Optimising the Optimiser: Meta NeuroEvolution for Artificial Intelligence Problems

A dissertation presented to University of Cape Town in full fulfilment of the degree:
MSc. Advanced Analytics
UNIVERSITY OF CAPE TOWN

Supervisor: Bruce A. Bassett[1] and
Co-supervisor: Allan E. Clark [2]

Max Nieuwoudt Hayes

February 2021

[1]the University of Cape Town, Department of Mathematics and Applied Mathematics
[2]University of Cape Town, Department of Statistical Sciences

Abstract

Since reinforcement learning algorithms have to fully solve a task in order to evaluate a set of hyperparameter values, conventional hyperparameter tuning methods can be highly sample inefficient and computationally expensive. Many widely used reinforcement learning architectures originate from scientific papers which include optimal hyperparameter values in the publications themselves, but do not indicate how the hyperparameter values were found.

To address the issues related to hyperparameter tuning, three different experiments were investigated. In the first two experiments, Bayesian Optimisation and random search are compared. In the third and final experiment, the hyperparameter values found in second experiment are used to solve a more difficult reinforcement learning task, effectively performing hyperparameter transfer learning (later referred to as meta-transfer learning).

The results from experiment 1 showed that there are certain scenarios in which Bayesian Optimisation outperforms random search for hyperparameter tuning, while the results of experiment 2 show that as more hyperparameters are simultaneously tuned, Bayesian Optimisation consistently finds better hyperparameter values than random search. However, BO took more than twice the amount of time to find these hyperparameter values than random search. Results from the third and final experiment indicate that hyperparameter values learned while tuning hyperparameters for a relatively easy to solve reinforcement learning task (Task A), can be used to solve a more complex task (Task B). With the available computing power for this thesis, hyperparameter optimisation was possible on the tasks in experiment 1 and experiment 2. This was not possible on the task in experiment 3, due to limited computing resources and the increased complexity of the reinforcement learning task in experiment 3, making the transfer of hyperparameters from one task (Task A) to the more difficult task (Task B) highly beneficial for solving the more computationally expensive task.

The purpose of this work is to explore the effectiveness of Bayesian Optimisation as a hyperparameter tuning algorithm on the reinforcement learning algorithm NEAT's hyperparemters. An additional goal of this work is the experimental use of hyperparameter value transfer between reinforcement learning tasks, referred to in this work as Meta-Transfer Learning. This is introduced and discussed in greater detail in the Introduction chapter.

All code used for this work is available in the repository:

- `https://github.com/maaxnaax/MSc_code`

Acknowledgements

I would like to thank my parents, Helen Hayes and Pierre Nieuwoudt, my Godparents, Katherine Evans and Paul Evans, and my supervisor, Bruce Bassett, and my co-supervisor Allan Clark. I would like to thank you for your support, your belief in me and your appreciation of my work. A big thank you for the motivation that you have given me throughout the completion of this dissertation, and for highlighting which areas would be interesting to pursue further. My goal has been to do research that will be useful to the machine learning research community.

Declaration of authorship

I, Max Niewoudt Hayes, know the meaning of plagiarism and declare that all of the work in the document, save for that which is properly acknowledged, is my own.

I, Max Hayes, declare that this thesis titled, *"Optimising the Optimiser: MetaNeroEvolution for Artificial IntelligenceProblems"* and the work presented in it are my own. I confirm that:

1. know that plagiarism is wrong. Plagiarism is to use anothers work and pretend that it is one's own.
2. I have used the Harvard convention for citation and referencing. Each contribution to, and quotation in, this dissertation from the work(s) of other people has been attributed, and has been cited and referenced. Any section taken from an internet source has been referenced to that source.
3. This dissertation is my own work, and is in my own words (except where I have attributed it to others).
4. I have not allowed, and will not allow, anyone, to copy my work with the intention of passing it off as his or her own work.

Signed:

Max Hayes

Date: 21/10/2021

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine Learning is an area of research focused on how computer programs can automatically learn from data and improve from experience, without being explicitly programmed to do so [1]. Building computer programs in this way has allowed researchers to tackle increasingly complex tasks. As the complexity of machine learning tasks increases, so does the complexity of the machine learning algorithms used to solve the tasks. This is especially true of a subset of machine learning tasks called *Reinforcement Learning* tasks. Reinforcement learning is essentially the branch of machine learning that deals with methods for finding optimal behavioral strategies to solve tasks. The task for example could be to play Chess, or to learn behaviour that can safely control a car. Reinforcement learning is then goal-directed learning from interaction with an environment[2] (the task at hand).

In order to reduce the amount of time required to solve a reinforcement learning task, and to improve solutions to tasks, two techniques are explored in this thesis, namely hyperparameter tuning and transfer learning, detailed below:

1. *hyperparameter Tuning.* Effectively developing machine learning models requires that one not only has methods for learning suitable model parameter values but also methods for learning hyperparameter values. Learning suitable hyperparameter values for a given machine learning task is called hyperparameter tuning [26].

2. *Transfer Learning.* Sometimes you can take knowledge, that a machine learning algorithm has learned from one task (Task A), and apply that knowledge to a separate task (Task B) [3]. This transfer of information is a useful way of preserving knowledge that may

have been costly to learn. This is called transfer learning, and typically model parameters learned for the initial task (Task A) are reused as the starting point for a model on a second, previously unseen, task (Task B).

Both of these techniques offer a means to improve existing machine learning algorithms

In this work, the techniques listed above are used in three different experiments. The purpose of the first two experiments in this thesis is to examine the effectiveness of hyperparameter tuning on reinforcement learning problems, with the end goal of finding hyperparameter sets that reduce the number of iterations until convergence to a solution for a given task, and to increase the quality of the solution.

The purpose of the third and final experiment is to explore the use of a combination of transfer learning and hyperparameter tuning when applied to reinforcement learning, by using hyperparameter values found while solving one task (Task A), on a new previously unseen task (Task B). This is a form of *Meta-Transfer Learning*. Again, transfer learning is where **model parameters** learned for the initial task (Task A) are reused as the starting point for a model on a second task (Task B), while, when performing meta-transfer learning, **algorithm parameters** are instead reused.

Bayesian Optimisation[1] is employed as a hyperparameter tuning algorithm to tune the hyperparameters of a reinforcement learning framework called NeuroEvolution of Augmenting Topologies[2]. Later, the results obtained using Bayesian Optimisation are compared to results obtained using a random search. Finally, hyperparameter values learned while solving one task are used to solve a harder task. This is useful because hyperparameter tuning is typically a difficult exercise.

Hyperparameter tuning can be difficult because there is usually no way of fully representing the performance of a machine learning algorithm, as a function of its hyperparameter values, with a mathematical formula. Consequently, we don't have the derivative of that function, making hyperparameter tuning tricky since most classical optimisation methods rely on gradient information. The two methods used to tune hyperparameters address this problem by either randomly exploring - negating the need for an explicit formula - or by building a surrogate function to be used as a proxy for the aforementioned formula.

Another problem is the ever-increasing cost of computations as reinforcement learning

[1] Introduced in detail in section **??**
[2] Introduced in detail in section **??**

tasks become more complicated. This makes meta-transfer learning more attractive. Optimal hyperparameter values can be learned while solving one task and then reused to solve a more computationally expensive task. That is, a task whose hyperparameter value tuning would otherwise require more computing time/power. Over the next few chapters (Chapter 2 to Chapter 5) the machine learning principles used in the three experiments are introduced, starting with Gaussian Processes.

In summary, the efficacy of reinforcement learning algorithms ability to solve reinforcement learning tasks is highly dependant on the algorithms hyperparameters and a delicate balance between them needs to be found. This dissertation aims to address this hyperparameter problem using Bayesian Optimisation as a means to find appropriate combinations of the different values for reinforcement learning algorithms hyperparemeters in order to minimise the computation cost associated with solving reinforcement learning tasks. The final experiment in this work studies the efficacy of using hyperparameter values found while solving a previous experiment on a new unseen task.

Chapter 2

Gaussian Processes

2.1 Introduction

This Chapter serves as an introduction to Gaussian Processes, which will be used with Bayesian Optimisation to find hyperparameter values for reinforcement learning tasks in the experiment Chapters. A Gaussian Distribution (usually referred to as a Normal distribution) is a symmetric bell-shaped probability distribution characterized by the distribution's mean μ and its variance σ^2. As a result of its symmetry, when drawing samples from a Gaussian distribution it is expected that there will be an equal number of observations higher and lower than the mean [9]. A *Gaussian Process* (GP) is a family of Gaussian distributions - an infinite number of Gaussian distributions - indexed by some control parameter x, where the mean and variance of each of the Gaussian distributions are correlated with one another[9].

From Figure 1, the mean of the GP describes a function. GPs can be thought of as distributions over functions where at each input point x the output is a Gaussian distribution, $f(x)$, assigned by the GP. $f(x)$ is a probability distribution of the possible values that a function could take at that point. The most likely value that a function will take at a given location x is the mean $m(x)$ of the Gaussian distribution $f(x)$, but the GP describes infinitely many functions [6]. During GP regression, a GP is conditioned on data in order to narrow down a collection of functions to a collection of functions that pass through a set of points, observed from a function that we

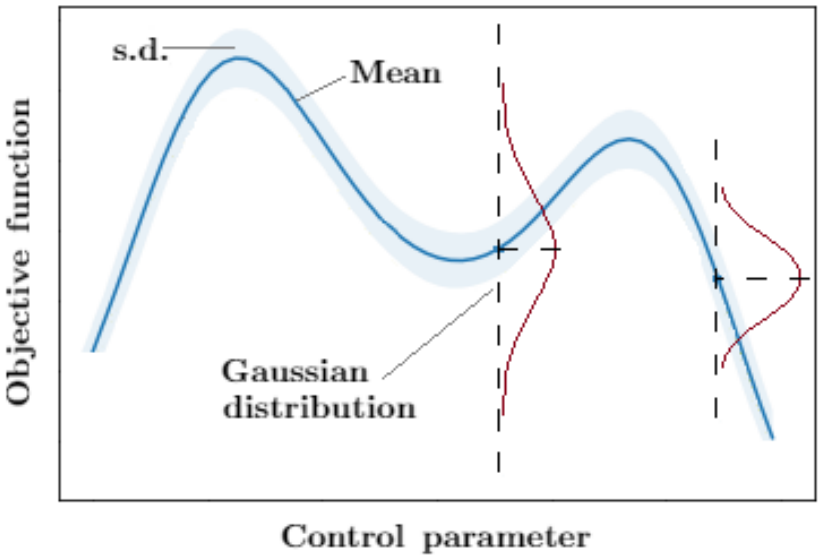


Figure 2.1: Two arbitrarily chosen points x_1 and x_2 and their corresponding bell shaped Gaussian distributions $f(x_1)$ and $f(x_2)$ (in maroon) are superimposed over a GP, made using code from [4].

wish to perform regression on [7].

2.2 Formal Definition

A GP is formally defined as a stochastic process that generates data located throughout some domain such that any finite subset of the range follows a Multivariate Gaussian distribution[6]. Let $\mathbb{R}$ be the domain of a GP. The GP maps any point $x_i \in \mathbb{R}$ to a Gaussian random variable $f(x_i)$ [6]. GPs can then be thought of as a generalization of the Multivariate Gaussian distribution to infinite dimensions. Instead of being parametrised by a mean vector and a covariance matrix, for any $x, x' \in \mathbb{R}$ a GP is parameterised by a mean function and a covariance function. Following [7] these functions are denoted:

$$m(x)$$

$$K\left(x, x'\right)$$

respectively. The GP defined with respect to the above functions is denoted:

$$f(x) \sim \mathcal{GP}\left(m(x), K\left(x, x'\right)\right), \forall x, x' \in \mathbb{R}$$

2.3 Covariance Function

Any positive definite function can be used as the covariance function. In this work the squared exponential kernel is used throughout as the covariance function. The squared exponential kernel was chosen because of its parameters l and σ_f which allow us to control the horizontal variation, or *wiggliness*, and vertical variation of the infinite set of functions defined by the GP [7]. For any two points $x, x' \in \mathbb{R}$, using the squared exponential kernel, their covariance is defined as:

$$\operatorname{cov}\left(f\left(x\right), f\left(x'\right)\right) = K\left(x, x'\right) = \sigma_f^2 \exp\left(-\frac{1}{2l^2}\left(x - x'\right)^T\left(x - x'\right)\right)$$

GPs as sets of infinitely many functions are discussed in detail in the next subsection.

Later in this chapter, a GP prior (To be introduced in Section 2.5.2) is used for GP regression. [8] states that setting the mean function to $m(x) = 0$ will suffice for regression on any data set since GPs are flexible enough to model the mean arbitrarily well. As a result of this GPs are usually only characterized by their choice of the covariance function. In practice, it is useful to normalise observations to have a mean zero for quicker convergence. Since the mean is initially set to zero, the covariance function can be simplified:

$$k\left(x, x'\right) = \mathbb{E}\left[f(x) f\left(x'\right)\right]$$

Here $\mathbb{E}$ is an expectation, which is essentially the arithmetic mean of a random variable taken in the infinite limit. Refer to [9] for more details on expectations and how they are calculated .

2.4 Distributions Over Functions

As previously mentioned, there are multiple ways of interpreting Gaussian Processes. Using the "Function Space View", a single GP can be thought of as describing a distribution over functions,

where inference can take place directly in function space[7]. Once this function space has been visualized, it is easier to grasp. To illustrate the function-space view of a GP we choose a finite number of evenly spaced input points $\mathbf{X}_* \subseteq [-5, 5]$, and record the corresponding covariance matrix $K(\mathbf{X}_*, \mathbf{X}_*)$ elementwise using the squared exponential kernel as the covariance function. Following [7] we sample a random vector f_* using this covariance matrix.

$$\mathrm{f}_* \sim \mathcal{N}(\mathbf{0}, K(\mathbf{X}_*, \mathbf{X}_*))$$

The process of sampling f_* from the distribution is repeated three times and the resulting sampled functions are shown in Figure 2.4 labeled 'Sample 1', 'Sample 2', and 'Sample 3'.

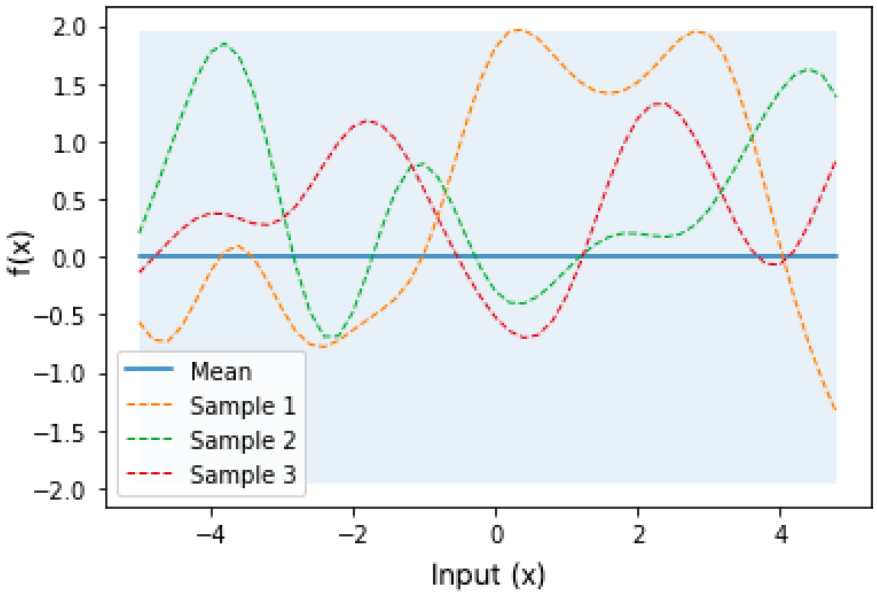


Figure 2.2: A GP prior (To be introduced in Section 2.5.2) along with three sample functions drawn from the GP prior and plotted. The light blue shaded region denotes twice the standard deviation at each input value x of the GP (GP prior) and the dark blue line represents the mean. The GP is a distribution over functions that assigns, for every x, a Gaussian distribution that represents the probability that a function will take on some value at the point.

2.5 Gaussian Process Regression

Gaussian Process Regression (GPR) is a machine learning technique that makes use of a GP to infer a specific distribution over functions to model some objective function y [9]. To infer this distribution, first, a GP prior (To be introduced in Section 2.5.2) is chosen by specifying the covariance function, then after having observed some values of the objective function y, that we wish to regress, the prior is updated to a posterior over functions. The posterior is then used to generate predictions [9].

For the rest of this chapter, and the Bayesian Optimisation chapter, consider a hidden objective function $y : \mathbf{X} \longmapsto \mathbb{R}$ with $y = -\sin(3x) - x^2 + 0.7x$ and domain $\mathbf{X} = [-1, 2]$ that we wish to perform regression on[1]. Note that this function has one global maximum but two local maxima on the interval.

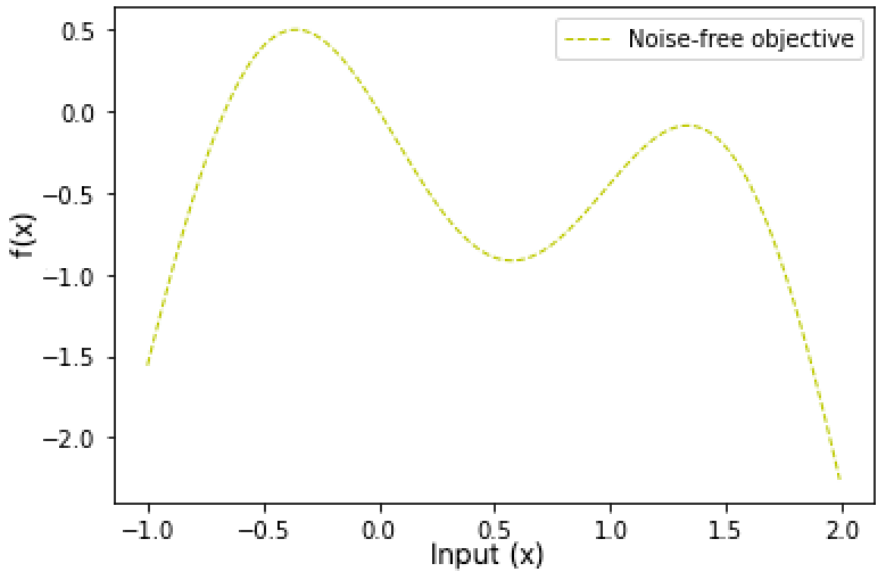


Figure 2.3: Noise-free hidden objective function

[1]This is a toy problem used to illustrate GP regression, and is not one of the three experiments for this dissertation.

