	**-74**	6	**-71**	-69	-97
Mountain car										
HN	22	1771840	**22.15**	1518094	**1150113**	-131	41	-108	-97	-200
DQN	0	8000000	100.0	0	8000000	-183	24	-198	-121	-200
HN→DQN (3)	21	5784260	72.3	1583286	5075801	-131	39	-109	-100	-200
HN→DQN (2, 3)	21	5733727	71.67	1649619	5075801	-129	36	-109	-100	-200
HN→DQN loop	**25**	5977865	74.72	1560565	6088169	**-107**	16	**-103**	-94	-168

Table 11: Comparison of baseline HyperNEAT and DQN with 3 selected proposed approaches. Results for $T_{max} = 8000000$, HN stands for HyperNEAT.

For Acrobot and the pairs of DQN and HN→DQN (3), DQN and HN→DQN (2, 3), DQN and HN→DQN loop, the null hypothesis is rejected at the default 5% significance level. For Mountain car and the pairs of DQN and HN→DQN (3), DQN and HN→DQN (2, 3), DQN and HN→DQN loop, the null hypothesis is rejected at the default 5% significance level.

We show that the median return reward of all of the proposed hybrid approaches on both Acrobot and Mountain car is better than for the baseline DQN and that the differences in the median return rewards are statistically significant.

We show the reward progress on all three tested environments in Figure 12. In the left column, we can see the reward progress for the median run selected from the 30 independent runs of each approach. The median run is considered w.r.t. the number of used time steps for Cart pole environment and w.r.t the final reward for Acrobot and Mountain car environments. In the right column, we analyse the runtime behaviour of the approaches. The graph is a horizontal cross-section of runtime distribution function (rtd) [23] for 0.75-quantile where $rtd : \mathbb{R} \times \mathbb{R} \to [0, 1]$ is defined as:

$$rtd(t, q) = P(t, q) = P[T \leq t, Q \leq q] \approx \frac{n(t, q)}{N} \tag{12}$$

which is a probability P that a solution of quality $Q \leq q$ is found in time $T \leq t$. We approximate this probability by counting $n(t, q)$ which is the number of time that the tested approach had quality $Q \leq q$ in timestep $T \leq t$ and calculating the average over the N independent runs. The resulting plot in Figure 12 is a cross-section of $rtd(t, q) = 0.75$, which we interpret as 75% of the runs will have a better performance (faster convergence, higher fitness) than is shown in the graph.

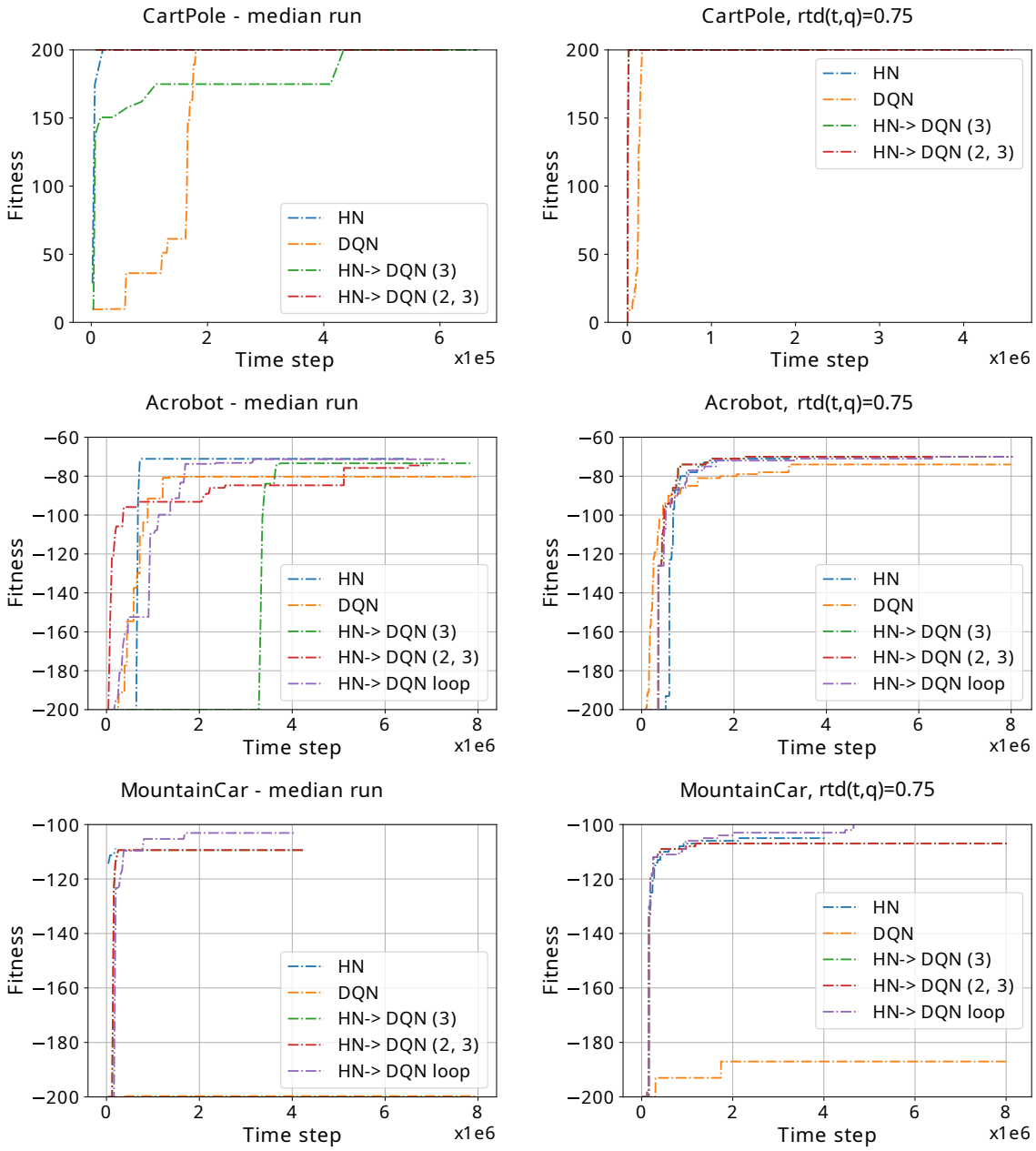


Figure 12: Performance of 3 selected approaches and 2 baselines on all environments. The left column shows the median run (for CartPole w.r.t. used time steps, for MountainCar and Acrobot w.r.t. final reward). The right column shows that with probability 75%, in time step t, the reward is R (calculated over 30 independent runs). HN is a shortcut for HyperNEAT.

6.5 Discussion

In this chapter, we interpret the experimental results and discuss both the proposed hybrid methods that worked and those that did not work. We also outline the open questions that were raised in the thesis.

In the experiment described in Section 6.4.1, we saw that for the selected problems, the policies represented by neural networks without biases perform the best. This suggests that the tested environments might be too simple. Thus, it would be interesting to test the approaches on more complex environments, where the problem of exploration vs. exploitation would have a higher significance. However, we did not do so for computational reasons. Moreover, we showed that it is better to design the genome with two different CPPNs, one generating weights and the other one generating bias, rather than having one CPPN generating both weights and bias. We saw that some of the resulting CPPNs are very compact, with only a few hidden nodes.

The experiment described in Section 6.4.2 points out the main problem in combining DQN and HyperNEAT that arises from the fact that the outputs of the two algorithms have a different interpretation. DQN produces Q-values that should follow the Bellman equation (1), while HyperNEAT does not have such interpretation. It just produces values for a given state such that the highest one corresponds to the winning action to be applied in the state. By setting the policy network found by HyperNEAT as the initial DQN policy network, we disrupted the coherence of the Q-values and made it harder for DQN to converge to a reasonably good policy. In Figure 13, we ran one episode of a well-performing policy from DQN in the Cart pole environment and reported the maximising Q-value for each of the observations that the agent encountered. Then, we took a well-performing HyperNEAT policy and reported the maximising output value for each of the observations collected by the previously run DQN. Since the scale of the outputs of DQN and HyperNEAT differs, we normalised the output values methodwise. We can observe the relative differences in consecutive output values for both of the algorithms. We can see that while the Q-values of DQN grow with the number of time steps as the pole on the cart stabilises, the HyperNEAT output values do not follow this trend.