2.5.1 Notation

When fitting a GP the output from the objective function is normalised to agree with the $m(x) = 0$. At this point, it is worth mentioning again that a GP maps any point x_i in the domain of y to a Gaussian random variable $f(x_i)$. GPR uses the random variable $f(x_i)$ to represent the possible output of the function y at x_i.

Let $\mathbf{y}$ be a training vector of $n \in \mathbb{N}$ observations at points $\mathbf{X}$. If observations are noisy then $\mathbf{y} = y(\mathbf{X}) + \epsilon$, where $\epsilon \sim N\left(0, \sigma_n^2\right)$ is assumed to be *Independent and Identically Distributed* (IID) noise, while the actual observations $y(\mathbf{X})$ are assumed to covary. If observations are noise free then $\sigma_n = 0$.

Let X_* be a vector of x values in the domain that we wish to use GPR to infer $y(\mathbf{X}_*)$ over.

2.5.2 prior

Initially, the objective function y is completely unknown and our GP prior is defined using a zero mean and the squared exponential kernel, with parameters $l = 1$ and $\sigma_f = 1$, as the covariance function [7].

2.5.3 Posterior and Predictions

Let

$$\mathcal{D} = \{(x_i, y_i) \mid x_i \in X \text{ and } y_i = y(x_i)\ \forall i \in \{1, \ldots, n\}\}$$

be a training set of n noisy observations of the objective function y and their corresponding x values. A distribution over functions specifies an infinite prior set of functions [7]. In this phase of GPR, the GP prior is conditioned on the training data $\mathcal{D}$, effectively narrowing down the infinite set of functions to an infinite set of functions that pass through the observed data points $\mathcal{D}$, specified by the GP posterior distribution over functions. The posterior is then used to infer the values of the objective function at test locations X_*, denoted $\mathbf{f}_*$, using Bayesian inference. GPs, like Gaussian distributions, are closed under conditioning and marginalization which means that the posterior distribution is again Gaussian. Following [7] the joint distribution of the training

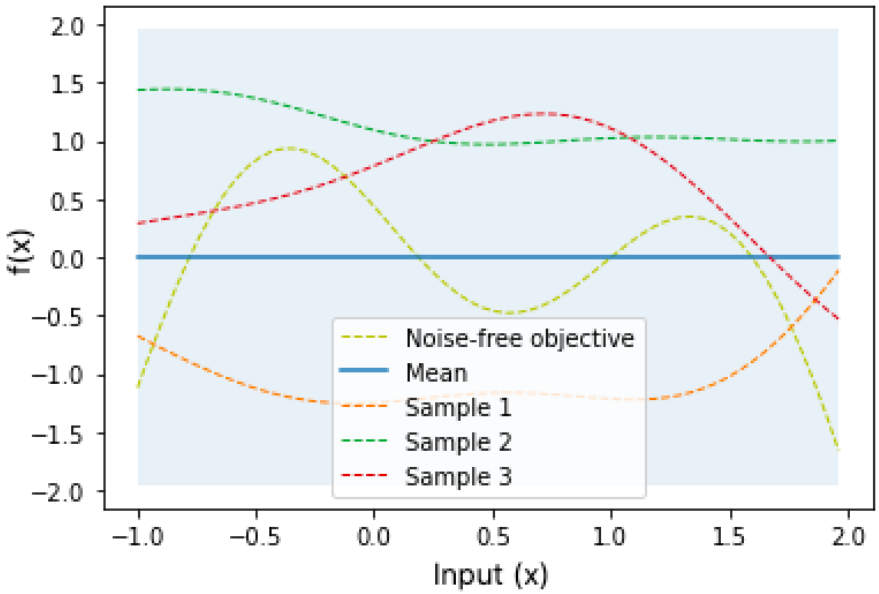


Figure 2.4: The GP prior along with three sample functions drawn from the GP prior are superimposed over the objective function y, illustrating the function view.

outputs, $\mathbf{f}$, and the test outputs $\mathbf{f}_*$ according to the prior is

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right)$$

[2] The marginal distribution of $\mathbf{f}_*$ can be calculated in closed form as[3]:

$$\mathbf{f}_* \mid X_*, X, \mathbf{y} \sim \mathcal{N}(\boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*)$$

where

$$\boldsymbol{\mu}_* = K(X_*,X)\left[K(X,X) + \sigma_n^2 I\right]^{-1}\mathbf{y}$$

$$\boldsymbol{\Sigma}_* = K(X_*,X_*) - K(X_*,X)\left[K(X,X) + \sigma_n^2 I\right]^{-1}K(X,X_*)$$

[2]Here N is a multivariate normal distribution with a zero mean and a covariance that has been broken up into the four smaller covariance matrices representing the covariance of the old data in the top left matrix, the covariance of the new data in the bottom right and the covariance of the new and old data with each other in the bottom left and top right matrices. Using the new data X_* in this way essentially allows us to include the new data in the GP in order to make inferences $\mathbf{f}_*$.

[3]see Appendix A.2 of [7] for a detailed proof

A training set $\mathcal{D}$ - A set of eight equally spaced points in the domain [-1,2] - and four different sets of noise parameter values σ_n and different covariance function kernel parameters l and σ_f are used to compare the different possible forms of the posterior distributions after marginalising plotted in Figure 2.5.

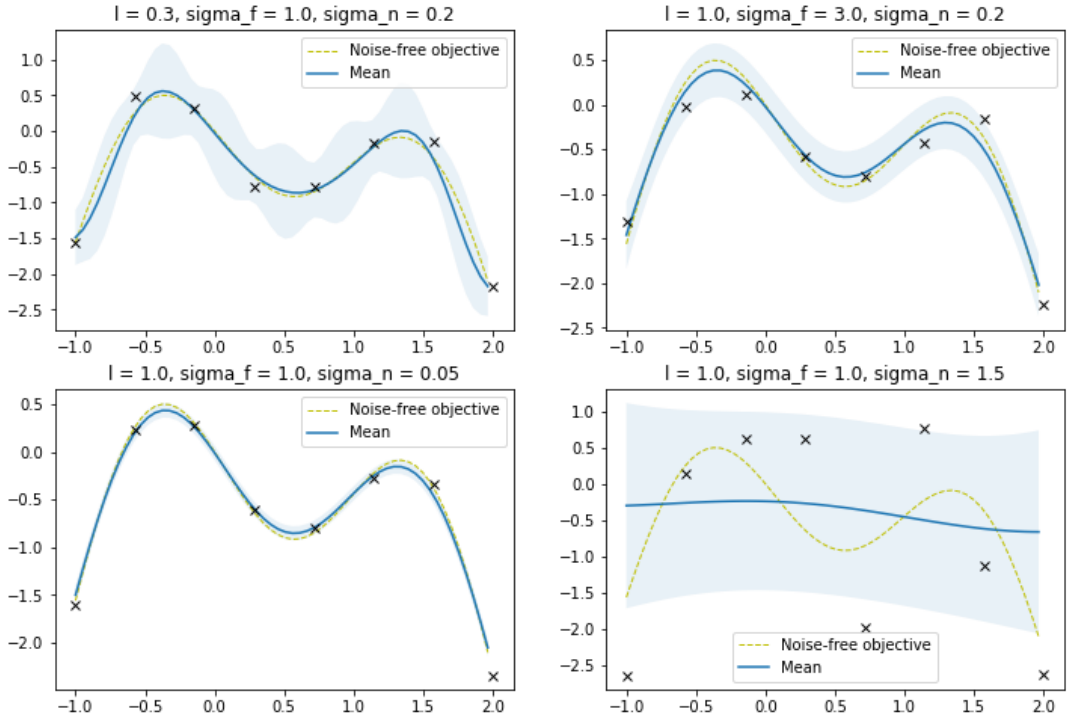


Figure 2.5: Four different GP posteriors, each with the same data, but different values for parameters σ_n, σ_f and l, illustrating the effect that the different values have on the shape and fit of the posterior GP. l and σ_f allow us to control the horizontal variation, or *wiggliness*, and vertical variation respectively of the GP. While σ_f is the variance of the (noise free) signal, while σ_n is the variance the variance of the noise [7]. Sampling multiple times at the same location can be used to estimate σ_n

2.5.4 Preventing Over and Under Fitting

Gaussian processes are prone to overfitting when datasets are too small. Even more so when prior knowledge of the covariance structure (wiggliness) is unknown. To prevent over or under fitting suitable estimates for parameters σ_f, σ_n and l need to be obtained. σ_n can be obtained by sampling the same x location multiple times and then averaging the responses. Using this estimate, or if σ_n is known, optimal values for l and σ_f can be estimated by maximizing the log marginal likelihood [7]. Let $\mathbf{K}_y = K(X, X) + \sigma_n^2 I$, then the log marginal likelihood of a GP is:

$$\log p(\mathbf{y} \mid \mathbf{X}) = \log \mathcal{N}(\mathbf{y} \mid \mathbf{0}, \mathbf{K}_y) = -\frac{1}{2}\mathbf{y}^T \mathbf{K}_y^{-1} \mathbf{y} - \frac{1}{2}\log |\mathbf{K}_y| - \frac{N}{2}\log(2\pi)$$

σ_n is known in this example since we constructed the objective function and we can impose that $\sigma_n = 0.1$ for the data generating process. Using $\sigma_n = 0.1$ the best parameters are obtained using the above log marginal likelihood and the best GP posterior for our objective function using the eight samples is plotted below. The best parameters should be found each time the GP is updated with new data.

2.5.5 Advantages and Limitations

Using a GP to approximate the output of a partially hidden objective function is an effective way of performing regression since sampling from this GP is computationally simple and it can be far less expensive to evaluate than the objective function itself. As previously stated GPs describe a Gaussian distribution $f(x)$ at each point x in the domain which describes the distribution of probable function values at that point allowing us to make predictions with uncertainty which is an advantage over other machine learning techniques such as Neural Networks if the uncertainty is important. However, there are limitations when implementing GPR. If the goal is to find the global maximum or minimum of the objective function with as few function evaluations as possible, GPR does not directly address this goal. GPR makes use of samples obtained from the objective function to fit the GP but a technique for guiding the choice of locations to observe has so far not been discussed.

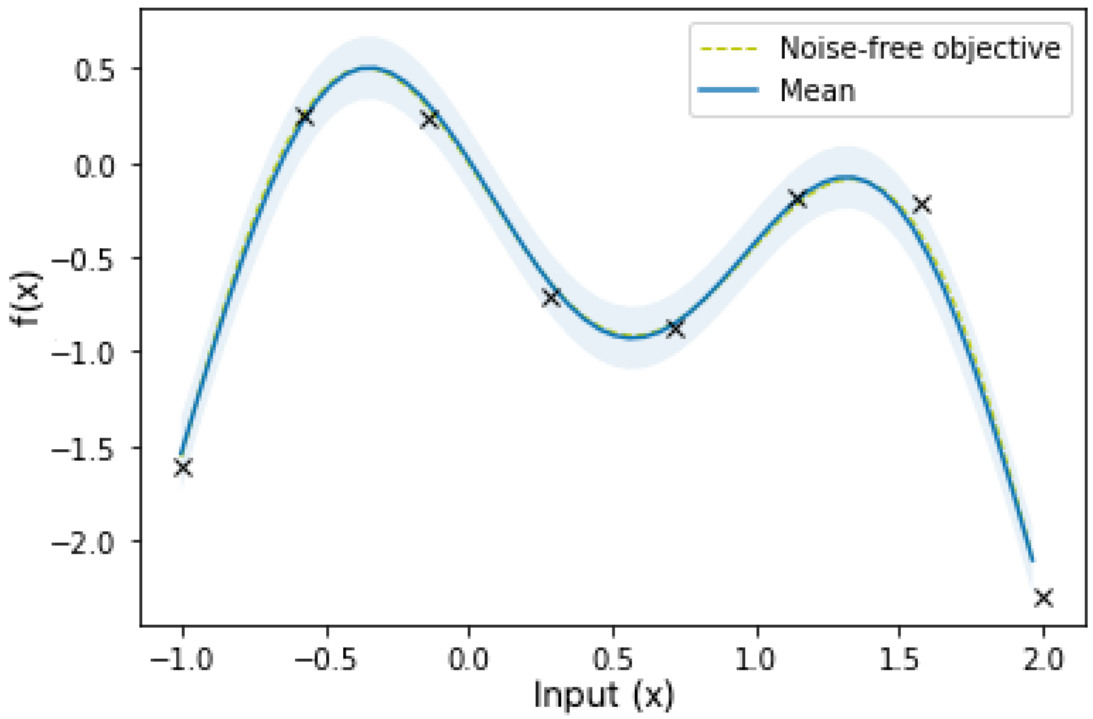


Figure 2.6: The GP posterior with parameters found by maximising the likelihood, illustrating the best GP posterior fit to the given data points made by following the tutorial[4]

Chapter 3

Bayesian Optimisation

This chapter introduces Bayesian Optimisation, an optimisation technique that typically employs Gaussian Processes which will be used to find optimal hyperparameter value sets for experiments 1 and 2 in this work. Consider the scenario in which we seek to find the global minimum of an objective function $y : \mathbf{X} \to \mathbb{R}$. This scenario is known as *Global Optimisation*. Typically one would either assume that y and its domain $\mathbf{X}$ are both convex - in which case convex optimisation is used to find the global minimum - or that either or both y and $\mathbf{X}$ are non-convex which is the case for most problems that deep learning seeks to address. Many deep learning techniques are gradient-based and rely on multiple evaluations of the objective function y to approximate its gradients.

Now consider the case in which y is an expensive to evaluate *Black Box* function - that is, its exact functional form is unknown - who's gradients would be too difficult to accurately estimate given the high cost associated with sampling. A suitable technique should then balance a trade-off between two goals [24]:

1. Exploration. Search for a global minimum, and
2. Exploitation. Using all current knowledge of the function, obtain a good solution in as few objective function evaluations as possible.

This trade-off is known as the explore exploit trade-off [24]. As machine learning tasks grow in complexity, the computational cost associated with exploring solutions to them also

grows, leading to problems where sampling is expensive. *Bayesian Optimisation* (BO) is a form of *Surrogate Optimization*. Surrogate Optimization techniques address this trade-off by making use of an inexpensive to evaluate surrogate function to approximate the expensive to evaluate objective function y, essentially simulating exploration of the expensive to evaluate objective function. In addition to using a surrogate function, BO also makes use of an *Acquisition function*, which is a function built using the surrogate function, used to efficiently guide the exploration process which increases the speed of convergence [5]. A Gaussian Process is typically used as the surrogate function for BO.

Bayesian Optimisation has gained popularity in the deep learning community as a hyperparameter tuning technique because of the high cost - in time and computing power - associated with tuning hyperparameters.

3.1 Bayesian Optimisation prior

There are many acquisition functions to choose from. This work focuses on the Expected Improvement acquisition function which, as the name suggests, is an acquisition function that proposes locations for exploration with the highest expected improvement, further discussed in 3.1.1. Before the acquisition function is used, a covariance function is chosen and a GP prior is specified for y denoted

$$p(y) = \mathcal{GP}(y; \mu, K)$$

Then given a set of two observations $\mathcal{D} = (X, y(X))$, the prior is conditioned on $\mathcal{D}$ to obtain the posterior distribution

$$f(x) \sim p(y \mid \mathcal{D}) = \mathcal{GP}\left(y; \mu_{y|\mathcal{D}}, K_{y|\mathcal{D}}\right)$$

Initially, $\mathcal{D}$ contains two elements, that is $|\mathcal{D}| = 2$, and the first two observations can be chosen at random in the domain or can be chosen using equal spacing in the domain $\mathbf{X}$. For example, using equal spacing, if $\mathbf{X} = [0, 1]$ then the first two observation locations $X = \{a, b\} \subseteq \mathbf{X}$ would be $a = \frac{1}{3}$ and $b = \frac{2}{3}$. The Expected Improvement algorithm, outlined in the next subsection, is used to guide all subsequent sampling.

3.1.1 BO Expected Improvement

The *Expected Improvement* (EI) acquisition function seeks to calculate $\forall x \in \mathbf{X}$ how much its function value $y(x)$ can be expected to improve over our current optimum and then uses the maximum expected improvement to iteratively guide exploration [5]. On a continuous domain using the objective function itself to calculate the EI would be intractable because it would require infinite function evaluations of the expensive to evaluate objective function that we are trying to optimise. BO instead makes use of the surrogate function - in our case the GP that has been conditioned on $\mathcal{D}$ - to calculate the aforementioned expected improvement [10]. Following [10], $\forall x \in \mathbf{X}$ the improvement function $I(x)$ is defined as:

$$I(x) = f(x) - f\left(x^+\right)$$

Here x^+ it is the x location of the highest posterior mean found so far. The likelihood of improvement $I(x)$ is then:

$$p(I) = \frac{1}{\sqrt{2\pi}\sigma(x)} \exp\left(-\frac{1}{2}\frac{\left[\mu(x) - f\left(x^+\right) - I\right]^2}{\sigma^2(x)}\right)$$

Here $\mu(x)$ and $\sigma(x)$ are the mean and the standard deviation of the GP posterior predictive at x, respectively. The EI acquisition function is then defined as $\alpha^{\mathrm{EI}}(x) = \mathbb{E}\left[I(x)\right]$, the expectation over the improvement function, and can be derived using the likelihood of improvement.

$$\alpha^{\mathrm{EI}}(x) = \int_0^\infty I \times \frac{1}{\sqrt{2\pi}\sigma(x)} \exp\left(-\frac{1}{2}\frac{\left[\mu(x) - f\left(x^+\right) - I\right]^2}{\sigma^2(x)}\right) dI$$

This would be a difficult integral to solve using numerical integration, but luckily (or by design) it has a closed form analytical solution outlined in [10]. Let $z(x) = \frac{\mu(x) - f\left(x^+\right)}{\sigma(x)}$ and Φ and ϕ be the c.d.f. and the p.d.f. of the standard normal distribution, then the closed form solution is:

$$\alpha^{\mathrm{EI}}(x) = \mathbb{E}\left[I(x)\right] = \sigma(x)\phi(z) + \left[\mu(x) - f\left(x^+\right)\right]\Phi(z)$$

3.1.2 Original BO Algorithm

Using Expected Improvement with noise-free observations, the pseudocode for Bayesian Optimisation is shown in Algorithm 1. The primary work on this algorithm is from [10].