The experiment described in Section 6.4.3 shows that while the buffer initialisation did not show better performance than the baseline, it was not much worse in the median reward. Moreover, when observing the runtime performance of HyperNEAT, we could notice that even when HyperNEAT does not learn a reasonably good policy, it does not spoil the sequent run of DQN much. The experiment clearly showed that it is crucial to use the ϵ-policy for filling the replay buffer.

Setting the external policy (see the experiment in Section 6.4.4) showed the best performance out of the three proposed initialisation methods. It is the most subtle initialisation method as we are not modifying the initial parameters of the DQN policy network, nor we are feeding it samples that were not generated by the DQN policy. The external policy helps to make more educated predictions during the loss calculation where it selects the 'best' action, which is something that the initial random DQN policy cannot do properly. The ex-

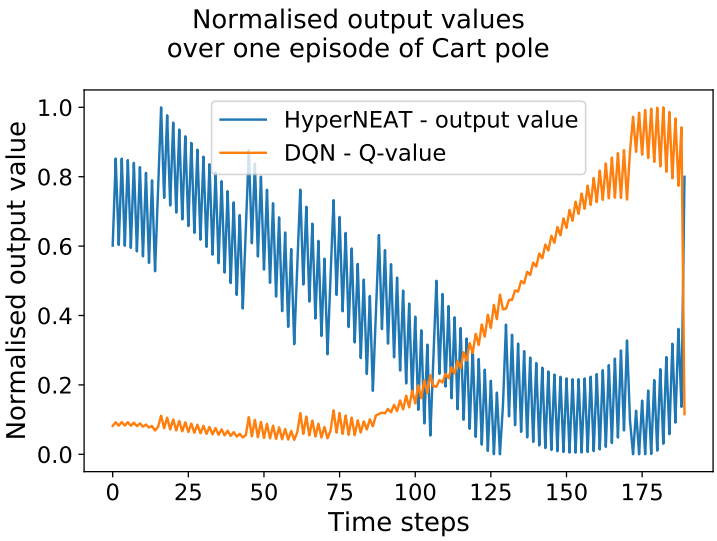


Figure 13: Normalised Q-values of DQN vs. output values of HyperNEAT during one episode of Cart pole

periment described in Section 6.4.5 shows that the combination of external policy and buffer initialisation provides the best median return rewards.

The full loop with both DQN and HyperNEAT showed promising results with a good resulting median reward, see Section 6.4.6. Moreover, the experiments that performed more iterations of shorter total time step length showed better performance than those that performed fewer iterations of longer length. This suggests that iterating the HyperNEAT and DQN helps to obtain a good solution faster. The general observation is that it is much easier to initialise HyperNEAT by DQN than vice-versa, as, in this initialisation direction, we loosen the requirements on the algorithm output interpretation as Q-values. It would be nice to test other alternatives of the loop algorithm. We could, for example, copy the DQN policy between each iteration of the loop algorithm, which would ensure continuity of the DQN policy between the sequent iterations helping to train it faster. On the other hand, it could cause stagnation of the DQN policy if the transferred solution was stuck in a local optimum. We could also change the initialisation strategy of HyperNEAT by initialising the population completely from the fine-tuned genome by mutating it into several different individuals and adding some random individuals to the initial population.

Finally, the runtime analysis of the two baselines and three selected proposed approaches shows that the HyperNEAT-based approaches perform well on a previously unseen problem. Though, here we need to stress out that the DQN hyperparameters were not optimised. It would be interesting to test the approaches on more unseen problems to get a better idea of whether the good performance on the unseen Mountain car problem was coincidental or not. We do not compare the algorithms based on the runtime because it is implementation-dependent. Though, the general observation is that HyperNEAT is faster as it is easily parallelisable.

7 Conclusion

This thesis proposes hybrid neuroevolution approaches for the reinforcement learning domain that builds upon the DQN and HyperNEAT algorithms. We designed DQN initialisation strategies using a policy evolved by HyperNEAT as well as a strategy for using a trained DQN policy to enhance the HyperNEAT population. We combined the aforementioned into a training loop that iteratively runs HyperNEAT and DQN. We experimentally evaluated the proposed approaches. The best results have been obtained for the hybrid approach that runs multiple iterations of the two steps – HyperNEAT and DQN.

We have shown that combining the HyperNEAT and DQN algorithms is challenging as they have different interpretability of their outputs. Additionally, reporting the performance of the algorithms is not straightforward either as both of the algorithms have a different workflow and random initialisations that change the performance each time the algorithms run.

While we have shown that the proposed initialisation strategies help improve the training by either converging faster or reaching a better median return reward, we would need to provide more extensive testing to draw any conclusions. The experimental results suggest that the proposed methods perform better than DQN, the more complicated the environment is to solve. We tested the approaches only on three classical reinforcement learning environments because the experiments would be extremely computationaly demanding otherwise. Hence, we used rather simple environments as proof-of-concept problems to demonstrate various possibilities of hybrid neuroevolution. To perform a fair comparison, we would also have to optimise both baseline algorithms' hyperparameters. It could also be beneficial to normalise the rewards, which could improve the stability of the training.

The diploma thesis raises many questions that had not been anticipated in advance. We selected only one specific representative of the evolutionary algorithm and one specific representative of a gradient-based algorithm, but the general idea to leverage an evolutionary-based method for exploration and gradient-based method for local-search could be employed for other algorithms as well. For example, instead of NEAT, we could use genetic programming to create a population of individuals representing functions generating the policy network parameters. The hybridisation of neuroevolution with other deep RL methods such as policy gradient methods could be another interesting direction of research. Note that the hybrid neuroevolution is not limited to applications in deep RL. Thus, its use for other problem domains such as regression can be investigated as well.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, 2017.

[2] Ronan Collobert, Jason Weston, Leon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural Language Processing (Almost) from Scratch. *Natural language processing*, pages 2493–2537, August 2011.

[3] J. Bobadilla, F. Ortega, A. Hernando, and A. Gutiérrez. Recommender systems survey. *Knowledge-Based Systems*, 46:109–132, July 2013.

[4] Daniel Kobler. Evolutionary Algorithms in Combinatorial Optimization. In Christodoulos A. Floudas and Panos M. Pardalos, editors, *Encyclopedia of Optimization*, pages 950–959. Springer US, Boston, MA, 2009.

[5] P.J Fleming and R.C Purshouse. Evolutionary algorithms in control systems engineering: a survey. *Control Engineering Practice*, pages 1223–1241, 2002.

[6] Kenneth O. Stanley and Risto Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, pages 99–127, 2002.

[7] Kenneth O Stanley, David D'Ambrosio, and Jason Gauci. A Hypercube-Based Indirect Encoding for Evolving Large-Scale Neural Networks. *Art. Life*, pages 185–212, 2009.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv*, 2013.

[9] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, pages 436–444, 2015.

[10] Boris Flach, Vojtěch Franc, and Jan Drchal. *Lecture notes in Statistical Machine Learning*. Czech Technical University, April 2021.

[11] Robert Legenstein. *Lecture notes in Deep Learning*. TU Graz, April 2021.

[12] Frederico Azevedo, Ludmila Carvalho, Lea Grinberg, Jose Farfel, Renata Ferretti-Rebustini, Renata Leite, Wilson Filho, Roberto Lent, and Suzana Herculano-Houzel. Equal Numbers of Neuronal and Nonneuronal Cells Make the Human Brain an Isometrically Scaled-Up Primate Brain. *The Journal of comparative neurology*, April 2009.

[13] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, pages 115–133, 1943. Springer.

[14] Herbert Robbins and Sutton Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, pages 400–407, 1951.

[15] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on ML*, volume 28 of *Proceedings of ML Research*, pages 1139–1147, Atlanta, Georgia, USA, 2013. PMLR.

[16] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*, 2014.

[17] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, pages 85–117, 2015.

[18] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, pages 1929–1958, 2014.

[19] S. Hochreiter. The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. *Int. J. Uncert. Fuzziness Knowl. B. Syst.*, pages 107–116, 1998.

[20] Vinod Nair and Geoffrey E. Hinton. Rectified Linear Units Improve Restricted Boltzmann Machines. In *Proceedings of the 27th International Conference on International Conference on ML*, ICML'10, pages 807–814, Madison, WI, USA, 2010. Omnipress.

[21] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv*, 2015.

[22] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training Recurrent Neural Networks. *arXiv*, 2013.

[23] Petr Pošík and Jiří Kubalík. *Lecture notes in Evolutionary Optimization Algorithms.* Czech Technical University, April 2021.

[24] Sean Luke. *Essentials of Metaheuristics.* 2009.

[25] B. Sareni and L. Krahenbuhl. Fitness sharing and niching methods revisited. *IEEE Trans. Evol. Computat.*, pages 97–106, 1998.

[26] Riccardo Poli, William Langdon, and Nicholas Mcphee. *A Field Guide to Genetic Programming.* 2008. (With contributions by J. R. Koza).

[27] Jiri Kubalik, Eduard Alibekov, Jan Žegklitz, and Robert Babuska. Hybrid Single Node Genetic Programming for Symbolic Regression. In *Transactions on Computational Collective Intelligence XXIV*, pages 61–82. Springer, Berlin, Heidelberg, 2016.

[28] David B D'Ambrosio, Jason Gauci, and Kenneth O Stanley. HyperNEAT: The First Five Years. *Springer*, Studies in Computational Intelligence, vol 557., 2014.

[29] Kenneth O. Stanley, Jeff Clune, Joel Lehman, and Risto Miikkulainen. Designing neural networks through neuroevolution. *Nat Mach Intell*, pages 24–35, 2019.

[30] Risto Miikkulainen. Neuroevolution. In Claude Sammut and Geoffrey I. Webb, editors, *Encyclopedia of Machine Learning*, pages 716–720. Springer US, Boston, MA, 2020.

[31] Javier Del Ser, Eneko Osaba, Daniel Molina, Xin-She Yang, Sancho Salcedo-Sanz, David Camacho, Swagatam Das, Ponnuthurai N. Suganthan, Carlos A. Coello Coello, and Francisco Herrera. Bio-inspired computation: Where we stand and what's next. *Swarm and Evolutionary Computation*, 48:220–250, 2019.

[32] Felix A Sosa and Kenneth O Stanley. Deep HyperNEAT: Evolving the Size and Depth of the Substrate. Evolutionary Complexity Research Group Undergraduate Research Report, University of Central Florida, Department of Computer Science, 2018.

[33] Felipe Petroski Such, Vashisht Madhavan, Edoardo Conti, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. *arXiv*, 2018.

[34] Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population Based Training of Neural Networks. *arXiv*, 2017.

[35] Adrien Lucas Ecoffet. Paper Repro: Deep Neuroevolution, April 2018. URL: `towardsdatascience.com/paper-repro-deep-neuroevolution-756871e00a66`.

[36] Yujin Tang, Duong Nguyen, and David Ha. Neuroevolution of Self-Interpretable Agents. *Proceedings of the 2020 GECCO*, pages 414–424, 2020.

[37] Yiming Peng, Gang Chen, Harman Singh, and Mengjie Zhang. NEAT for large-scale reinforcement learning through evolutionary feature learning and policy gradient search. In *Proceedings of the GECCO*, pages 490–497, Kyoto Japan, 2018. ACM.

[38] Ang Li, Ola Spyra, Sagi Perel, Valentin Dalibard, Max Jaderberg, Chenjie Gu, David Budden, Tim Harley, and Pramod Gupta. A Generalized Framework for Population Based Training. *arXiv*, 2019.

[39] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD*, pages 1487–1495, Halifax NS Canada, 2017. ACM.

[40] Thomas G. van den Berg and Shimon Whiteson. Critical factors in the performance of HyperNEAT. In *Proceeding of GECCO '13*, Amsterdam, Netherlands, 2013. ACM.

[41] Kenneth O Stanley, Jeff Clune, David B D'Ambrosio, Colin D Green, Joel Lehman, Gregory Morse, Justin K Pugh, Sebastian Risi, and Paul Szerlip. CPPNs Effectively Encode Fracture: A Response to Critical Factors in the Performance of HyperNEAT. Technical Report CS-TR-13-05, University of Central Florida Dept. of EECS, 2013.

[42] Sebastian Risi and Kenneth O. Stanley. Indirectly Encoding Neural Plasticity as a Pattern of Local Rules. In Stéphane Doncieux, Benoît Girard, Agnès Guillot, John Hallam, Jean-Arcady Meyer, and Jean-Baptiste Mouret, editors, *From Animals to Animats 11*, Lecture Notes in Computer Science, pages 533–543, Berlin, Heidelberg, 2010. Springer.

[43] J. Clune, K. O. Stanley, R. T. Pennock, and C. Ofria. On the Performance of Indirect Encoding Across the Continuum of Regularity. *IEEE Transactions on Evolutionary Computation*, pages 346–367, 2011.

[44] Phillip Verbancsics and Kenneth O. Stanley. Constraining connectivity to encourage modularity in HyperNEAT. In *Proceedings of GECCO '11*, Dublin, Ireland, 2011. ACM.

[45] Sebastian Risi and Kenneth O. Stanley. An Enhanced Hypercube-Based Encoding for Evolving the Placement, Density, and Connectivity of Neurons. *Art. Life*, pages 331–363, 2012.

[46] Zdeněk Buk. NEAT in HyperNEAT Substituted with Genetic Programming. In *Adaptive and Natural Computing Algorithms*, Berlin, Heidelberg, 2009. Springer.

[47] Risto Miikkulainen, Jason Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving Deep Neural Networks. *arXiv*, 2017.

[48] Jonas da Silveira Bohrer, Bruno Iochins Grisci, and Marcio Dorn. Neuroevolution of Neural Network Architectures Using CoDeepNEAT and Keras. *arXiv*, 2020.

[49] Alan McIntyre, Matt Kallada, Cesar G. Miguel, and Carolina Feher da Silva. neat-python, 2019. URL: `github.com/CodeReclaimers/neat-python`.

[50] Uber Research and Joel Lehman. PyTorch-NEAT, 2018. URL: `github.com/uber-research/PyTorch-NEAT`.

[51] Cristian Bodnar. TensorFlow-NEAT, 2018. URL: `github.com/crisbodnar/TensorFlow-NEAT`.

[52] Uber Research. Deep-Neuroevolution, 2019. URL: `github.com/uber-research/deep-neuroevolution`.

[53] Jeff Clune and Hod Lipson. Evolving 3D objects with a generative encoding inspired by developmental biology. *SIGEVOlution*, pages 2–12, 2011.

[54] Francesco Calimeri, Aldo Marzullo, Claudio Stamile, and Giorgio Terracina. Blood Vessel Segmentation in Retinal Fundus Images Using Hypercube NeuroEvolution of Augmenting Topologies (HyperNEAT). pages 173–183. Smart Innovation, Systems and Technologies, 2019.