Algorithm 1 Maximise y

$GP \leftarrow$ GP prior
$i \leftarrow 1$
$\mathcal{D} \leftarrow \varnothing$
while $|\mathcal{D}| < 2$ and $i \leq 2$ **do**
    Select a point x_i at random
    Compute $y(x_i)$
    Add element $(x_i, y(x_i))$ to $\mathcal{D}$
    $i \leftarrow i + 1$
end while
$i \leftarrow 1$
Condition GP on $\mathcal{D}$
Compute $\alpha_2^{\mathrm{EI}}(x)$
while $\alpha_i^{\mathrm{EI}}(x) > 0 \; \forall x \in \mathbf{X}$ **do**
    Compute $\alpha_i^{\mathrm{EI}}(x)$
    $x_\alpha \leftarrow \mathrm{argmax}_{x \in \mathbf{X}} \, \alpha_i^{\mathrm{EI}}(x)$
    Compute $y(x_\alpha)$
    Add element $(x_\alpha, y(x_\alpha))$ to $\mathcal{D}$
    Condition GP on $\mathcal{D}$
    $i \leftarrow i + 1$
end while

3.1.3 Stopping Conditions

The goal is to find the optimal value of the objective function as quickly as possible. A stopping condition is critical in order to prevent unnecessary exploration. In Algorithm 1, the algorithm converges when the expected improvement is zero or less [11]. That is, there is no more improvement to be gained from exploring. If objective function evaluations are noise-free then it is inevitable that the expected improvement becomes zero or less [11]. However in scenarios where noise-free objective function evaluations are not possible the expected improvement is always greater than zero since there is always some uncertainty to be reduced and the acquisition function typically suggests exploration at the estimated global maximum multiple times. In practice, it is useful to use an alternative stopping condition.

There are many different stopping conditions for BO and it is a field of great study. Below is a list of a few stopping conditions [10, 11].

- Max Iteration. Stop when some maximum number of iterations is reached.
- Max Time Limit. Stop when some maximum time limit is reached.
- Hybrid stopping criterion. Expected improvement is used to iteratively propose the next sample location. Stop when the probability of improvement (PI) is less than some pre-specified probability.
- Tolerance. Stop when the distance between two consecutive observations is less than some tolerance.

3.1.4 BO Algorithm with Stopping Conditions

Algorithm 2 below is nearly identical to Algorithm 1, the purpose of including it is simply to show how one would use the chosen stopping condition to halt BO by breaking the second while loop.

Algorithm 2 Maximise y using BO

$GP \leftarrow$ GP prior
$i \leftarrow 1$
$\mathcal{D} \leftarrow \varnothing$
while $|\mathcal{D}| < 2$ and $i \leq 2$ **do**
    Select a point x_i at random
    Compute $y(x_i)$
    Add element $(x_i, y(x_i))$ to $\mathcal{D}$
    $i \leftarrow i + 1$
end while
$i \leftarrow 1$
Condition GP on $\mathcal{D}$
Compute $\alpha_2^{\mathrm{EI}}(x)$
while not $STOPPING\_CONDITION$ **do**
    Compute $\alpha_i^{\mathrm{EI}}(x)$
    $x_\alpha \leftarrow \mathrm{argmax}_{x \in \mathbf{X}} \alpha_i^{\mathrm{EI}}(x)$
    Compute $y(x_\alpha)$
    Add element $(x_\alpha, y(x_\alpha))$ to $\mathcal{D}$
    Condition GP on $\mathcal{D}$
    $i \leftarrow i + 1$
end while

3.2 BO Example

Consider the objective function y from the previous chapter with noise $\epsilon \sim N\left(0, 0.2^2\right)$. Figures 3.2 and 3.1 illustrate ten iterations of GPR performed on y. The left column shows the GP fit to the observed data points at iteration i. In the right column, the standard deviation of the GP at iteration i is plotted and sampling is guided by choosing the x location associated with a new highest standard deviation.

In figure 3.2, using GPR, a location near to the global maximum is recommended for exploration by the ninth iteration and it is not clear that exploration should be stopped. On the tenth iteration, exploration continues and a sub-optimal location (roughly $x = 0.7$) is proposed as the next location to explore but the maximum number of ten iterations is reached, and GPR is halted.

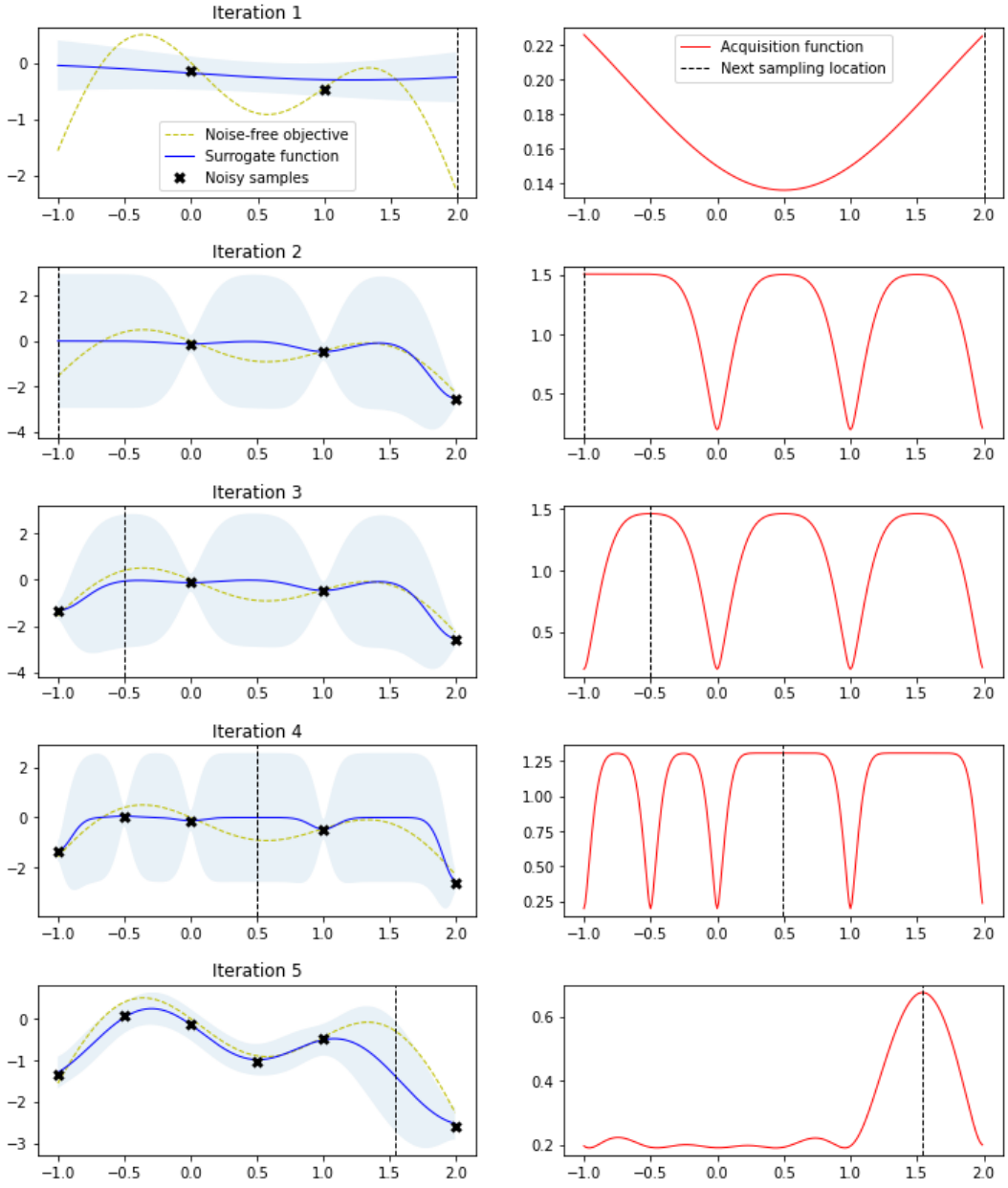


Figure 3.1: Ten iterations of Gaussian Process fitting using GPR, using the standard deviation of the GP as the acquisition function, are shown. No exploitation is occurring, only exploration, as knowledge gained at each iteration, is not exploited to guide exploration near the global maximum.

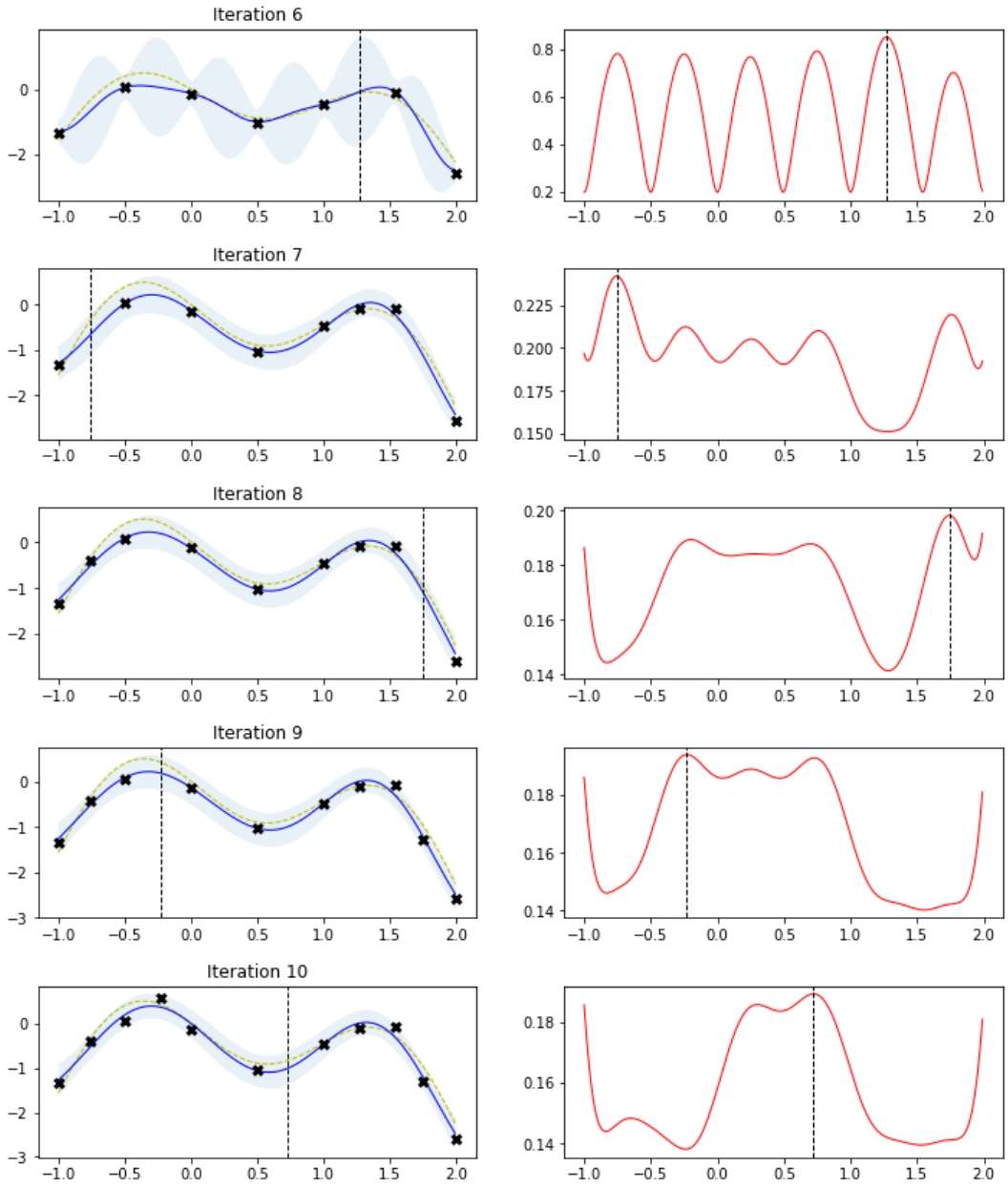


Figure 3.2: The last iterations, six through ten, of Gaussian Process fitting using GPR

The following figures illustrate ten iterations of BO performed on the same objective function. BO makes use of GPR, but instead of sampling in order to reduce uncertainty, the acquisition function is used to guide sampling in order to find the global maximum in as few objective function evaluations as possible. As previously mentioned EI is used as the acquisition function.

Using BO a location near to the global maximum is recommended for exploration by the fourth iteration and all subsequent exploration takes place near the global maximum until the maximum number of iterations, ten, is reached. The example in Figure 3.3 illustrates why the use of tolerance as a stopping condition is sometimes useful. Using a suitable choice for tolerance BO would have halted after five iterations.

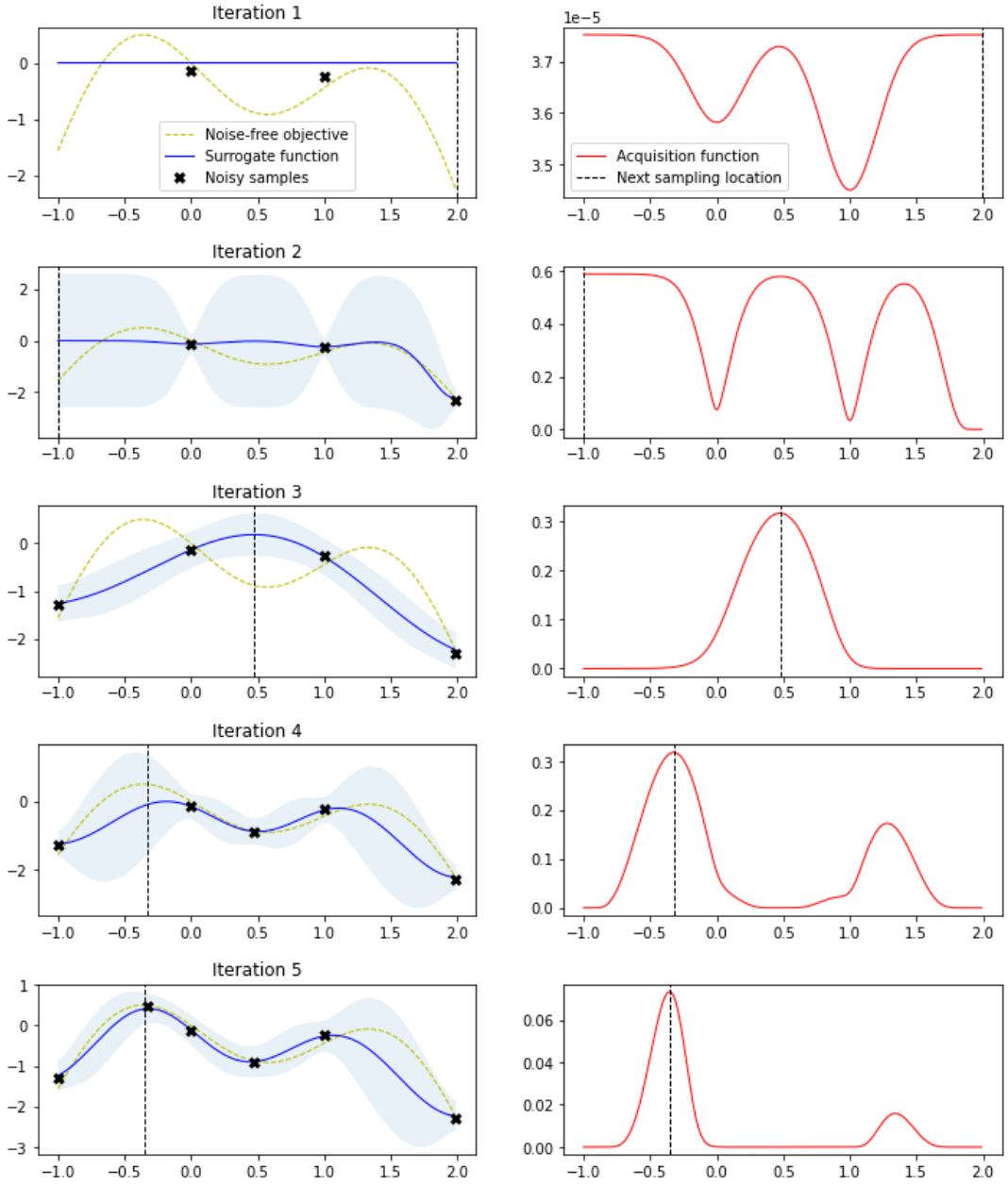


Figure 3.3: Bayesian Optimisation. Ten iterations of Gaussian Process fitting, using the Expected Improvement acquisition function are shown. After the first few iterations, sampling converges to within a small neighborhood around the global maximum