[55] Phillip Verbancsics and Josh Harguess. Feature Learning HyperNEAT: Evolving Neural Networks to Extract Features for Classification of Maritime Satellite Imagery. In *Information Processing in Cells and Tissues*, pages 208–220. Springer, Cham, 2015.

[56] David Ha. Neurogram, July 2015. URL: `blog.otoro.net/2015/07/31/neurogram/`.

[57] Jimmy Secretan, Nicholas Beato, David B. D'Ambrosio, Adelein Rodriguez, Adam Campbell, Jeremiah T. Folsom-Kovarik, and Kenneth O. Stanley. Picbreeder: A Case Study in Collaborative Evolutionary Exploration of Design Space. *Evolutionary Computation*, pages 373–403, 2011.

[58] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv*, 2016.

[59] Matthew Hausknecht, Piyush Khandelwal, Risto Miikkulainen, and Peter Stone. HyperNEAT-GGP: a HyperNEAT-based atari general game player. In *Proceedings of GECCO '12*, Philadelphia, Pennsylvania, USA, 2012. ACM.

[60] Felipe Petroski Such, Vashisht Madhavan, Rosanne Liu, Rui Wang, Pablo Samuel Castro, Yulun Li, Jiale Zhi, Ludwig Schubert, Marc G. Bellemare, Jeff Clune, and Joel Lehman. An Atari Model Zoo for Analyzing, Visualizing, and Comparing Deep Reinforcement Learning Agents. *Proceedings of IJCAI 2019*, 2019.

[61] Jacob Schrum. Evolving indirectly encoded convolutional neural networks to play tetris with low-level features. In *Proceedings of GECCO '18*, pages 205–212, Kyoto Japan, 2018. ACM.

[62] Nick Cheney, Robert MacCurdy, Jeff Clune, and Hod Lipson. Unshackling evolution: evolving soft robots with multiple materials and a powerful generative encoding. In *Proceeding of GECCO '13*, page 167, Amsterdam, The Netherlands, 2013. ACM Press.

[63] Suchan Lee, Jason Yosinski, Kyrre Glette, Hod Lipson, and Jeff Clune. Evolving Gaits for Physical Robots with the HyperNEAT Generative Encoding: The Benefits of Simulation. In Anna I. Esparcia-Alcázar, editor, *Applications of Evolutionary Computation*, Lecture Notes in Computer Science, pages 540–549, Berlin, Heidelberg, 2013. Springer.

[64] Jan Drchal, Jan Koutnik, and Miroslav Snorek. HyperNEAT controlled robots learn how to drive on roads in simulated environment. In *2009 IEEE Congress on Evolutionary Computation*, pages 1087–1092, Trondheim, Norway, 2009. IEEE.

[65] Jan Dvorský. *Neuroevolutionary design of control strategy of a multi-legged robot.* Bachelor thesis, Czech Technical University in Prague, 2013.

[66] Evert Haasdijk, Andrei A Rusu, and A E Eiben. HyperNEAT for Locomotion Control in Modular Robots. *Evolvable Systems: From Biology to Hardware*, pages 169–180, 2010.

[67] E. Bahçeci and R. Miikkulainen. Transfer of evolved pattern-based heuristics in games. *2008 IEEE Symposium On Computational Intelligence and Games*, 2008.

[68] Sabre Didi. Multi-Agent Behavior-Based Policy Transfer. 2016.

[69] Brian D. Boyles. Evolving Scout Agents for Military Simulations. Master's thesis, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, 2015.

[70] Christian Kroos and Mark Plumbley. Neuroevolution for Sound Event Detection in Real Life Audio: A Pilot Study. In *Proceedings of the Detection and Classification of Acoustic Scenes*, 2017.

[71] Michal Pěchouček, Branislav Bošanský, and Jiří Kléma. *Lecture notes in Introduction to AI.* Czech Technical University, April 2021.

[72] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall Press, USA, 3rd edition, 2009.

[73] Satinder Singh, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvári. Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms. *Machine Learning*, pages 287–308, 2000.

[74] 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*, 518:529–533, 2015.

[75] E. Todorov, T. Erez, and Y. Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012.

[76] OpenAI. Robogym, 2020. URL: `github.com/openai/robogym`.

[77] Benjamin Ellenberger. PyBullet Gymperium, 2019. URL: `github.com/benelot/pybullet-gym`.

[78] OpenAI. Roboschool, 2017. URL: `github.com/openai/roboschool`.

[79] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. *arXiv*, 2018.

[80] Antonin Raffin, Ashley Hill, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, and Noah Dormann. Stable Baselines 3, 2019. URL: `github.com/DLR-RM/stable-baselines3`.

[81] Andrew G. Barto, Richard S. Sutton, and Charles W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, pages 834–846, 1983.

[82] Osbert Bastani, Yewen Pu, and Armando Solar-Lezama. Verifiable Reinforcement Learning via Policy Extraction. *CoRR*, 2018.

[83] OpenAI. openai/gym, 2021. URL: `github.com/openai/gym`.

[84] Richard S. Sutton. Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding. In *NIPS*, 1995.

[85] Andrew William Moore. Efficient Memory-Based Learning for Robot Control. 1990.

Appendices

Appendix A Genome types experiment – Figures

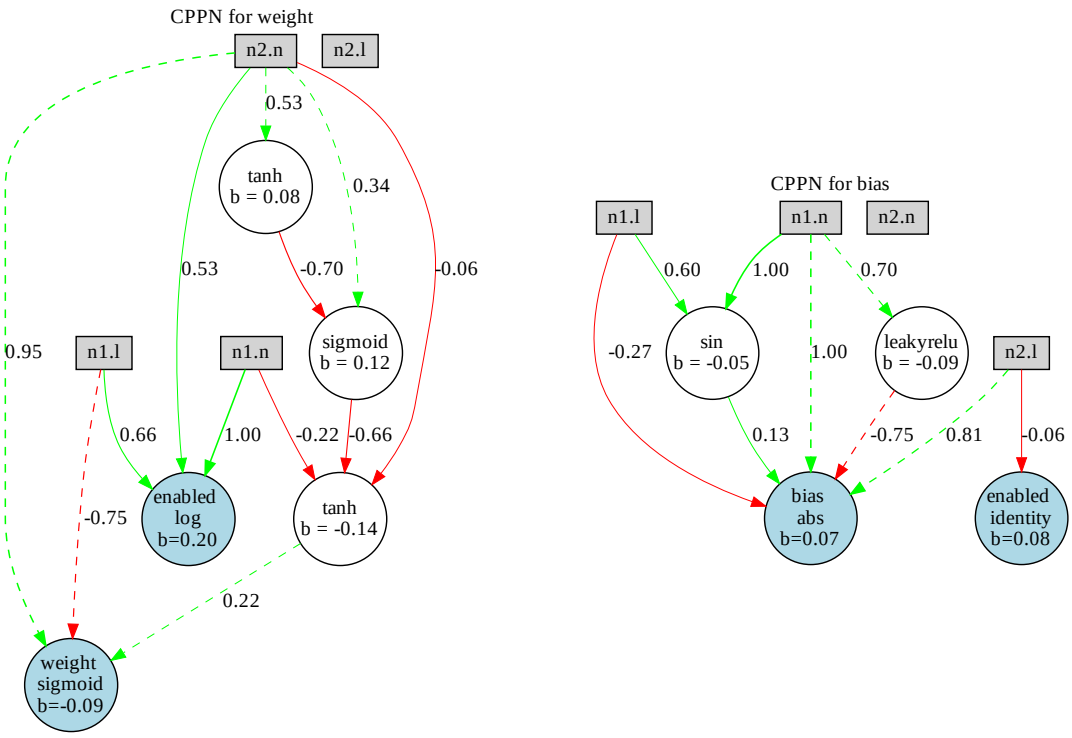


Figure 14: Example of an evolved genome with a more complex topology that is converted to the corresponding CPPNs. This genome was evolved in experiment described in Section 6.4.1 and tested on Acrobot environment. The dashed connections are not activated. The green connections have a positive weights, the red connections have a negative weight.