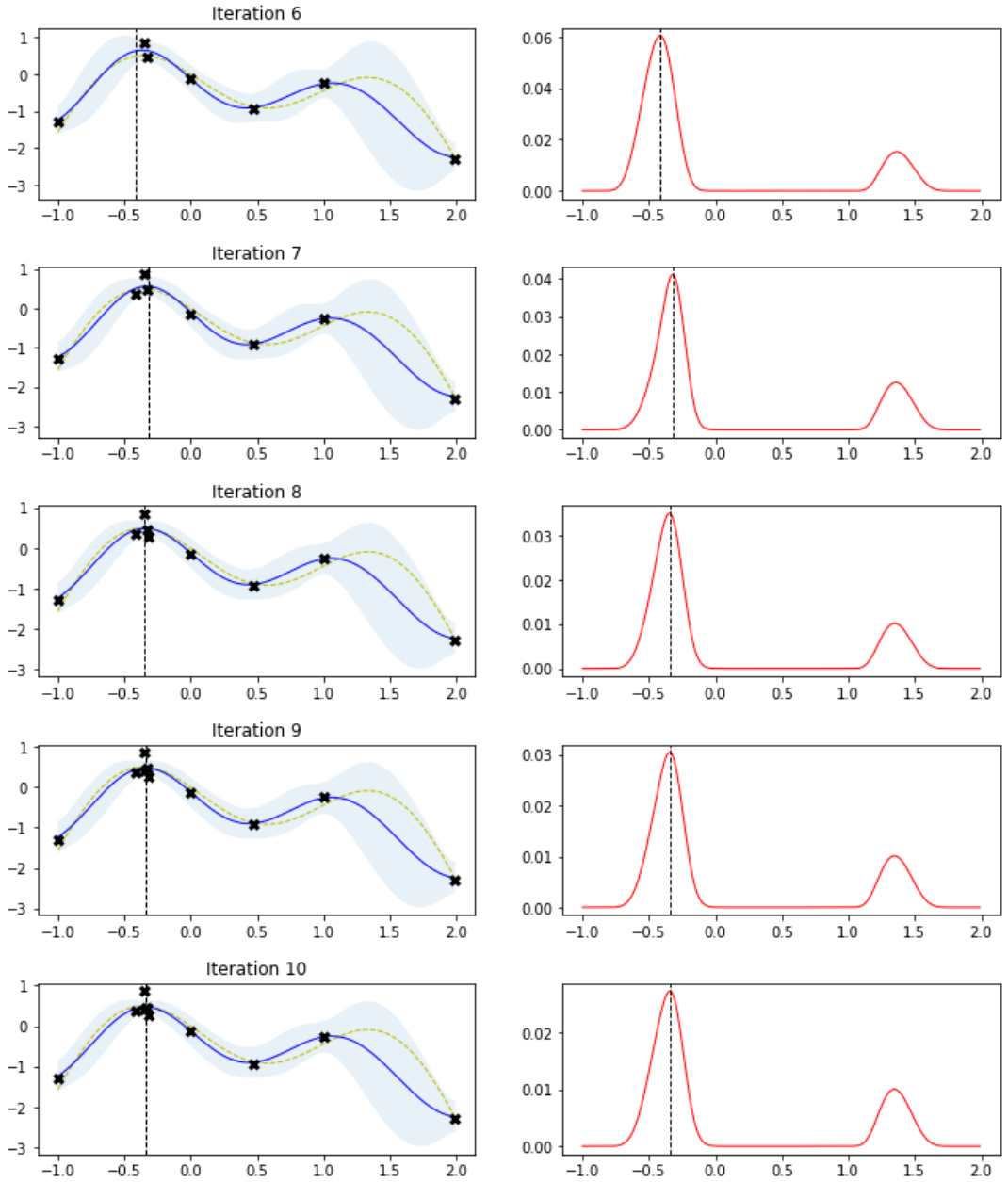


Figure 3.4: The last five iterations of Gaussian Process fitting, using the Expected Improvement acquisition function

Chapter 4

NeuroEvolution of Augmenting Topologies

NeuroEvolution algorithms are a class of *Genetic Algorithm*[12], which are algorithms that mimic evolutionary biology in order to search for global optima. They work by starting with an initial population of candidate solutions that are tested and ranked against each other. Simulated natural selection then proposes candidate solutions to be mutated and combined, simulating the biological process of breeding and passing down slight variants of genomes (detailed in this section) [13]. Strictly speaking, GAs are a class of optimisation techniques typically used to solve non-linear or non-differentiable optimisation problems. Non-differentiable optimisation problems are optimisation problems where access to the gradient of the cost function is not available.

This work makes use of NeuroEvolution to solve RL tasks, and BO is used to find optimal hyperparemter values while solving the reinforcement learning tasks. NeuroEvolution was chosen because it has many (more than 100) hyperparameters, making it difficult to search the entire hyperparameter space.

There are many different ways of implementing GAs, and one such way is NeuroEvolution [12]. In order to lay out the foundation of NeuroEvolution of Augmenting Topologies, NeuroEvolution is introduced in the NeuroEvolution section of this chapter and a brief overview of *Artificial Neural Networks* (ANN) is outlined in the next section.

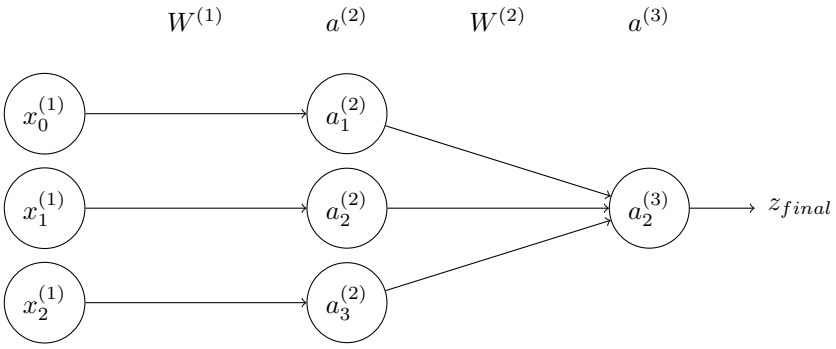


Figure 4.1: An example of the flow of information through a Feed-Forward Neural Network with three inputs, one output and a hidden layer with three hidden neurons

4.1 Feed Forward Neural Networks

ANNs are computational models loosely based on our limited understanding of the human brain. ANNs consist of ANN nodes, inspired by neurons, the brain's computational nodes, and ANN connections, inspired by synapses, the interconnections between the brain's neurons [14]. Information is manipulated by a series of different operations as it flows through an ANN.

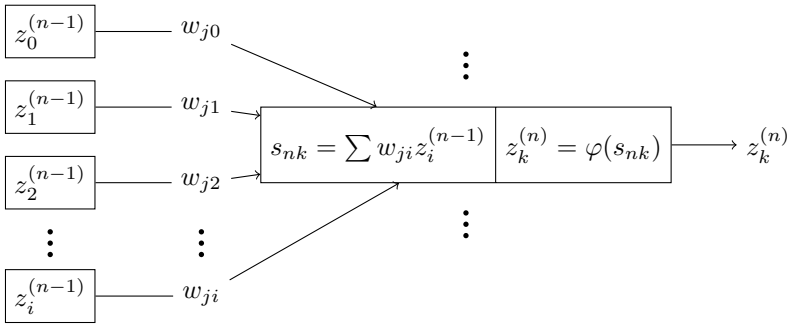
Figure 4.1 is a diagram showing signal flowing through an example ANN. This is one of the infinitely many examples of an ANN topology or layout. This particular topology is an example of a *Feed Forward Neural Network*, a type of ANN where there are no cycles between nodes meaning that information flows in one direction. In this example, information flows from:

- An input layer on the left, with three inputs, to
- A hidden layer in the middle, with three hidden neurons each with three inputs and one output, to
- The output layer on the right, with a single output node.

Two vectors of weights hold the values of the weights on the connections between layers of neurons. The vectors of weights are denoted W_1 and W_2 in the example and in general w_{ji} is the i^{th} weight of the j^{th} weight vector:

$$W_j = \begin{bmatrix} w_{j1} \\ w_{j2} \\ \vdots \\ w_{jm} \end{bmatrix} \tag{4.1}$$

In the above diagram, the superscripts represent the layer that a node is in. $x^{(1)}$ is the input layer and all subsequent layers are denoted $a^{(n)}$ for some $n \in \{2, \ldots, N\}$, where N is the number of layers. To illustrate the signal flow at each individual node consider node $a_k^{(n)}$, the k^{th} node in the n^{th} layer $a^{(n)}$. The signal flow at this node is:



The output $z_k^{(n)}$ from node $a_k^{(n)}$ is a function φ of s_{nk} the sum of the product of the outputs from the previous layer $a^{(n-1)}$ and the weights W_j, between the two layers. $\varphi(s_{nk})$ is called the activation function, and it is used to calculate the output to the next neurons in the ANN [14]. s_{nk} is linear for all n and k. There are many different types of activation functions, and they can be linear or nonlinear, but they are usually nonlinear and used to introduce nonlinearity into ANNs. Introducing this nonlinearity allows ANNs to fit nonlinear functions.

An alternative to a Feed-Forward Neural Network is a *Recurrent Neural Network* (RNN). RNNs differ from Feed Forward Neural Network only in that cycles are permitted in RNNs, meaning that information can be passed on from step to step in a sequence of observations. This is useful when observations are not independent of the sequence of previous observations. For example, the sequence in which words appear in a sentence is important. The sentence “The food is not bad here" has a completely different meaning to the sentence “The bad food is not here", even though they both contain the same words. The meaning of the sentence is dictated by the order of the words.

4.1.1 Approximating Functions Using Neural Networks

The goal of making such a network is to approximates the process that generates some objective function y by creating a mapping (mapping inputs to outputs) from observed data sampled from the objective function that we wish to fit. In the example above, the final output was z_{final}. This output is used to calculate how well the ANN maps inputs to outputs when compared to the data from actual objective function y. There are many different ways of quantifying this similarity and it is usually referred to as the error. There are many different error functions and they are usually a function of the difference between the output from the ANN and the actual output from y given the same inputs. The lower the error the closer the output from the ANN is to the output from y given the same input.

4.1.2 Issues with Approximating Functions Using Neural Networks

The overall goal is to use the ANN to create an approximation of the process that generates y. To this end, a lower error does not necessarily mean a better approximation because data drawn from an objective function can never fully represent the objective function itself and, for a finite number of data points (observation locations and the corresponding outputs from y), a trained ANN is only one of the infinitely many possible mappings that pass through the points. This is not to mention that objective function evaluations could be noisy, which would also affect the fit of the neural network. Collectively these two problems are known as over-fitting. A network that has overfit a data set is a network that has a low error but is not necessarily a network that accurately approximates the process that generates y. One way of preventing over-fitting is a method called Holdout [14]. Using Holdout the data is split into test and training tests. The test data is used to calculate the final estimate of the ANNs performance, the error after it has been trained on the training data. Over-fitting is discussed further in a later chapter.

4.1.3 The Universal Approximation Theorem

An ANN with 1 hidden layer can theoretically approximate any continuous function for inputs within a specific range [15]. This is known as the Universal Approximation Theorem. There are multiple papers [15, 16, 17, 18] proving the theorem using different assumptions, one of the earliest being [15]. These papers only prove the existence of solutions and they do not propose methods for training ANNs to find these solutions. In practice, multiple hidden layers are used because training shallow networks (Networks with few hidden layers) is computationally

intractable for some functions that we may wish to approximate using current training methods.

This is similar to the theoretical flexibility that Gaussian Processes have to fit any objective function while only being characterised by their covariance functions and not their mean functions. Like the Universal Approximation Theorem, this is theoretically correct but could also lead to computationally intractable situations were incorporating some domain knowledge into the specification of the mean function or normalising the output that we wish to approximate can reduce the number of training steps needed to perform Gaussian Process Regression. For ANNs, in order to find suitable solutions a learning rule is needed to update weights and biases with the goal of reducing the error between the ANNs outputs and the actual observed outputs from the objective function that we wish to fit. The Universal Approximation Theorem simply addresses the ability of an ANN to fit a data set and doesn't address over-fitting. As previously mentioned, a perfect fit to the data does not necessarily mean that the ANN is an accurate approximation of the objective function everywhere, only on the observed data from the objective function.

4.1.4 Training Algorithms

A training algorithm, sometimes referred to as a learning algorithm, is an algorithm that updates the weights and biases of ANNs in order to reduce the difference between the ANNs outputs and the objective function's outputs known as error. This process is referred to as the training process. One of the most popular training algorithms is the backpropagation algorithm [19]. Ba at a high level can be thought of as a mathematical framework for efficiently calculating the partial effects that infinitely small changes in each of the weights and biases have on the total error of an ANN. These partial effects are formally known as the partial derivatives of the weights and biases with respect to the error function. In order to obtain these partial derivatives, backpropagation makes use of the chain rule, a mathematical formula for calculating the derivatives of variables that can be defined in terms of other variables. Let l be the learning rate, a hyperparameter that controls the size of the updates to the weights, and let $\frac{\partial E}{\partial W}$ be the partial derivatives of error function w.r.t. parameters. Backpropagation then uses Gradient Descent, a method for adjusting the weights in the direction that most reduces the error by subtracting the partial derivatives from the current weights to obtain a new set of weights as shown below.

$$\text{New Weights} = \text{Old Weights} - l\frac{\partial E}{\partial W}$$

This process is repeated multiple times until some convergence criteria are met. Once training is complete, using the Holdout technique, unseen test data is used to evaluate the fit of the neural network. There are many different variations of Backpropagation and different Gradient Decent weight update rules [14]. Although backpropagation has been responsible for most of the advances in modern machine learning, it only addresses the values of the weights and biases, not the structure or architecture of the ANN itself. For example, it may be useful to have a training algorithm that optimises the weights and the number of hidden layers, or how many connections there should be between nodes. One such algorithm is NeuroEvolution of Augmenting Topologies [13].

4.2 NeuroEvolution

NeuroEvolution algorithms are a class of algorithms that find appropriate weights for given ANN topologies using genetic algorithms [13]. Where typically a training algorithm would use calculus to update the weights of a specific ANN for a given problem, NeuroEvolution instead starts with an initial populating of many randomly initialised ANNs. Each of the ANNs in this population of ANNs is a candidate solution to the problem. A candidate solution is a member of a set of possible solutions to a given problem. The candidate solutions are ranked by their fitness, a function of the error associated with each ANN. There is a lot of freedom when defining the fitness of a candidate solution and fitness functions are defined on a case by case basis to suit the given problem. The fittest members of the population are “breed", and the children of the fittest members replace some of the candidate solutions in the population with the lowest fitness, simulating natural selection [13]. A mutation is performed by randomly perturbing the weights of the fittest ANNs, and these mutated ANNs also replace some of the least fit candidate solutions. The goal is to create a new population of ANNs that better solve the given problem.

4.3 When NeuroEvolution is Useful

For the purpose of illustrating a scenario in which genetic algorithms are more suitable than gradient-based training algorithms for optimisation, consider a simple ANN that only has two weights, w_1 and w_2, and an objective function $y(X)$ that we wish to approximate using the ANN. Given a dataset X, the total error of the ANN is a function of the weights w_1 and w_2. Figure 4.2 is a plot of the error with all possible weight combinations in a fixed domain $w_1, w_2 \in [0, 0.8]$.

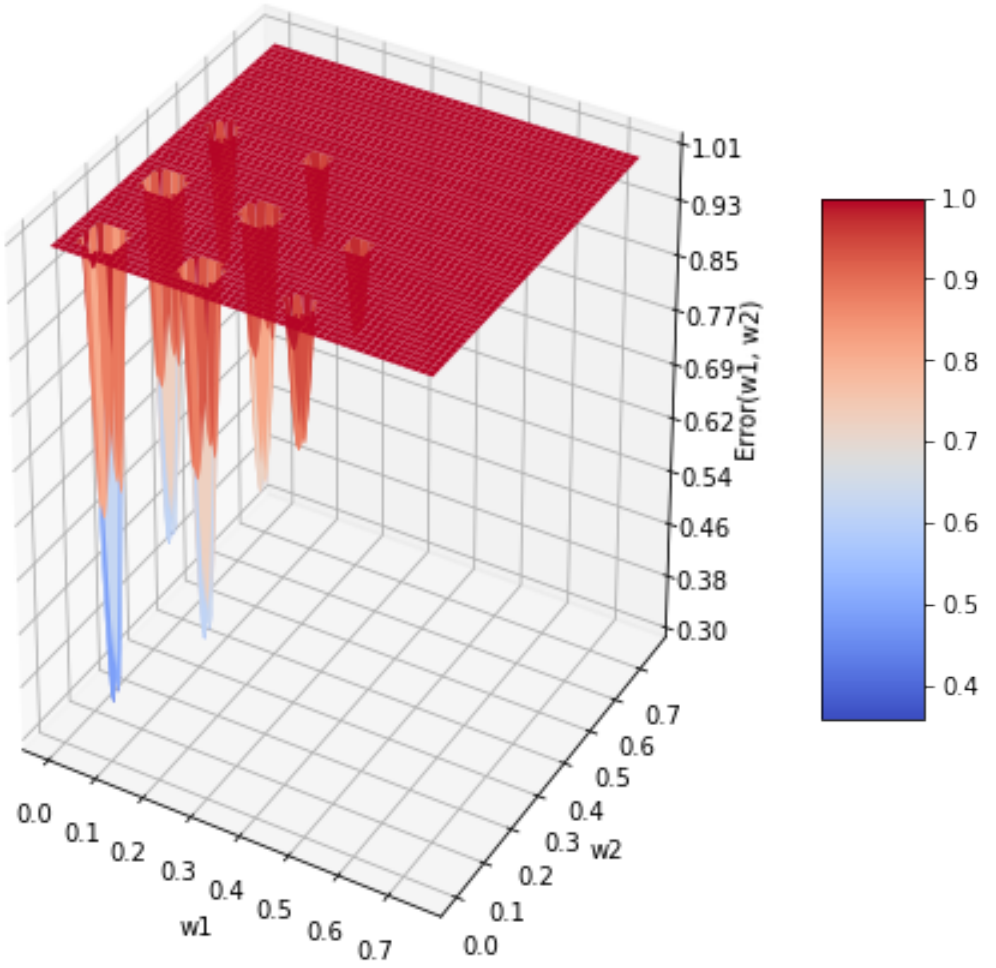


Figure 4.2: An example of a “Swiss Cheese Surface" where there are multiple local minima, and the majority of the surface has a gradient of zero.

This type of surface is known as an error Surface, and this is a particularly difficult error Surface to minimise for two reasons:

- The Partial error Gradients. Most of the weight combinations yield the same error, 1, meaning that the partial gradients of the error with respect to each of the weights are equal to zero in most of the search space ($\frac{\partial E}{\partial w_1} = 0$ and $\frac{\partial E}{\partial w_2} = 0$). This means that for gradient-based training algorithms the error gradients provide no useful information for

updating the weights in most of the weight search space ($w_1, w_2 \in [0, 0.8]$).

- Local and Global Minima. The Global Minima is surrounded by multiple Local Minima. In this scenario, more exploration of the weight search space is needed to find the Global Minima.

Genetic algorithms do not rely on error gradients to perform weight updates and are therefore better suited to situations that are prone to the partial error gradients issue outlined in the first point above. Although certain gradient-based (ADAM, and other Stochastic Gradient Descent-based training algorithms [20]) training algorithms seek to address the second issue, genetic algorithms are better for exploring more of the solution space because updates are cheaper to compute without calculating gradients, meaning that more of the search space can be explored for the same computational cost. Although both genetic algorithms and Stochastic Gradient Descent-based training algorithms include randomness for exploration, neither of them guarantees to reach of the global minima [20].

The error Surface plotted above can occur in reinforcement learning tasks (reinforcement learning is introduced in the reinforcement learning chapter), where the majority of weight combinations lead to incorrect solutions and only a small handful of weight combinations lead to near-optimal or optimal solutions

4.4 NeuroEvolution of Augmenting Topologies Introduction

NeuroEvolution of Augmenting Topologies (NEAT) is a genetic algorithm that simultaneously evolves the topology and the weights and biases of an ANN. NEAT begins with a randomly initialised *population* of simple ANNs that are candidate solutions to a given problem. Each candidate solution ANN in the population is referred to as a *Genome* [13]. The genomes are iteratively complexified by adding new neurons and connections over generations, while simultaneously updating the weights using crossover and mutation. Starting with simple network topology and iteratively increasing the complexity in this way increases the chance of finding a minimal ANN topology that solves a given problem. In paper [13], Kenneth O. Stanley and Risto Miikkulainen first presented NEAT which, like most genetic algorithms, relies on three key evolutionary operations to evolve solutions:

- Selection. In order to simulate natural selection, or survival of the fittest, a fitness criterion needs to be specified. This fitness criterion is then used to rank the genomes (candidate solutions in the population) and the fittest ones are selected for crossover.
- Crossover. To emulate breeding, connections and topologies of selected genomes are combined with each other to form new genomes.
- Mutation. Mutation is used to mutate existing weights, by randomly perturbing them, and mutation is used to add new structure, connections, and nodes, to existing network topologies.

Part of NEAT's unique objective, to evolve ANN topologies, poses a unique problem. Using ANNs, there are many different ways of representing a solution to a given optimization problem, and two ANNs may have similar output or similar error scores, given the same dataset, but have completely different topologies and weights, making conventional crossover impossible. This problem is referred to as the *Competing Conventions Problem* [21]. NEAT addresses this problem using a novel encoding that keeps track of genomes throughout the evolution process. genomes are then sorted into species using this encoding, and crossover is performed only between genomes from the same species. Species are introduced and discussed in the "Speciation" section.

4.5 NeuroEvolution of Augmenting Topologies Algorithm

At a high level, the NEAT algorithm consists of three steps, an initialisation step and two steps that are repeated in a loop until some convergence criteria are met. The steps are:

- Step 1. Create a population of ANNs.
- Step 2. Evaluate the fitness of each of the ANNs in the population.
- Step 3. Create a new population using:
  - Crossover. Select *parents* based on their fitness and combine a random ratio of "DNA" from each of the parents to form a new *child* candidate solution.
  - Mutation. Randomly perturb the "DNA" of a few of the parent ANNs.

Step 1 is the initialisation step and steps 2 and 3 are repeated until some stopping criteria have been met [13].

Selection, Mutation, and Crossover are concepts from genetic algorithms and are not unique to NEAT. Another key concept from genetic algorithms used for problem formulation is the idea of Genotype and Phenotype.

4.5.1 Genome vs. Phenome

The *Genome* is the so-called digital "DNA" of a candidate solution and can be thought of as the recipe for making a specific ANN, a *Phenome*. If the Genome is a list of ingredients and instructions then the *Phenotype* is the final product, the ANN made using the Genome as its recipe [13]. genomes are separated into two sets of genes, the Node Genes which can be thought of as the ingredients, and the Connection Genes which can be thought of as the instructions for connecting the Node Genes. Figure 4.3 is from chapter 3 of "NeuroEvolution of Augmenting Topologies (NEAT)" in the paper "Evolving Neural Networks through Augmenting Topologies" by Kenneth O. Stanley and Risto Miikkulainen [12]. In the figure, a Genome and the resulting ANN are shown.

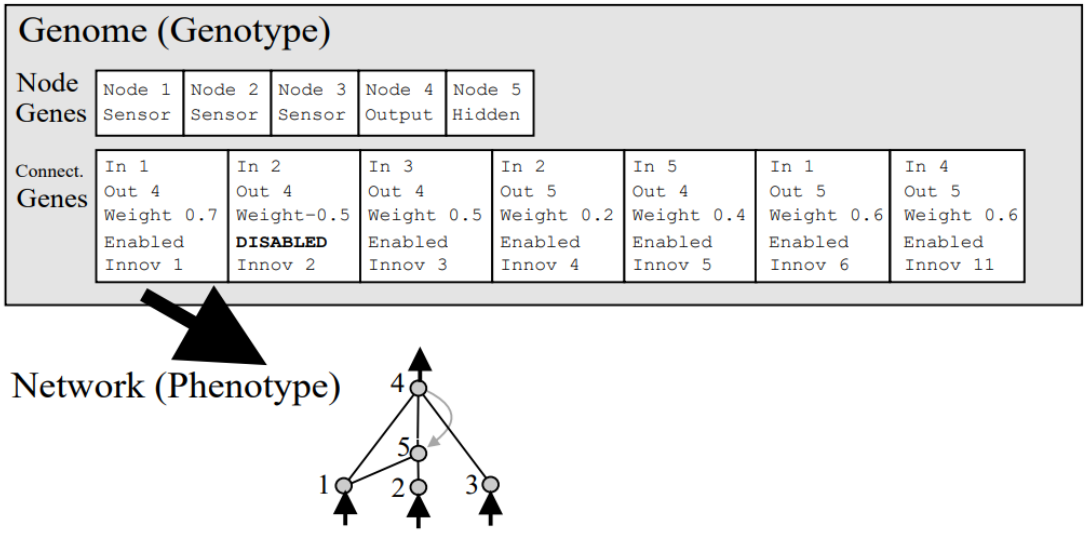


Figure 4.3: An example of a NEAT Genome, and its corresponding NEAT Phenotype, illustrating how a NEAT Genome serves as a NEAT Phenotype blueprint. Refer to [12] for more detail.

Each node gene specifies a node and its function. The three node functions are Input, Output, and Hidden. In the figure above and in the paper [12], Input nodes are referred to as

“Sensor" nodes. Each Connection Gene specifies a single connection between nodes, its direction, the weight associated with it, and an Innovation Number [13]. Innovation Numbers are discussed in the Speciation section.

4.5.2 Mutation

Mutation is the process of altering the Genes in the Genome. During mutation, there are five main network topology alterations that can be performed at random:

- A new connection can be added between existing nodes in the Genome.
- A connection can be deleted between existing nodes in the Genome.
- A new node can be added to the Genome.
- A node can be deleted from the Genome.
- The enabled status of a connection can be replaced (E.g. if the enabled status of a given connection was set to true, then it would be replaced with false).

Each of the above five alteration are associated with a hyperparameter which is the probability that that alteration happens. Whenever mutation occurs, all five alterations can occur with some probability to each node or connection. Values for these hyperparameters are usually manually defined by the user [13]. This dissertation explores the automatic tuning of these hyperparameters. The respective hyperparameters for the five points above:

- *prob_add_connection*
- *prob_delete_connection*
- *prob_add_node*
- *prob_delete_node*
- *enabled_mutate_rate*

Again, as stated in the introduction, the efficacy of NEAT is highly dependant on these hyperparameters and a delicate balance between the five probabilities needs to be found. In the figure above, from the same paper from [12], the process of adding a connection and adding

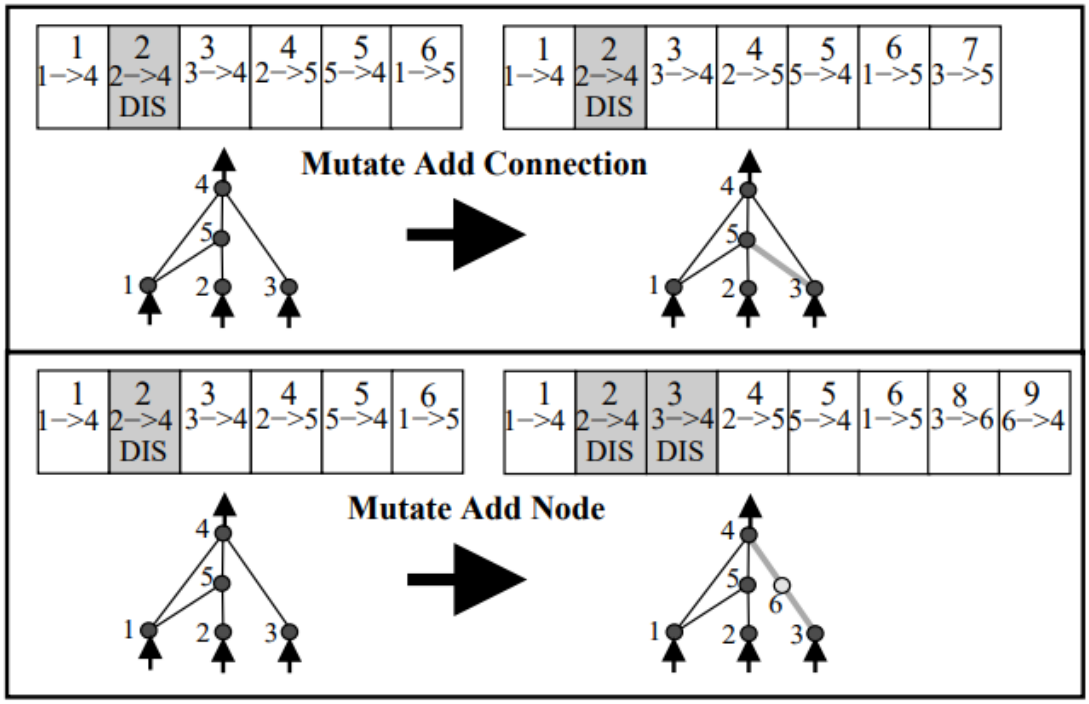


Figure 4.4: The add connection and add node NEAT mutations are illustrated, showing how the topology of an ANN grows, either by adding a connection or a node from [12].

a node is illustrated. In each of the rectangles in the genomes above, there are two numbers and an arrow between them which represent the nodes being connected and the direction of the connection. Above the two numbers connected by an arrow is another number. This is called the Innovation Number, and is discussed in detail in the “Speciation" section below. Below the two numbers connected by an arrow is either nothing or the letters “DIS". Nodes are added between nodes that have an existing connection, effectively breaking a connection into two connections with a node in between them. In the figure, “DIS" appears below connections that have been broken into two. This is to show that there was a connection, but it is currently disconnected. In the mutation step, the weights of an ANN can also be mutated by adding a small random value to each weight. Specifically: “There was an 80% chance of a genome having its connection weights mutated, in which case each weight had a 90% chance of being uniformly perturbed and a 10% chance of being assigned a new random value" [12].

4.5.3 Speciation

Whenever a new connection is created, it is assigned a unique number. As mentioned in the "Genotype vs. Phenotype" section above this number is called the Innovation Number, and it is used to keep track of the origin of connections in subsequent child genomes. genomes are separated into species according to their similarity, which is a function of each Genome's Innovation Numbers and Weights. In the literature, [12] the similarity is referred to as the compatibility distance, denoted δ. In order to calculate δ, we need to introduce some values that δ depends on. Let E be the number of genes from one parent that appear outside the range of the other parent's Innovation Numbers, and D be the number of disjoint genes between two genomes. D is the number of genes that appear in one parent genome, but not in the other and are within the range of the other's Innovation Numbers [13]. Let $\bar{W}$ be the means of the differences between matching genes in both genomes. Then the compatibility distance is [12]:

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

To illustrate how this is calculated in practice, E, D can be calculated for the two genomes "Parent1" and "Parent2" in the "Crossover" section of Figure 4.5. E, the number of excess genes, is three, the gene with Innovation Numbers $6, 7, 8$. D, the number of disjoint genes. $D = 2$ in the same figure mentioned above, with innovation Numbers 9,10. The hyperparameter $compatibility\_threshold$ is an upper bound of the similarity, or compatibility distance δ, is used to create species within a population of ANNs.

4.5.4 Crossover

Crossover is the process of creating new *Child genomes* from two existing *Parent genomes*. Genes from the Parent genomes can either be disjoint, excess or included in both genomes. Here excess Genes nonmatching Genes at the end of the genomes. Genes with the same Innovation Number do not necessarily have the same Weights assigned to them because of the weight mutation process in previous generations. From genes that match in both parents, the Child Genome randomly inherits a mixture of the weights from the matching genes. All other genes, the excess, and disjoint genes, are included from the fittest parent [12].

In order to avoid the Competing Conventions Problem, mentioned in section 4.4, Crossover can only be performed on genomes that have been deemed similar enough. The quantification of the similarity between genomes is beyond the scope of this work and covered

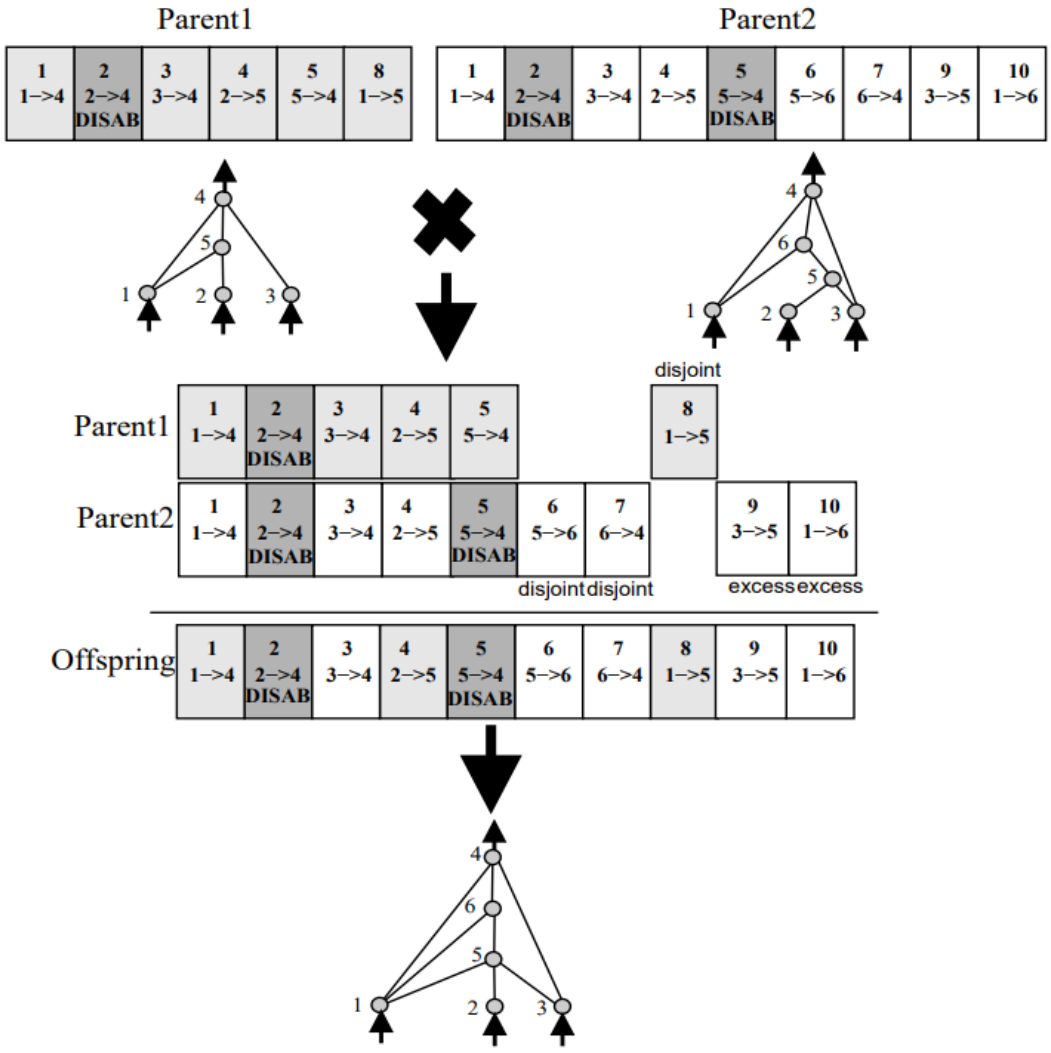


Figure 4.5: A figure from [12] illustrates the Crossover operation performed on two Parent genomes, Parent1 and Parent2, and the subsequent Child Genome, Offspring.

in detail in the paper [12].

4.5.5 NEAT in This Work

Using NEAT, ANN topology parameters that would usually be hyperparameters are included as model parameters to be optimised, because of NEAT's ability to find ANN weights and ANN topologies simultaneously. With most other ANN optimising algorithms, an ANN topology is chosen before optimisation starts and the ANN topology stays constant throughout the optimisation process. NEAT, then, is a way of reducing ANN hyperparameters, but in doing so introduces its own set of hyperparameters. This work focuses on automatically finding suitable values for NEAT's hyperparameters while solving reinforcement learning problems.

Again, the efficacy of NEAT is highly dependant on these hyperparameters and a delicate balance between the five probabilities needs to be found.