Appendix B DQN policy initialisation – Figures

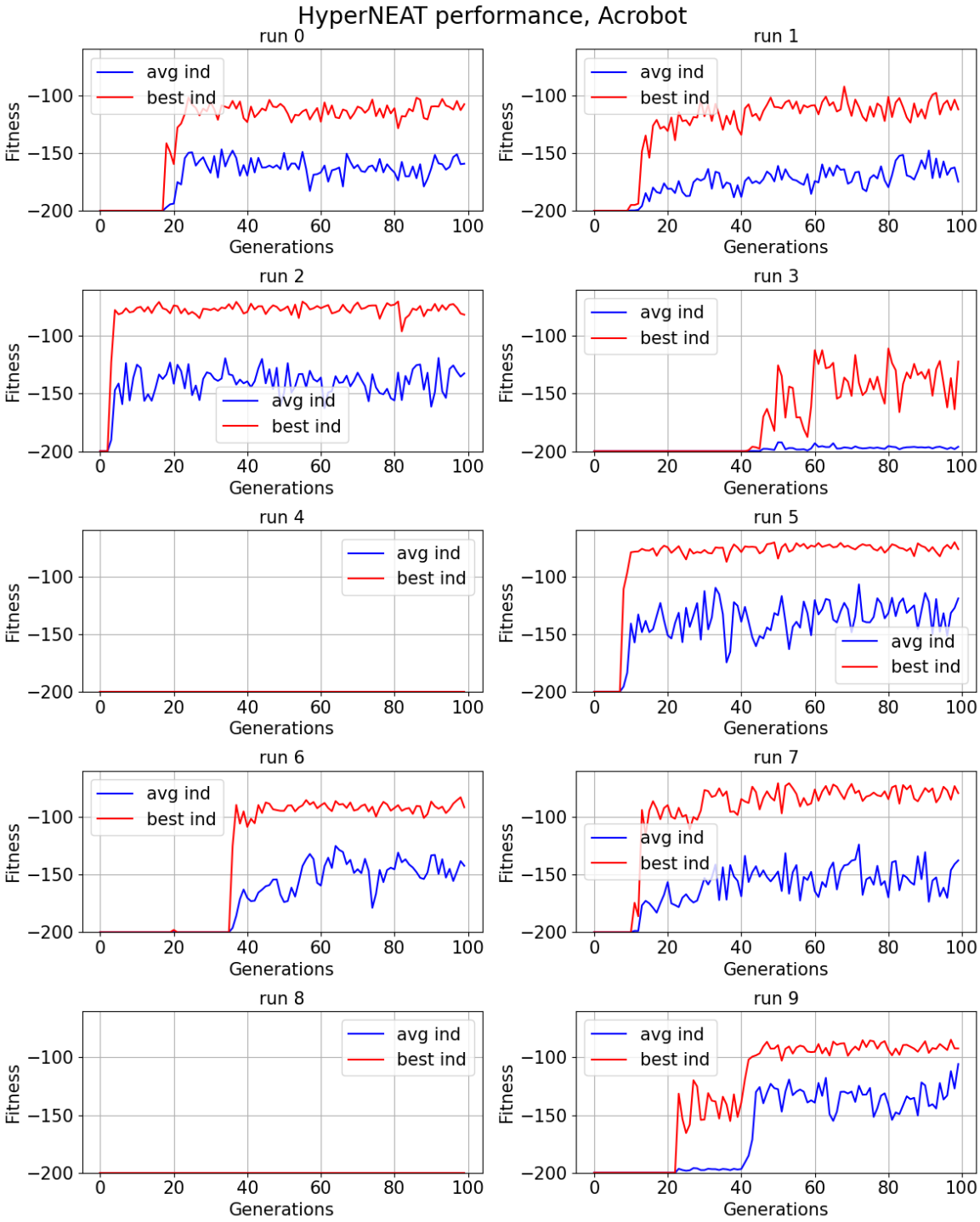


Figure 15: Experiment described in Section 6.4.2: HyperNEAT→DQN with DQN policy network initialisation, HyperNEAT part.

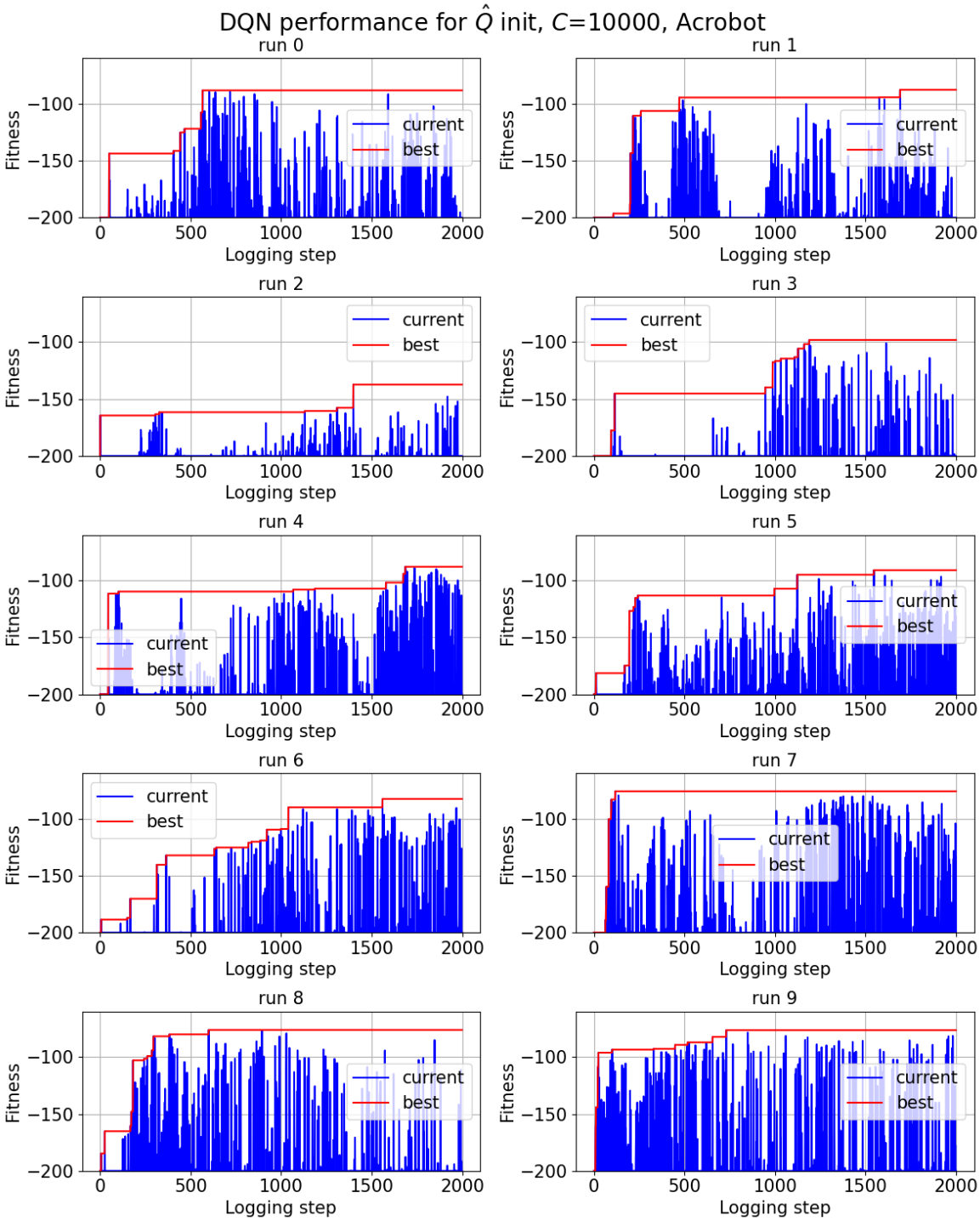


Figure 16: Experiment described in Section 6.4.2: HyperNEAT→DQN with DQN policy network initialisation, DQN part.

Appendix C External policy initialisation – Figures

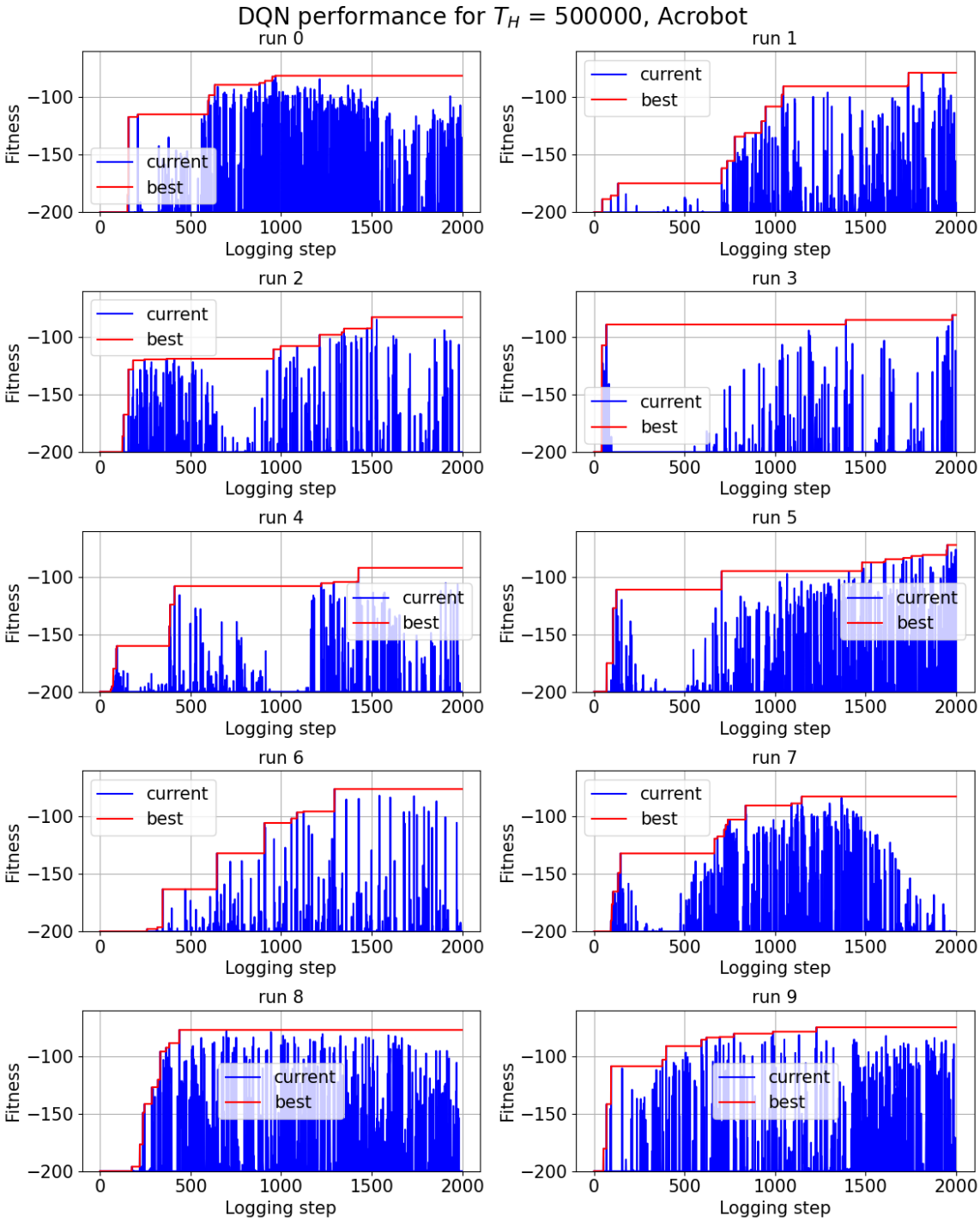


Figure 17: Experiment described in Section 6.4.4: HyperNEAT→DQN with external policy H. The external policy was used for the first $T_H = 500000$ time steps which corresponds to the first 250 logging steps.

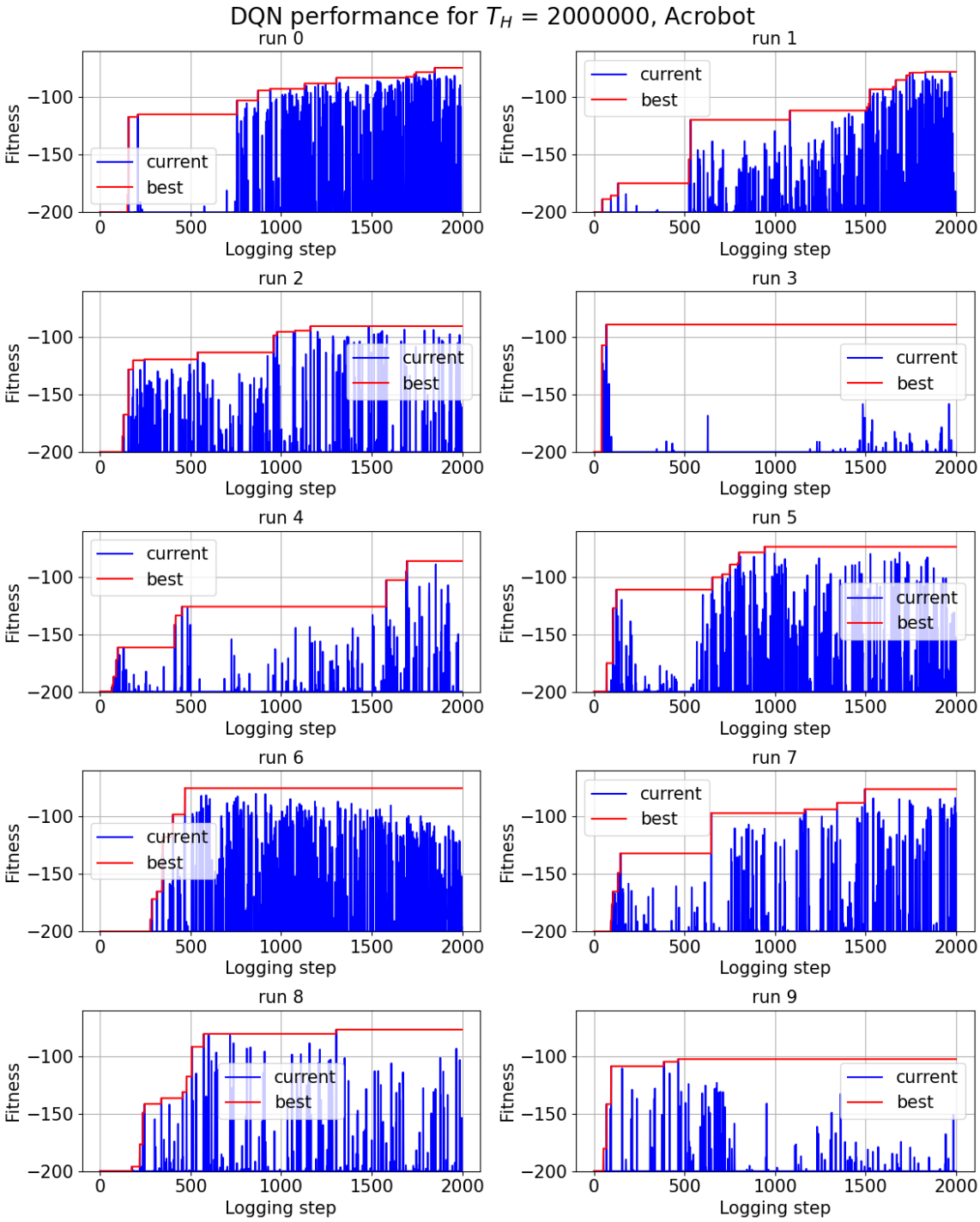


Figure 18: Experiment described in Section 6.4.4: HyperNEAT→DQN with external policy H. The external policy was used for the first $T_H = 2000000$ time steps which corresponds to the first 1000 logging steps.