This is because together they control the rate with which the networks increase or decrease in complexity and are usually chosen by using conventional wisdom or trial and error. This dissertation aims to address the hyperparameter problem using Bayesian Optimisation as a means to find appropriate combinations of the different values of the above probabilities in order to minimise the number of generations that NEAT has to evaluate and evolve before finding a suitable solution.

In this work we study the simultaneous tuning of 4 NEAT hyperparameters, Although 4 is much smaller than the total of 54 hyperparameters, it suffices to explore the key idea that we want to address, namely: can BO help improve the NEAT algorithm.

Chapter 5

Reinforcement Learning

The question "How can computers learn to solve problems without being explicitly programmed?" [22] is often used as the question that characterises machine learning problems. Thinking of *Reinforcement Learning* (RL), in the same way, as a set of learning problems characterized by a question, the question is: How can intelligent behaviour be learned, by agents taking actions in dynamic environments, with the goal of maximizing some reward? RL is one of the three main categories within machine learning. These categories are:

- Supervised Learning, or function approximation problems, where given inputs x and outputs from an objective function $y(x)$, the task is to learn a function g, $y = g(x)$, with the goal of being able to use g to map new unseen x values to corresponding approximate y values.
- Unsupervised Learning, where only inputs x values are given and the goal is to find some $g(x)$ that gives you a compact description of the x's that you have seen, which can then be used to describe or categorise subsequent unseen x values.
- Reinforcement learning, discussed in the next section.

5.1 Reinforcement Learning Tasks

A *Reinforcement Learning* (RL) task is a task characterised by an *Agent* and the *Environment* that the agent is in, and their interaction [2]. Their interaction is shown in Figure 5.1. The

Agent is initially in some starting state in the environment and has a set of actions that it can perform. Once the agent has chosen and executed an action, the environment will return some reward signal, generated by a prespecified reward function which returns a real number, and also return the next state that the agent will be in [2].

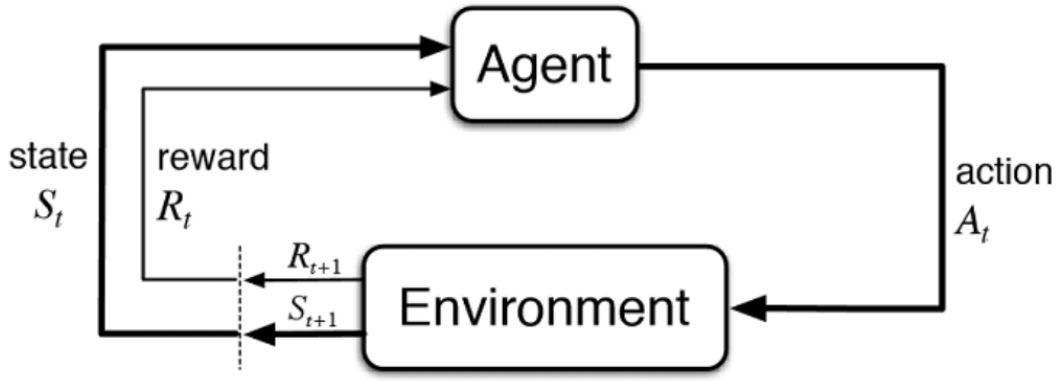


Figure 5.1: From the textbook "Introduction to Reinforcement Learning" [2], shows how reinforcement learning problems are posed. Let S_t and R_t be the current state and the current reward signal at step t of the process respectively. The agent observes S_t and selects action A_t in response to its observation. S_{t+1} and R_{t+1} are the next state and the next reward generated and returned by the environment, with some randomness, as a response to action A_t from state S_t. S_{t+1} and R_{t+1} are then observed by the agent at step $t+1$ and the cycle continues until some stopping criteria is reached.

For example, figuring out how to successfully land a spaceship on the Moon can be posed as an RL task. Imagine a simulation used to train an astronaut (The agent) who is given three buttons that control the three thrusters of a space ship while landing the space ship on the Moon (The spaceship and the Moon are the environment). Using the controls, the astronaut can either fire the left engine, right engine, center engine or do nothing (The set of available actions). At any time step, the astronaut can visually observe the ship's position with respect to the Moon (The current state), choose one of the buttons to press, or press none of them, and then again observe the space ships' orientation (The next state). Once the space ship makes contact with the Moons' surface it either crashes or lands without enough impact to break the ship. While the ship is approaching the Moon, there is no feedback as to how well the astronaut is landing the ship, until the ship has reached the Moon's surface, but all of the actions leading up to this moment have an impact on the final outcome (The reward) and should be judged by the final outcome. After enough retries, the astronaut will eventually learn how to land the ship without breaking it (Maximising the reward) [32].

5.2 Markov Decision Processes

A *Markov Decision Processes* (MDP) is a useful framework used to encapsulate most RL tasks [2]. MDP's are useful because of the Markov Property, which will be introduced in this section. MDPs consist of:

- A set of states $\mathcal{S}$, that represent every state that an agent can be in.
- A set of all possible actions $\mathcal{A}$ with $\mathcal{A}(S)$, the set of all actions available at state S.
- A Transition Function $T(S_t, A_t, S_{t+1})$ is the probability distribution $Pr(S_{t+1}|S_t, A_t)$ which is a model of the probability of transitioning between states given an action. Transition functions in some sense describe the rules of the environment.
- Reward Function $R(S_t, A_t, S_{t+1})$, covered in the Reward Function section below.
- Episode, a sequence that represents the states, actions, and rewards observed from the first initialisation state of an MDP to the last/terminal state.
- A Done boolean returned by the environment with each state in the MDP. A Boolean value indicating that an episode is finished (1 for finished and 0 for still in progress).

If for each state S_t the next state S_{t+1} is only dependant on state S_t and not by all of the states leading up to S_t (I.e. states $S_{t-1}, S_{t-2}, \dots$), then the process is said to have the *Markov Property*[1] sometimes called the memoryless property [2]. Formally an RL task that satisfies the Markov Property is called a Markov Decision Process. In order to satisfy this property, each state needs to be completely characterized by the state itself, which limits the types of RL tasks that can be posed as MDPs. There are however many tricks that can be used in order to incorporate information from previous states into the current state, for example when posing a physical system as an MDP using not only the position of elements in the system but also the element's position derivatives helps incorporate valuable information from previous states into the current state. Another example of how this problem can be overcome is presented in the "Neural Networks as reinforcement learning Function Approximators" section below.

[1]The Markov property can in some cases be limiting because it is not always possible to capture all of the dynamics of a process within a single snapshot (state) of the process. In these cases, where it is not clear how to ensure that the Markov Property is met, then it is useful to use a Recurrent Neural Network to keep track of past states. One could argue that, since an agent is part of the MDP and the ANN is part of the agent, using a Recurrent Neural Network is essentially encoding past states into the present state, effectively creating a means to represent the process as an MDP.

5.2.1 Reward Function

In order for agents to gauge how well they are performing at a certain task, some positive or negative feedback is needed from the environment. To this end, all MDPs have a reward function specified by the person creating the MDP [2]. Reward functions are defined on a case-by-case basis but they all must given a state, an action, and the subsequent state, return a numerical signal. RL algorithms then seek to find a policy that maximises the reward. For a given computer game, the reward at each time step could be the number of frames that the player has stayed alive irrespective of whether or not the game has been won or lost. For the same game, the reward function could be binary, returning one only if the game has been won and zero for all other steps. The second representation of reward is more desirable, even though it is more sparse.

It is important to carefully define the reward function because incorrectly specified reward functions can lead to undesirable agent behaviour. For example, if the end goal of a game is to successfully land a space ship, but the reward function simply returns one for every step of the game that the space ship remains intact, then an agent could learn that simply hovering above the ground would maximise the reward as this would maximise the length of the game. Carefully aligning our objectives with the objectives of Machine Learning algorithms is an area of ongoing debate.

5.3 Neural Networks for Reinforcement Learning

There are two different kinds of functions that agents rely on to guide them through an MDP. These functions characterize how agents select actions from state:

- Policy Functions map states to actions, describing the agents behaviour. The space of all policy functions is the space of all possible behaviors.
- Value Functions map state-action pairs to values that represent the utility of certain actions in certain states of the MDP. The values are then be used to guide agents' actions.

Typically, in order to solve RL tasks, ANNs are used to approximate either a value function, a policy function, or both. Policies and value functions are both defined mathematically.

5.4 NEAT Applied to Reinforcement Learning

Although NEAT can be used to evolve an ANN to approximate any objective function, it was created with the specific goal of solving RL tasks [12]. NEAT evolves a population of Policy Functions to search directly in the policy space with the end goal of finding multiple NNs that perform well at the given RL task. [12]

5.4.1 Advantages of NEAT

- As previously mentioned, genetic algorithm based methods work in the absence of gradients, which either might not exist or could be computationally expensive to calculate.
- NEAT can be run in parallel since having a population of multiple candidate NNs means that they can be simultaneously evaluated.
- Because of the inexpensive computational cost of updating NNs using NEAT and the parallel nature of NEAT, more of the solution space can be explored.
- The risk associated with the choice of the initial topology of the NNs is mitigated since NEAT evolves more favourable topologies which suit the task at hand.

5.4.2 Disadvantages of NEAT

- NEAT has a total of 54 hyperparameters, controlling a wide range of different aspects of the algorithm such as the characteristics of the NEAT population, the way in which members of the population reproduce, neural architecture starting points, activation functions and more [13]. The convergence of NEAT is highly dependant on the choice of its many (fifty-four) hyperparameter values, most of which are interdependent on each other. (This is the problem being directly addressed in this work.)
- If a highly-complex ANN topology, specifically a large one, is required to solve a given task, NEAT is inefficient since it evolves the ANNs from the initial most simple ANN Topology, a fully connected feed-forward ANN with no hidden layers. [23]

5.4.3 Literature Addressing the Disadvantages

The second disadvantage listed above is directly addressed in the paper “A hypercube based encoding for evolving large-scale neural networks” [23] written by the creator of NEAT, Kenneth O. Stanley et al. HyperNEAT, the algorithm proposed in the paper, uses an indirect encoding method to evolve large NNs. From the paper: “HyperNEAT” puts forward an indirect encoding called connective compositional pattern-producing networks (CPPNs) that can produce connectivity patterns with symmetries and repeating motifs by interpreting spatial patterns generated within a hypercube as connectivity patterns in a lower-dimensional space." [23]. In the paper: "A NeuroEvolution Approach to General Atari Game Playing" [25] it is stated that the: "Results indicate that direct-encoding methods (i.e. NEAT) work best on compact state representations while indirect-encoding methods (i.e. HyperNEAT) allow scaling to higher-dimensional representations (i.e. the raw game screen)."

Although HyperNEAT is a step in the right direction - extending NEAT allowing it to more efficiently create complex topologies - it is not clear that HyperNEAT can perform well on RL tasks with high-level features and geometrically unrelated features.

5.4.4 Addressing the Disadvantages of NEAT hyperparameter Optimisation

In this dissertation, the first disadvantage listed above is directly addressed using Bayesian Optimisation and random search to uncover potentially optimal hyperparameter value combinations. In addition to this, the second disadvantage is indirectly addressed. Selecting specific hyperparameter combinations can lead to complex ANN topologies being found more quickly than with other hyperparameter combinations.

For example, using the four hyperparameters mentioned in the NEAT Mutation chapter above, consider two different combinations:

hyperparameters	Combination 1	Combination 2
prob_add_connection	0.9	0.5
prob_delete_connection	0.2	0.5
prob_add_node	0.9	0.5
prob_delete_node	0.2	0.5

Table 5.1: A table comparing different NEAT hyperparameter value combinations

Using NEAT with Combination 1 for a given task will lead to quicker growth of the number of Nodes and Connections in the NNs, while Combination 2 would lead to a near-constant Network topology complexity for all NNs in the population of candidate solutions. Using a hyperparameter optimisation algorithm to find suitable hyperparameter combinations for a given task can in this way indirectly represent the complexity of the task at hand.

5.4.5 Exploration vs. Exploitation Trade-off

If too much time is spent exploring most of the time is spent in very bad regions. If you spend all the time exploiting then one gets stuck with your first guesses and never explore to find better solutions increasing the chance of getting stuck in local minima. To address the trade off between exploring and exploiting, NEAT makes use of a population of candidate solutions that simultaneously explore the solution space. Only solutions deemed the fittest can continue to evolve.

Typically, in the context of RL, to *Exploit* means for an agent to perform an action that is perceived as the optimal action based on past experience, and to *Explore* means that an agent performs a non-optimal action (non-optimal based solely on the agents' past experience) with the goal of observing the reward signal from the previously unobserved state-action pair [24]. Exploration and exploitation in the context of NEAT as a solution to RL tasks differ from the above descriptions in that individual agents are always acting exploitatively. The balance between exploitation and exploration is instead dictated at a population-wide level rather than at an agents' policy level. The balance is governed by the following hyperparameters:

- pop_size: The number of agents in each generation. Having more agents increases the number of non-optimal agents while increasing exploration using “genetic diversity".
- $max\_stagnation$: Is the maximum number of generations that a species can survive, before being removed from the population (deemed extinct), without improving its $species\_fitness\_func$. The $species\_fitness\_func$ in this work is the mean of all of the individual agents' fitnesses in the species.
- $species\_elitism$: Is the minimum number of species that are unconditionally preserved from extinction irrespective of how many generations the “elite" species stagnate for. A higher number results in non-optimal species of agents being preserved, resulting in less exploitation and more exploration. A lower value for $species\_elitism$ results in less genetic diversity in turn resulting in more exploitation than exploration.

Four of the five hyperparameters introduced in the "Mutation" section above, namely *prob_add_connection*, *prob_delete_connection*, *prob_add_node* and prob_delete_node also influence the explore vs exploit trade-off. The ability of the NEAT algorithm to evolve NNs to solve an RL task is highly dependent on these four hyperparameters because they control the rate of exploration when evolving ANN topologies. Their effect on the rate of exploration is dependent on each other. In the next section, NEAT hyperparameter values are automatically tuned, and in experiment 2 of the next section, the four hyperparameters mentioned above are simultaneously tuned.

A policy-based solely on exploration would always yield a non-optimal reward, while a policy based solely on exploitation has the risk of converging in a local minima. Striking a balance between exploration and exploitation can increase the chance of finding an optimal solution. This balance is known as the *Exploration vs. Exploitation Trade-off* in RL.

There are a few possible areas that this work could be extended to deal with. One such area is discrete hyperparameter values, discussed in section 8.1.

Chapter 6

Experiments: NEAT hyperparameter Tuning

NEAT has many hyperparameters and the convergence of the NEAT algorithm to a solution to an RL task is highly dependant on these hyperparameters. In this section, we explore the effectiveness of NEAT hyperparameter tuning algorithms and hyperparameter transfer learning (meta-transfer learning). As previously mentioned, hyperparameter tuning is a form of meta-optimisation [26], since optimisation of an optimisation algorithm is taking place. Specifically, in the context of NEAT, this means that while NEAT's objective is to find a suitable set of model parameters that solve a given RL task, the hyperparameter tuning algorithm's objective is to find a set of hyperparameters that allow NEAT to converge to suitable model parameters as quickly as possible. In the following sections, three experiments are performed and two hyperparameter tuning algorithms are compared, namely Bayesian Optimisation and random search. The three experiments are:

- Experiment 1: Exclusive Or. This is a simple proof of concept, tuning a single NEAT hyperparameter, showing that there are certain scenarios in which Bayesian Optimisation is more effective than random search for hyperparameter tuning.
- Experiment 2: CartPole. This is a more complex experiment where a total of eight full trials of NEAT hyperparameter tuning are performed. For the first four, of the eight trials, Bayesian Optimisation is used to find suitable sets of NEAT hyperparameters. Next, the final four trials of NEAT hyperparameter tuning are performed using random search. This

is done in order to compare the effectiveness of Bayesian Optimisation against random search. The resulting eight NEAT hyperparameter sets are then used in the next (final) experiment.

- Experiment 3: Lunar Lander. Finally, the eight sets of NEAT hyperparameters found in the previous eight trials, found in experiment 2, are used to solve an even more complex RL task. The results found using the eight hyperparameter sets on the new more complex task (Lunar Lander) are then compared to results found using ten randomly generated NEAT hyperparameter sets on the same task (Lunar Lander). Using NEAT hyperparameters on a new task, that were found while solving the previous task, is effectively a form of meta transfer learning[1].

6.1 Experiment Design

For experiment 1 and experiment 2, BO is used to find the sets of NEAT hyperparameters that result in the lowest average number of generations until convergence to a solution for a given RL task [2]. The goal of the experiments is to gauge how effective Bayesian Optimisation is as a hyperparameter tuning algorithm for NEAT's hyperparameters. The results are compared to a random search in the same hyperparameter space. Again, the criteria by which a set of hyperparameters is judged is the average number of NEAT generations, until convergence to a solution, for a given RL task.

Figure 6.1 details the process described in the paragraph above. First, using BO, a set of NEAT hyperparameter values that maximise the Expected Improvement acquisition function are used to create ten populations of NEAT agents. Since no evaluation of hyperparameters has been done, and the GP is still a GP prior, the first hyperparameter sets are randomly initialised. NEAT is then employed to solve the given RL task ten times, once for each of the ten different

[1] Transfer learning, in short, is a machine learning method where model parameters developed for a task are reused as the starting point for a model on a second, previously unseen, task [3]. Meta-transfer learning differs from transfer learning in that instead of reusing model parameters on a new task, algorithm hyperparameters are reused.

[2] The number of generations is averaged over ten full runs using a single set of hyperparameters, for each run. This is done in order to reduce the effect of the random seed so that a fair comparison can be made between different sets of hyperparameters. One full run is from a naive randomly initialised NEAT population that has been evolved using the NEAT genetic algorithm to a NEAT population where at least one member agent of the population can solve the RL task at hand is considered one full run.

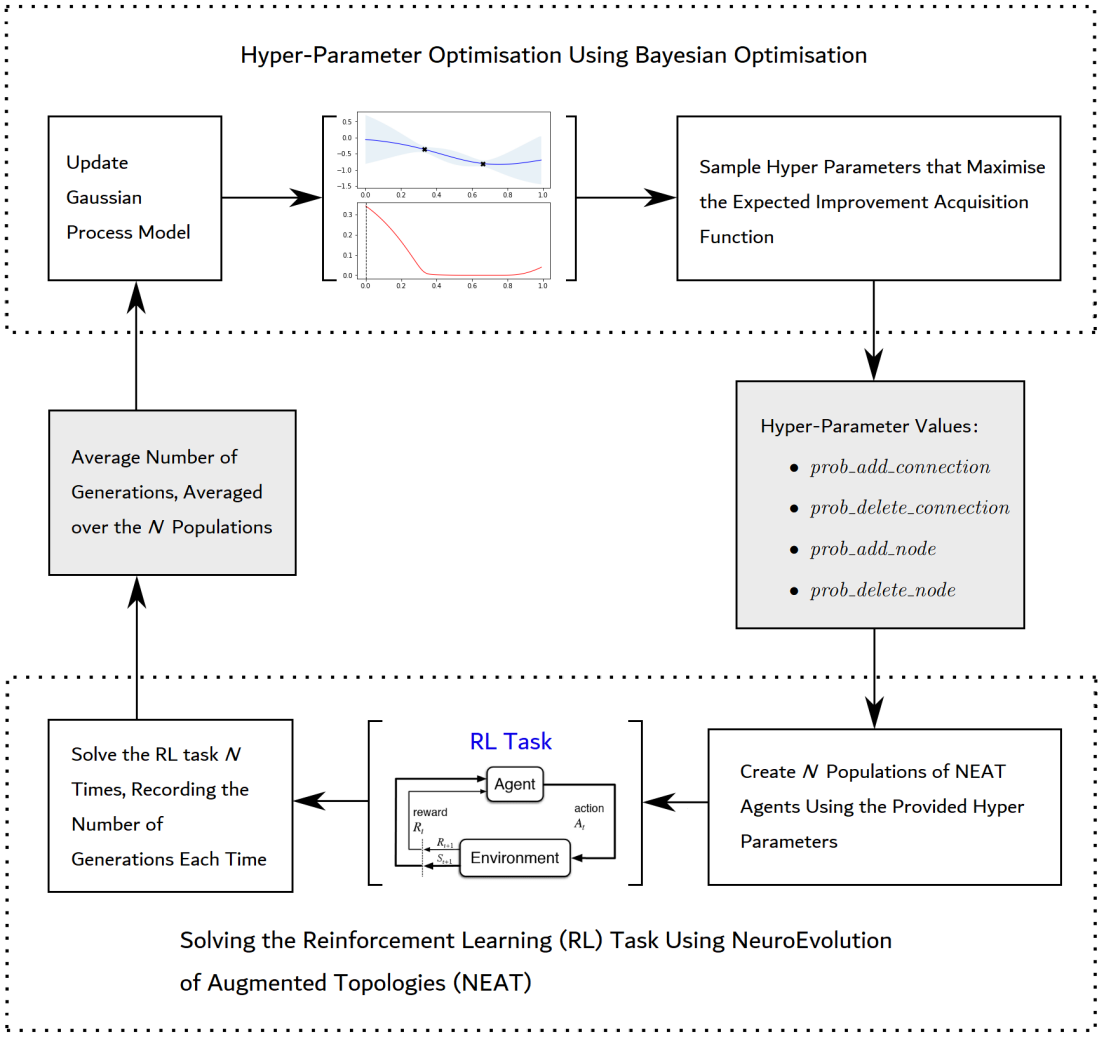


Figure 6.1: The optimisation cycle for experiment 1 and experiment 2 in this chapter, outlining the link between the two optimisation processes, BO and NEAT. In experiment 3, hyperparameter values found while optimising hyperparameters in experiment 2 are used on a new unseen task.

populations. The average number of NEAT generations until convergence to a solution to the RL task, averaged over the ten full NEAT runs, is then used as a data point along with the hyperparameter values to update the BO hyperparameter tuning algorithms' GP. Using the Expected Improvement BO algorithm a new set of NEAT hyperparameter values is proposed. This process is repeated until some prespecified BO stopping criteria is met.

Once this process is complete, and an optimal set of hyperparameters has been found, the optimisation cycle is repeated, but instead of using BO to find the optimal set of hyperparameters, a random search is performed in the same hyperparameter space to be used as a baseline to compare BO against.

An RL task is characterised by an *Agent* and the *Environment* that the agent is in. For all experiments in this section, the RL tasks are posed as MDPs. To better understand the use of Bayesian Optimisation as a means to find a suitable value for a NEAT hyperparameter or a set of values for a set of NEAT hyperparameters, experiments on three different RL tasks are performed. The tasks are outlined in greater detail in the subsections below. They appear in increasing order of complexity, that is the complexity of the ANN required to solve the given RL task.

Experiment 1 is a proof of concept and focuses on the tuning of the single NEAT hyperparameter $enabled\_mutate\_rate$. In the second experiment of this section the same NEAT hyperparameters: $node\_add\_prob$, $node\_delete\_prob$, $connection\_add\_prob$ and $connection\_delete\_prob$ are simultaneously tuned using BO. In the third experiment, the effectiveness of different sets of values for the NEAT hyperparameters (The eight different sets of values for the four NEAT hyperparameters found in the CartPole MDP in experiment 2) is examined the Lunar Lander task. The four NEAT hyperparameters mentioned above were chosen for tuning, out of the 54 available NEAT hyperparameters, because they are all probabilities - therefore limited to values on the interval $[0, 1]$ - and having a predefined continuous interval is required when modeling using a GP for BO.

For clarity, the reason for averaging over N NEAT populations is:
A NEAT population consists of agents, characterised by their hyperparameters and their policy networks. The policy networks are randomly initialised. The number of NEAT generations until convergence for a given set of hyperparameters is dependant on this random initialisation. Comparing two NEAT populations, both with an identical set of hyperparameters, one population may lead to a solution for the given RL task within fewer generations than the other because of the random initialisation of each agent's policy network. This means that using a single population's number of generations until convergence is a noisy criterion by which to judge a set of hyperparameters. In order to reduce this noise, the average number of generations averaged over N randomly initialised NEAT populations, is used as the criteria to judge a set of NEAT hyperparameters.

Throughout the three experiments, different experiment parameters are used. It is important to draw a distinction between the aforementioned NEAT hyperparameters being tuned, by the hyperparameter tuning algorithms, and the experiment parameters.

6.1.1 Experiment Parameters

Experiment parameters are the parameter values that the three experiments are initialized with. The following list details the experiment parameters.

- T: The RL MDP task being solved.
- $|P|$: For NEAT population P, $|P|$ is the number of NEAT agents (NEAT genomes) in the population.
- N: The number of NEAT populations initialised with the same hyperparameter values, used to solve the same RL task. During hyperparameter tuning, the average of the N population fitness's is then taken and used as a score for the specific set of hyperparameter values that the N populations were initialised with.
- $Steps_{max}$: The maximum number of steps permitted in the RL MDP, T, per episode.
- $Gens_{max}$: The maximum number of NEAT generations permitted per trial.
- $Genome_{runs}$: The number of times to run a single NEAT agent. The average of these runs is used as the score for that specific NEAT agent which contributes to the overall fitness of the population.
- $Trials_{max}$ The maximum number of hyperparameter tuning iterations permitted. One hyperparameter tuning iteration is one trial.
- D_{ij}: Data set generated in experiment $i\ in\{1, 2, 3\}$ using hyperparameter tuning algorithm $j \in \{BO, RandomSearch\}$. Each entry in D_{ij} is a tuple, $(Score, Hyperparameters)$, which is a record of the scores attained when using a specific set of hyperparameter values.

These experiment parameters are left unchanged throughout the course of each experiment, but each experiment has its own set of experiment parameter values. An example of the reasoning behind different experiment parameters between experiments is the value for N, the number of NEAT populations to be averaged over. In experiment 1 (the Exclusive Or experiment) N is set

to 20, while N for experiment 2 (the CartPole experiment) is 10. This is because the CartPole environment is significantly more computationally expensive to simulate than the Exclusive Or environment, so a lower value for N was required in order to complete the CartPole experiment.

The experiment parameters for each experiment are further detailed in 6.2.1, 6.3.1 and 6.4.1.

6.2 Exclusive Or

The RL task at hand is the XOR problem. The XOR - denoted $\oplus$ - or the "exclusive or" problem is a simple classification problem characterized by the truth table below: Thinking of

p	q	$p \oplus q$
0	0	0
0	1	1
1	0	1
1	1	0

Table 6.1: The XOR truth table. The first two columns represent all truth-value combinations that propositions p and q can take. The last column, $p \oplus q$, shows the output of the XOR operator given the different truth value p and q pairs as inputs.

XOR as a logic gate, the output from the gate is one when exactly one of the two inputs is one, and zero for any other input pair. The XOR classification problem is plotted below. As a classification problem, XOR is a simple example of a linearly inseparable pattern and is often used to demonstrate the effective use of an NN, with at least one hidden layer and non-linear activation functions, as a nonlinear classifier.

Although the XOR problem is a classification problem, classification problems can be posed as RL tasks by setting the fitness function to be a function of the error between the target classification and the output of an RL ANN. Moreover, while skewed class distributions are not an issue for the XOR problem, posing classification problems as RL tasks have been shown to reduce problems associated with skewed class distributions [27].

Let y_i be the output from an XOR logic gate for inputs p_i and q_i. Set as an RL task, each

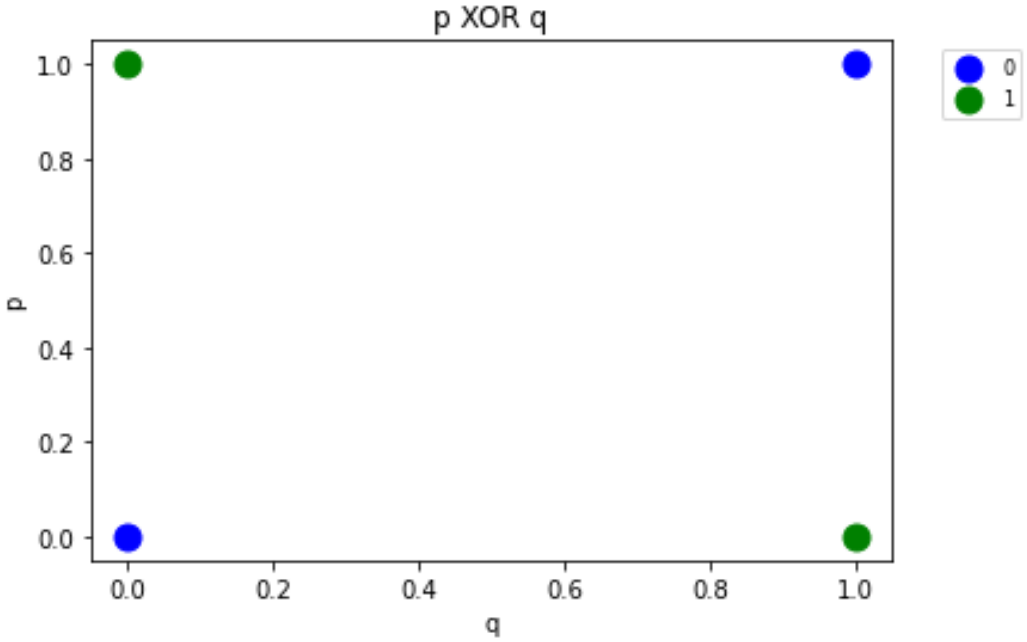


Figure 6.2: The XOR classification problem. It is impossible to separate the two classes with a single line.

agent's fitness function for the XOR problem can be defined as:

$$fitness = \left(4 - \sum_{i=1}^{4} |y_i - \pi\left(p_i, q_i\right)|\right)^2$$

Here $\pi(p_i, q_i)$ is an agent's policy network, π, evaluated at p_i and q_i. Each agent is required to correctly classify four points, which if correctly classified, would result in $|y_i - \pi\left(p_i, q_i\right)|$, from the fitness function above, being equal to zero for all i and an overall fitness score of four. The task is considered solved if an agent's fitness score is above 3.9. This fitness function is from the book "Hands-On NeuroEvolution with Python" [28], and it is not necessarily the only fitness function that could be used for this RL task. At this point, it is important to remember that, in all of the experiments, NEAT is being used to maximise the fitness, while BO is being used to minimise the number of generations that the NEAT algorithm evolves for in order to maximise the fitness.

6.2.1 Exclusive Or Experiment

For the first, proof of concept, experiment the single NEAT hyperparameter *enabled_mutate_rate* was chosen for tuning. It was chosen because it is one of the few hyperparameters that is not coupled to another hyperparameter in the way that, for example, *node_add_prob* and *node_delete_prob* are coupled to each other.

The two hyperparameter tuning algorithms, BO and random search are used to find a value for the aforementioned NEAT hyperparameter *enabled_mutate_rate*. For a deeper dive into the technical underpinning of this experiment, refer to the pseudocode algorithms 3 through 5 in the "Pseudocode for experiment 1" section of the appendix.

Exclusive Or Experiment Parameters

Parameter	Value
T:	XOR
$\|P\|$:	150
N:	20
$Steps_{max}$:	1
$Gens_{max}$:	500
$Trials_{max}$	12
$Genome_{runs}$:	1

Table 6.2: $Genome_{runs}$ and $Steps_{max}$ are both set to 1 since the XOR problem is a classification problem and there are no steps in the environment, which also means that there are no randomly initialised environment states. Since this is a classification problem it is relatively computationally inexpensive to simulate when compared to the other environments in experiment 2 and experiment 3.

6.2.2 Exclusive Or Experiment Results

In Table 6.3, twelve iterations of both NEAT hyperparameter tuning algorithms are compared. The random search is initially much better than BO at finding a suitable value for the single hyperparameter being tuned. It is important to note that the first two BO iterations are exploratory. By the third iteration, which is the first iteration that the acquisition function is used, the mean number of generations immediately improves over the best random search result found so far. All subsequent sampling takes place at the same value for the single hyperparam-

Iteration	Random Trial	BO Trial
1	72.2	180
2	441.8	404.6
3	253.8	60.2
4	101.8	81.6
5	121.8	51.8
6	240.2	68
7	150.4	82.4
8	70.8	71.4
9	65.4	61.2
10	415.0	62.2
11	394.0	49
12	425.6	68.2
μ	229.4	103.4
σ	146.2	96.6

Table 6.3: Twelve sequential iterations of NEAT hyperparameter optimisation using random search and twelve using Bayesian Optimisation are compared. The column titled “Random" shows the results from the twelve iterations of NEAT hyperparameter tuning using random search. The column titled “BO" shows the results from the twelve iterations of NEAT hyperparameter tuning using Bayesian Optimisation. The two last rows show the mean and the standard deviation of each column respectively. For the first two iterations, BO chooses random hyperparameter values to test, all subsequent BO parameter selections (from iteration 3 to iteration 12) are chosen by maximising the EI function. Initially, random search appears to be doing better than BO, for iterations 1 and 2, but because of BO's pointed search, hyperparameter values chosen for most subsequent BO iterations outperform the random search algorithm. This results in BO having a far lower mean and standard deviation.

eter: $enabled\_mutate\_rate := 0.$ [3], All subsequent variation is a consequence of the random initialisation of the NEAT populations and the randomness associated with NEAT mutation, crossover. The mean of the twelve BO results is far lower than the mean of the twelve random search results. This indicates that BO is performing a more pointed search in a region of the search space that the BO algorithm has deemed useful for exploration.

In the Figure 6.3, we can see that after iteration 3 the $enabled\_mutate\_rate$ value, recommended by the acquisition function, is near-optimal and all subsequent exploration takes place in a neighborhood around this value. BO exploration could have been halted earlier if using two acquisition function recommendations proposed in the same location, to some tolerance, was used as the stopping condition instead of using a maximum of ten iterations (with the two additional initial exploratory samples, sampled at values 0.33333, and 0.66666 of $enabled\_mutate\_rate$).

[3]This is an edge value in the domain, and BO is particularly good at exploring the edge values when compared to random search.