Appendix D HyperNEAT→DQN loop – Tables, Figures

T^i_{max}	q	T_H	done	T_{avg}	$\frac{T_{avg}}{T_{max}}$%	T_{std}	T_{med}	R_{avg}	R_{std}	R_{med}	R_{max}	R_{min}
Acrobot, DQN												
4e5	0.01	2e5	0	8000000	100.0	0	8000000	-78	4	-77	-73	-86
4e5	0.5	2e5	0	8000000	100.0	0	8000000	-78	4	-77	-73	-87
4e5	0.75	2e5	0	8000000	100.0	0	8000000	-77	4	-76	-73	-87
4e5	0.9	2e5	0	8000000	100.0	0	8000000	-79	4	-78	-73	-87
8e5	0.01	4e5	0	8000000	100.0	0	8000000	-84	7	-85	-71	-99
8e5	0.5	4e5	0	8000000	100.0	0	8000000	-84	7	-84	-71	-99
8e5	0.75	4e5	0	8000000	100.0	0	8000000	-84	7	-84	-71	-99
8e5	0.9	4e5	0	8000000	100.0	0	8000000	-83	6	-82	-71	-94
16e5	0.01	8e5	0	8000000	100.0	0	8000000	-81	3	-81	-75	-89
16e5	0.5	8e5	0	8000000	100.0	0	8000000	-81	3	-82	-75	-84
16e5	0.75	8e5	0	8000000	100.0	0	8000000	-81	3	-82	-75	-84
16e5	0.9	8e5	0	8000000	100.0	0	8000000	-80	3	-81	-73	-84
4e6	0.01	2e6	0	8000000	100.0	0	8000000	-82	8	-79	-73	-101
4e6	0.5	2e6	0	8000000	100.0	0	8000000	-84	8	-84	-73	-101
4e6	0.75	2e6	0	8000000	100.0	0	8000000	-84	9	-84	-73	-101
4e6	0.9	2e6	0	8000000	100.0	0	8000000	-84	9	-84	-73	-101
Acrobot, HyperNEAT												
4e5	0.01	2e5	0	6474555	80.93	696695	6395349	-78	10	-70	-69	-100
4e5	0.5	2e5	0	6474454	80.93	696823	6395349	-78	10	-70	-69	-100
4e5	0.75	2e5	0	6473793	80.92	696819	6392046	-78	10	-70	-69	-100
4e5	0.9	2e5	0	6469811	80.87	694830	6389987	-78	10	-70	-69	-100
8e5	0.01	4e5	0	6444218	80.55	697855	6451861	-77	9	-70	-69	-95
8e5	0.5	4e5	0	6444218	80.55	697855	6451861	-77	9	-70	-69	-95
8e5	0.75	4e5	0	6444147	80.55	697860	6451604	-77	9	-70	-69	-95
8e5	0.9	4e5	0	6440439	80.51	697386	6441371	-77	9	-70	-69	-95
16e5	0.01	8e5	0	6215327	77.69	518156	6057495	-72	6	-70	-68	-85
16e5	0.5	8e5	0	6215394	77.69	518162	6057800	-72	6	-70	-68	-85
16e5	0.75	8e5	0	6215298	77.69	518004	6057800	-72	6	-70	-68	-85
16e5	0.9	8e5	0	6214654	77.68	517083	6056054	-72	6	-70	-68	-85
4e6	0.01	2e6	0	6556365	81.95	861981	6433503	-91	38	-73	-69	-200
4e6	0.5	2e6	0	6486373	81.08	881009	6248684	-93	39	-74	-69	-200
4e6	0.75	2e6	0	6556484	81.96	861849	6433503	-91	38	-73	-69	-200
4e6	0.9	2e6	0	6556493	81.96	861829	6433503	-91	38	-73	-69	-200

Table 12: Experiment described in Section 6.4.6, results for $T_{max} = 16000000$, both of the algorithms were given half of the T_{max} budget.

T^i_{max}	q	T_H	done	T_{avg}	$\frac{T_{avg}}{T_{max}}$%	T_{std}	T_{med}	R_{avg}	R_{std}	R_{med}	R_{max}	R_{min}
Acrobot, DQN												
4e5	0.01	2e5	0	4000000	100.0	0	4000000	-89	11	-86	-76	-112
4e5	0.5	2e5	0	4000000	100.0	0	4000000	-92	11	-86	-78	-112
4e5	0.75	2e5	0	4000000	100.0	0	4000000	-92	11	-86	-78	-112
4e5	0.9	2e5	0	4000000	100.0	0	4000000	-92	11	-86	-78	-108
8e5	0.01	4e5	0	4000000	100.0	0	4000000	-90	12	-85	-78	-118
8e5	0.5	4e5	0	4000000	100.0	0	4000000	-92	15	-84	-78	-122
8e5	0.75	4e5	0	4000000	100.0	0	4000000	-92	15	-84	-78	-122
8e5	0.9	4e5	0	4000000	100.0	0	4000000	-89	13	-82	-78	-122
16e5	0.01	8e5	0	4000000	100.0	0	4000000	-87	7	-84	-77	-100
16e5	0.5	8e5	0	4000000	100.0	0	4000000	-86	9	-84	-75	-103
16e5	0.75	8e5	0	4000000	100.0	0	4000000	-86	9	-84	-75	-103
16e5	0.9	8e5	0	4000000	100.0	0	4000000	-85	7	-83	-75	-102
4e6	0.01	2e6	0	4000000	100.0	0	4000000	-97	32	-86	-73	-173
4e6	0.5	2e6	0	4000000	100.0	0	4000000	-101	32	-89	-73	-173
4e6	0.75	2e6	0	4000000	100.0	0	4000000	-98	31	-89	-73	-173
4e6	0.9	2e6	0	4000000	100.0	0	4000000	-98	31	-89	-73	-173
Acrobot, HyperNEAT												
4e5	0.01	2e5	0	3388710	84.72	346031	3459711	-78	10	-70	-69	-100
4e5	0.5	2e5	0	3388710	84.72	346031	3459711	-78	10	-70	-69	-100
4e5	0.75	2e5	0	3388049	84.7	345914	3456408	-78	10	-70	-69	-100
4e5	0.9	2e5	0	3387082	84.68	344335	3456885	-78	10	-70	-69	-100
8e5	0.01	4e5	0	3432460	85.81	441206	3363858	-104	48	-84	-69	-200
8e5	0.5	4e5	0	3432460	85.81	441206	3363858	-104	48	-84	-69	-200
8e5	0.75	4e5	0	3432408	85.81	441231	3363858	-104	48	-84	-69	-200
8e5	0.9	4e5	0	3431385	85.78	441274	3363858	-104	48	-84	-69	-200
16e5	0.01	8e5	0	3386959	84.67	363561	3317717	-99	50	-71	-68	-200
16e5	0.5	8e5	0	3386955	84.67	363566	3317717	-99	50	-71	-68	-200
16e5	0.75	8e5	0	3386955	84.67	363566	3317717	-99	50	-71	-68	-200
16e5	0.9	8e5	0	3386549	84.66	363591	3315707	-99	50	-71	-68	-200
4e6	0.01	2e6	0	3446845	86.17	420841	3387507	-108	47	-88	-70	-200
4e6	0.5	2e6	0	3382717	84.57	394540	3376009	-98	38	-84	-70	-200
4e6	0.75	2e6	0	3446845	86.17	420841	3387507	-108	47	-88	-70	-200
4e6	0.9	2e6	0	3446845	86.17	420841	3387507	-108	47	-88	-70	-200

Table 13: Experiment described in Section 6.4.6, results for $T_{max} = 8000000$, both of the algorithms were given half of the T_{max} budget.

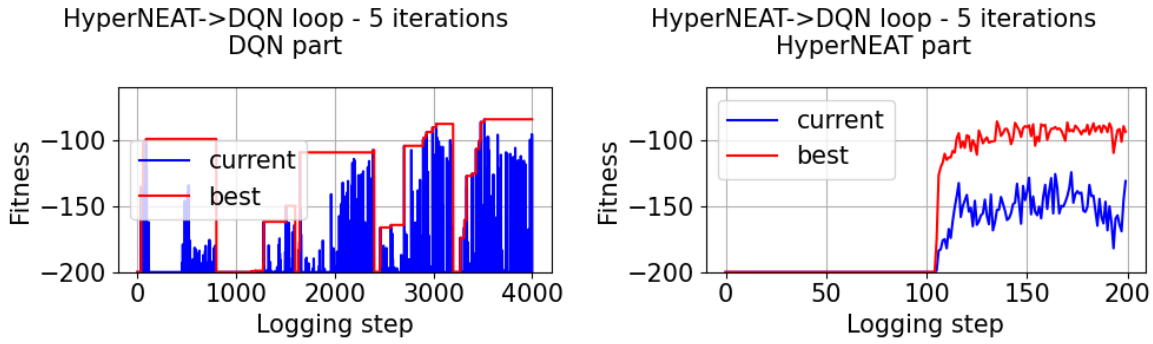


Figure 19: Experiment described in Section 6.4.6, example of one run of HyperNEAT→DQN loop with the merged statitistics, 5 iterations were performed

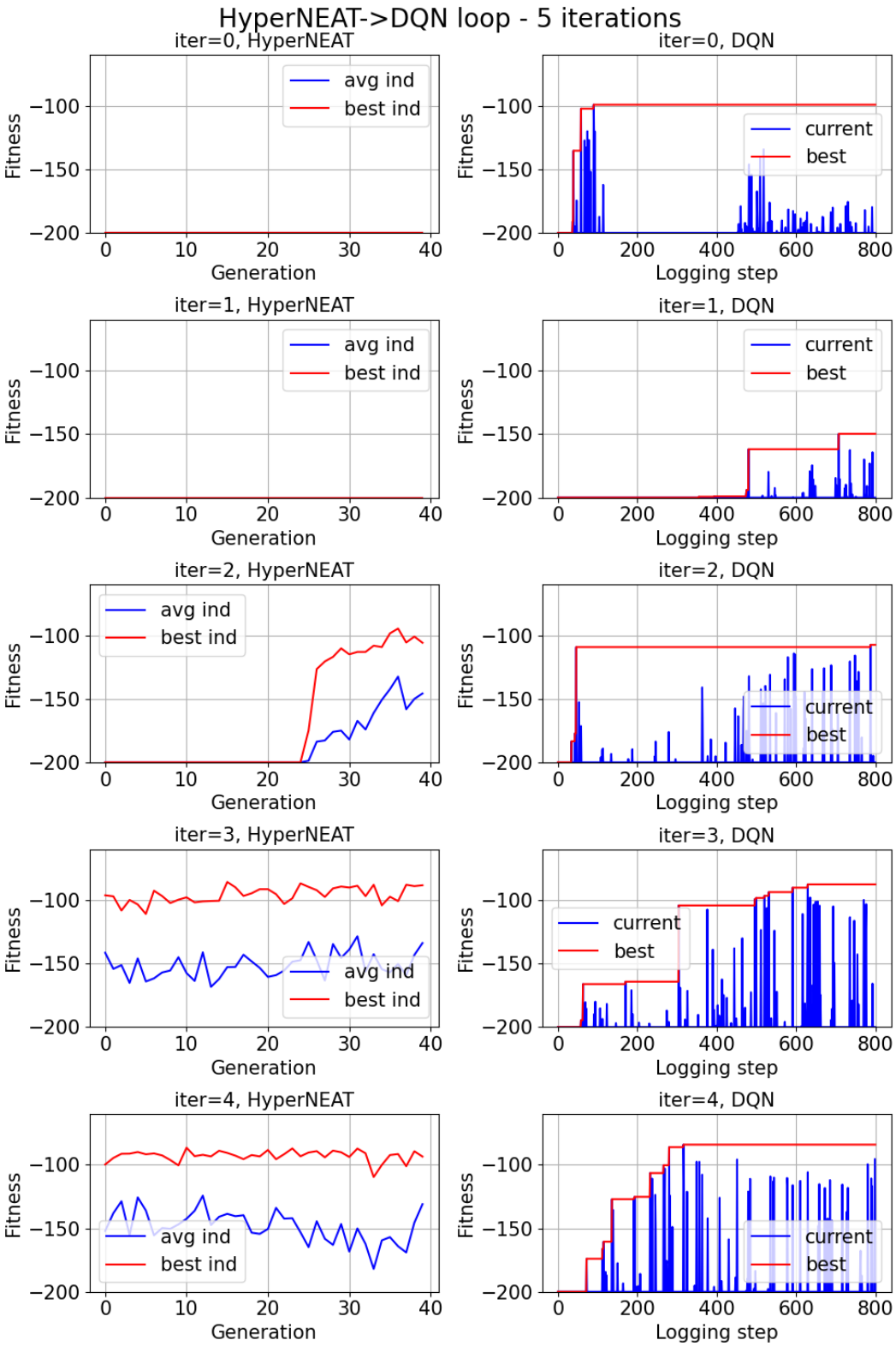


Figure 20: Experiment described in Section 6.4.6, example of one run of HyperNEAT→DQN loop with the original statitistics, 5 iterations were performed

Appendix E NEAT configuration file

```
[MultiGenome]
    num_inputs              = 4
    num_hidden              = 0
    initial_connection      = partial_nodirect 0.5
    feed_forward            = True
    compatibility_disjoint_coefficient = 1.0
    compatibility_weight_coefficient   = 3.0
    conn_add_prob           = 0.03
    conn_delete_prob        = 0.005
    node_add_prob           = 0.02
    node_delete_prob        = 0.005
    activation_options      = relu sigmoid tanh sin log abs square
                              identity cube gauss clip leakyrelu selu
    activation_mutate_rate  = 0.1
    aggregation_default     = sum
    aggregation_options     = sum
    aggregation_mutate_rate = 0.0
    bias_init_mean          = 0.0
    bias_init_stdev         = 0.1
    bias_replace_rate       = 0.005
    bias_mutate_rate        = 0.4
    bias_mutate_power       = 0.01
    bias_max_value          = 30.0
    bias_min_value          = -30.0
    response_init_mean      = 1.0
    response_init_stdev     = 0.0
    response_replace_rate   = 0.0
    response_mutate_rate    = 0.1
    response_mutate_power   = 0.01
    response_max_value      = 1.0
    response_min_value      = -1.0
    weight_max_value        = 1
    weight_min_value        = -1
    weight_init_mean        = 0.0
    weight_init_stdev       = 1.0
    weight_mutate_rate      = 0.94
    weight_replace_rate     = 0.005
    weight_mutate_power     = 0.1
    enabled_default         = True
    enabled_mutate_rate     = 0.01
    single_structural_mutation = True
[DefaultSpeciesSet]
    compatibility_threshold = 4.0
[DefaultStagnation]
    species_fitness_func = max
    max_stagnation       = 15
    species_elitism      = 1
[MultiReproduction]
    elitism            = 1
    survival_threshold = 0.2
```