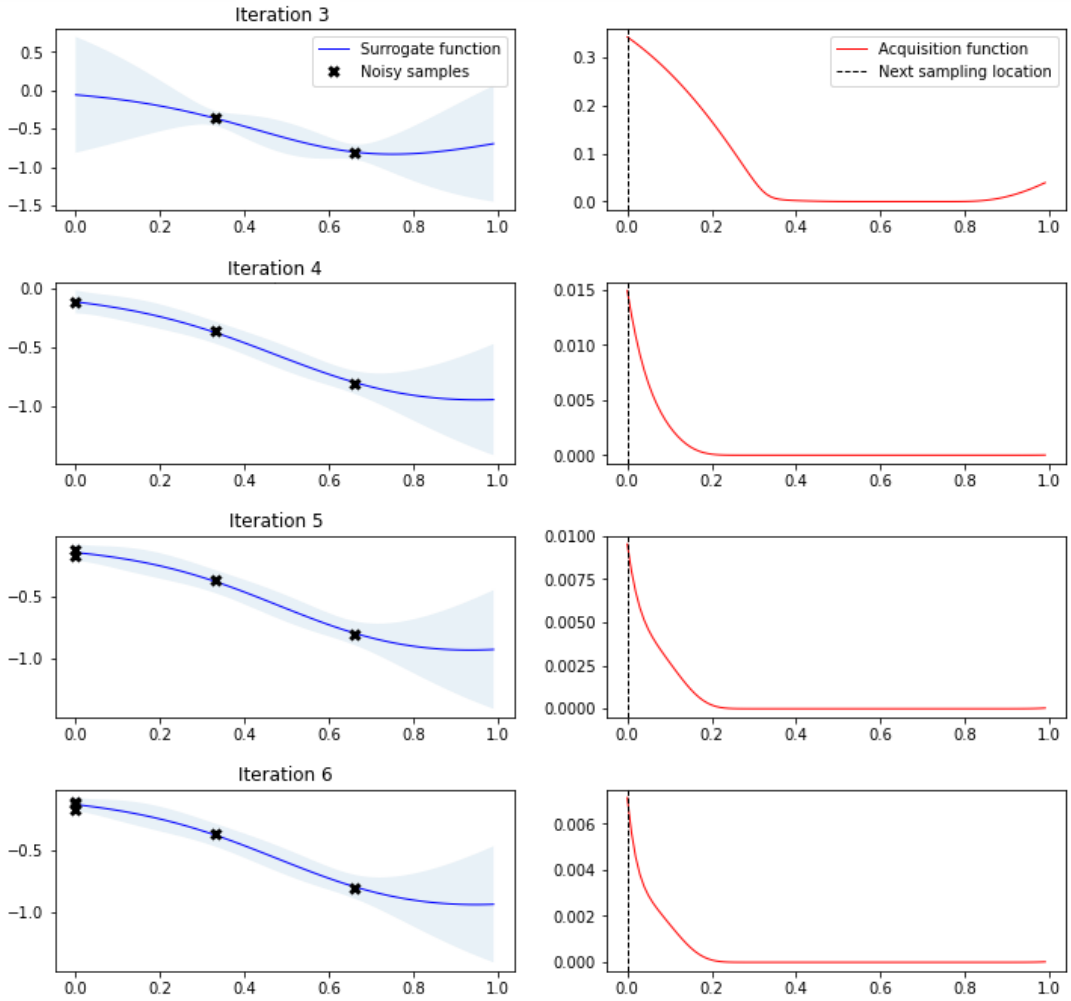


Figure 6.3: The x axis in all subplots represents the value chosen for $enable_mutate_rate$, and the y axis represents the normalised performance of NEAT given y. The first four iterations of Bayesian Optimisation guided using the acquisition function (The first two samples were sampled at 0.33333 and 0.66666, after which the EI acquisition function proposes the next sampling locations), where the optimal value for the single NEAT is found within three iterations. The left-hand column plots show the GP iteratively fit that approximates the relationship between the value for the NEAT hyperparameter and its corresponding speed to convergence. The right column of plots shows the acquisition function recommending the same point multiple times, indicating convergence. The x-axis of the left GP plots is the negative normalised number of generations until convergence. The negative is used because we seek to minimise the number of generations and the EI acquisition function used here was built to maximise a function.

6.3 CartPole

For the next MPD, the RL task is a pendulum attached to a cart needs to be balanced. The goal is to find a policy that ensures the pendulum remains balanced for a certain amount of steps. The pendulum is on an “un-actuated joint" meaning that agents have no direct control over the pendulum itself. Instead, at any given step, agents have indirect control over the pendulum using actions -1 and $+1$ that move the cart to the left or the right respectively. In Figure 6.3 the original MDP on the left and a recreation of the MDP on the right.

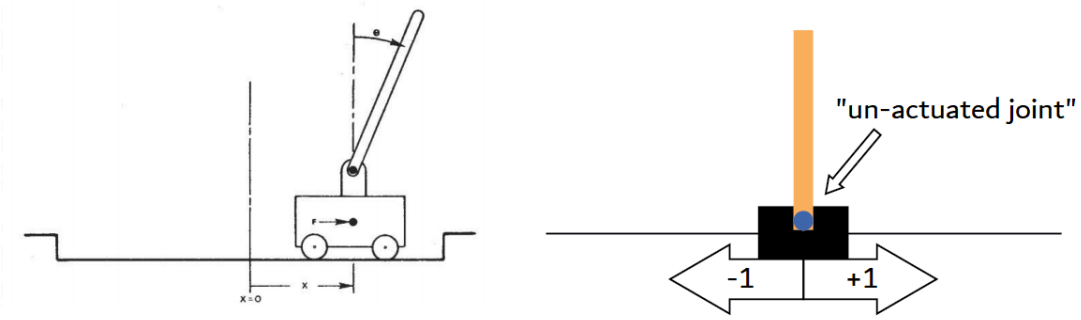


Figure 6.4: The left half of the figure is a diagram of the original RL task outlined in the paper "Neuronlike adaptive elements that can solve difficult learning control problems" [30], written by Sutton, Barto and Anderson. The right half of the figure is a screenshot with labels from an OpenAI environment, a recreation of the original RL task, namely "CartPole-v0" [31], which is used in this experiment. Both images illustrate the CartPole MDP

The set of valid actions available to an agent in the CartPole MDP is the discrete set: $\{-1.0, 1.0\}$. An MDP's observation space is the space that describes the format of the observations that agents can observe. The observation space for this MDP consists of five values, the position of the cart relative to the centre of the environment, the angle of the pole, their respective derivatives, and the Boolean "done" which takes the value 0 when the task is still in progress, and 1 if the task is completed. Completed in this setting can mean either lost or won.

6.3.1 CartPole Experiment

The goal for this experiment is to run eight trials of NEAT hyperparameter tuning in order to find eight sets of NEAT hyperparameter values that allow NEAT to solve the CartPole MDP within as few generations as possible. Initially, four trials of BO are run to find four sets of NEAT hyperparameter values, and the remaining four sets are found using random search. For all eight trials, the four NEAT hyperparameters listed below are simultaneously tuned:

- *prob_ add_ connection*
- *prob_ delete_ connection*
- *prob_ add_ node*
- *prob_ delete_ node*

The four NEAT hyperparameters above were selected for this experiment because they are limited to values on the interval $[0, 1]$ since they are all probabilities. Moreover, these hyperparameters were chosen because they are inter-dependant on each other and an incorrect balance between these hyperparameters leads to slow convergence, or no convergence at all, of the NEAT algorithm to an acceptable solution for the CartPole task.

The rewards returned by the CartPole MDP are simply +1 for each time step that the pole remains balanced. For this experiment, the NEAT fitness function is chosen to be the mean cumulative reward received from the CartPole MDP over 100 episodes. This was selected as the fitness function because each episode of CartPole has a maximum of 200 time steps, and if a mean score of 195.0 over 100 consecutive episodes is attained then the MDP is considered solved. [4]

In summary, each of the eight trials make use of either BO or random search to minimise the average number of NEAT generations until the CartPole RL task is solved. Eight trials are performed and their results are compared in the Table 6.4. Refer to section “Pseudocode for experiment 2" in the appendix for the in-depth technical details of how each trial is structured.

[4] Using the Python package "GPyOpt" BO is performed, as detailed in the "Experiment Design" section above, with the maximum number of BO iterations set to 100. The maximum number of BO iterations mentioned here does not include 10 initial random iterations. The purpose of the 10 initial iterations is to get enough data to fit the initial GP to, after which the acquisition function is used to guide all of the subsequent 100 sample locations. With this in mind random search with 110 iterations, in the same hyperparameter space, is performed as a benchmark to compare BO against.

CartPole Experiment Parameters

Parameter	Value
T:	CartPole
$\|P\|$:	150
N:	50
$Steps_{max}$:	1000
$Gens_{max}$:	150
$Trials_{max}$	110
$Genome_{runs}$:	100

Table 6.4: Experiment parameters for experiment 2

Here $Steps_{max}$ is 1000, while $Gens_{max}$ is far lower than in the previous experiment. Since $Steps_{max}$ is the maximum number of steps that each agent can take in the environment, 1000 is a substantial increase and results in a huge computational overhead for solving the CartPole MDP. To reduce this computational overhead $Gens_{max}$ was reduced from 500 in experiment 1 to 150 for the CartPole experiment. This decrease was possible because NEAT typically solves the CartPole MDP within fewer generations than it solves the XOR MDP. This is because XOR is a classification problem which requires a stricter stopping condition, because classifications must be correct to some tolerance. $Genome_{runs}$ has also increased, from 1 to 100. This was necessary because NEAT agents are sensitive to the random initialisation of the environment. Similarly, averaging over ten NEAT populations, since $N := 50$, counteracts the effect that randomly initialised agents within NEAT populations have on the score for a set of hyperparameters, essentially making the score for the hyperparameter values less noisy.

6.3.2 CartPole Experiment Results

Trial Number	Mean Generations
1	3.4
2	3.4
3	3.4
4	3.4
5	3.4
6	3.6
7	3.4
8	3.4

Table 6.5: Results from trials 1 through 8, using Bayesian Optimisation to tune the four NEAT hyperparameters. This table differs from Table 6.3 in that Table 6.3 was a table illustrating 12 iterations within 1 trial, and this table illustrates the best results obtained over eight full BO trials. The best results (Again, the best results in this context refer to the best performing NEAT genomes found while evolving NEAT using hyperparameter values found using either BO or random search. The NEAT genomes are judged by the amount of generation until convergence, with fewer being better.) in this context refer to the best performing NEAT genomes found while evolving NEAT using hyperparameter values found using either BO or random search. The NEAT genomes are judged by the amount of generation until convergence, with fewer being better. Here all but one of the trials mean generations is 3.4. Trial 6 shows that not all BO trials reach the low of 3.4.

Trial Number	Mean Generations
9	3.6
10	3.4
11	3.4
12	3.7
13	3.7
14	3.7
15	3.8
16	3.7

Table 6.6: Results from trials 9 through 16, using random search to tune the four NEAT hyperparameters.

From the two above tables, it is clear that using Bayesian Optimisation consistently finds a set of NEAT hyperparameters that result in a mean of 3.4 NEAT generations until convergence to a solution for the CartPole MDP. hyperparameter values found using random search lead to similar outcomes, without consistency. The worst mean number of generations from all sixteen trials was 3.8, found using random search, which is 11.76% worse than 3.4.

Interpreting the CartPole Experiment Results

Each NEAT Genome is evaluated ten times. Each evaluation of the NEAT Genome is called an iteration. The number of generations until convergence to a solution to the RL task at hand is the score for that specific Genome at that specific evaluation iteration. The mean of the ten scores is then used as the score for the specific hyperparameter value set proposed by one of the two hyperparameter tuning algorithms. The lowest scoring hyperparameter value set is deemed the best, because it indicates that that specific set of hyerparameter values lead to the NEAT algorithm finding a Genome that solves the RL task at hand within the fewest number of generations. In Table 6.7 we examine the number of generations until convergence of the genomes with the lowest mean score (the mean across all ten iterations) from BO trials five through eight and Random trials thirteen through sixteen.

Trial Number	It 1	It 2	It 3	It 4	It 5	It 6	It 7	It 8	It 9	It 10
5	3	3	4	3	4	4	3	3	4	3
6	4	4	3	3	4	3	3	4	4	3
7	4	3	4	3	3	3	4	4	5	3
8	5	3	3	3	3	5	3	3	3	3
13	4	4	5	3	3	4	3	3	4	4
14	4	3	3	4	4	3	4	4	5	3
15	4	4	4	5	3	3	4	4	4	3
16	3	3	5	3	4	3	3	4	5	4

Table 6.7: Comparing the number of generations until convergence for each of the ten iterations of the genomes with the lowest mean score from BO trials five through eight and Random trials thirteen through sixteen.

Figure 6.5 and 6.7 show that the distribution of scores is almost never the same across trials. That is, in each of the eight trials (four BO trials and four Random trials), the found hyper parameters lead to different distributions of scores.

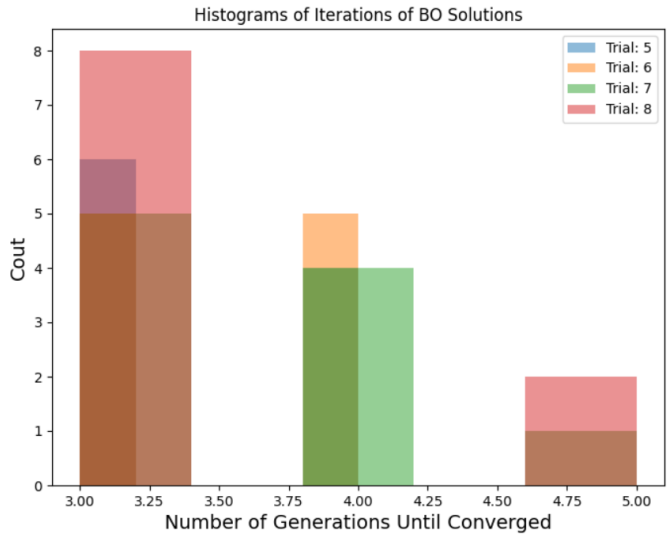


Figure 6.5: BO Histograms. Overlapping histograms showing the distribution of NEAT generations until convergence for each of the four best NEAT genomes found in the four BO trials, trials five through eight.

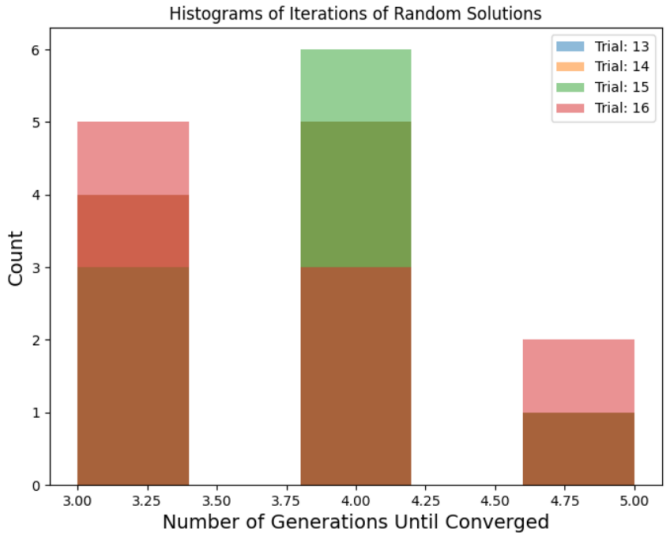


Figure 6.6: Random Histograms. Overlapping histograms showing the distribution of NEAT generations until convergence for each of the four best NEAT genomes found in the four Random trials, trials thirteen through sixteen.

In Table 6.8, we compare two of the results, one from the first BO trial (Trial 9) and one from the first random search trial (Trial 9). The BO trial (Trial 1) performed 5.56% better

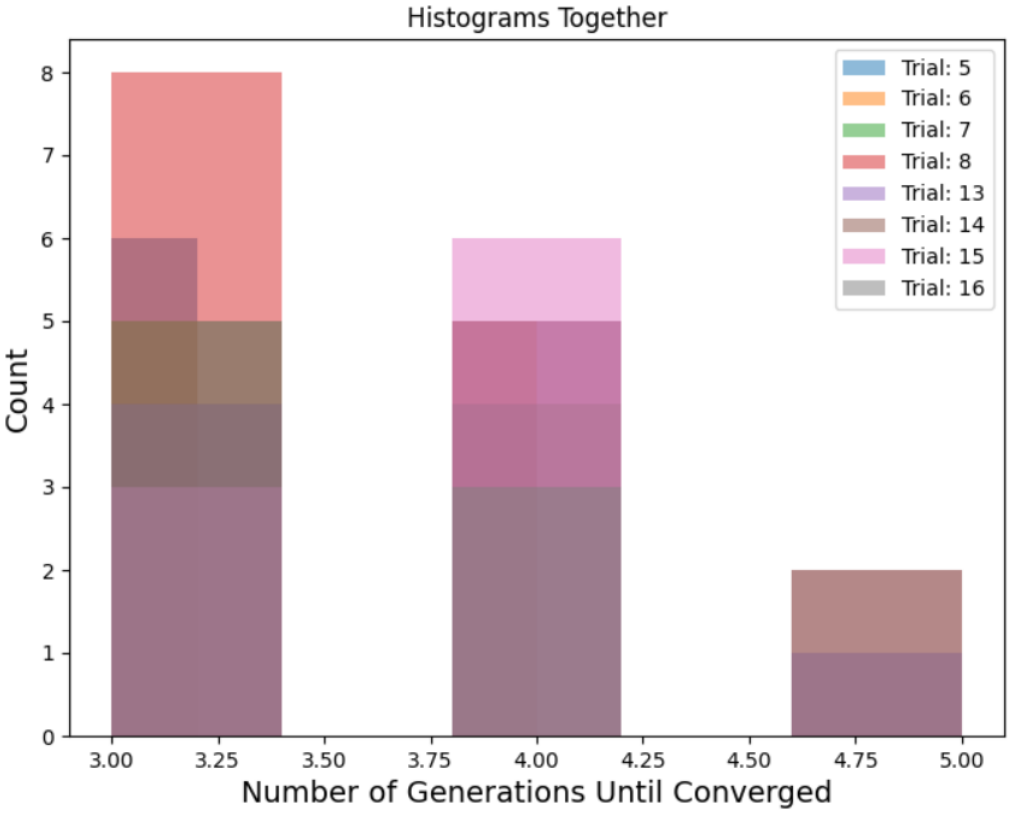


Figure 6.7: All histograms from Figure 6.5 and Figure 6.6 plotted together.

Metric	Random Trial	BO Trial	Improvement over Random (%)
Mean Generations	3.6	3.4	+5.56
Standard Deviation	9.75	20.16	-106.77
Run Time (s)	13996	34608	-147.27

Table 6.8: Comparing the mean generations, the standard deviation of generations, and run time between a Random Trial and a BO Trial.

than the random search trial (Trial 9), but the BO trial was nearly two and a half times slower (in seconds) than the random search trial. 34608 seconds (9.61 hours) was a typical BO trial length, and in general, the random searches were much quicker. This is likely due to the extra computational overhead associated with the cost of computing the four-dimensional Gaussian Process posterior at each iteration.

Interestingly, unlike in experiment 1, here the standard deviation of mean NEAT generations from the BO trial is more than twice that of the standard deviation of the random

search trial. This is likely due to the combination of an increased search space and BO's ability to explore edge cases in the domain, leading to more extreme results for some hyperparameter value combinations. One aspect of our results that we were not able to resolve is the large standard deviation of the BO results. This is left to future work.

Trial	Connection Add	Connection Del	Node Add	Node Del
Trial 1, BO Trial	1.0	0.44460	0.0	0.0
Trial 9, Random Trial	0.99146	0.37979	0.08910	0.47478

Table 6.9: Results from trials 1, using Bayesian Optimisation to tune the four NEAT hyperparameters, compared to the Results from trials 9, using random search to tune the four NEAT hyperparameters.

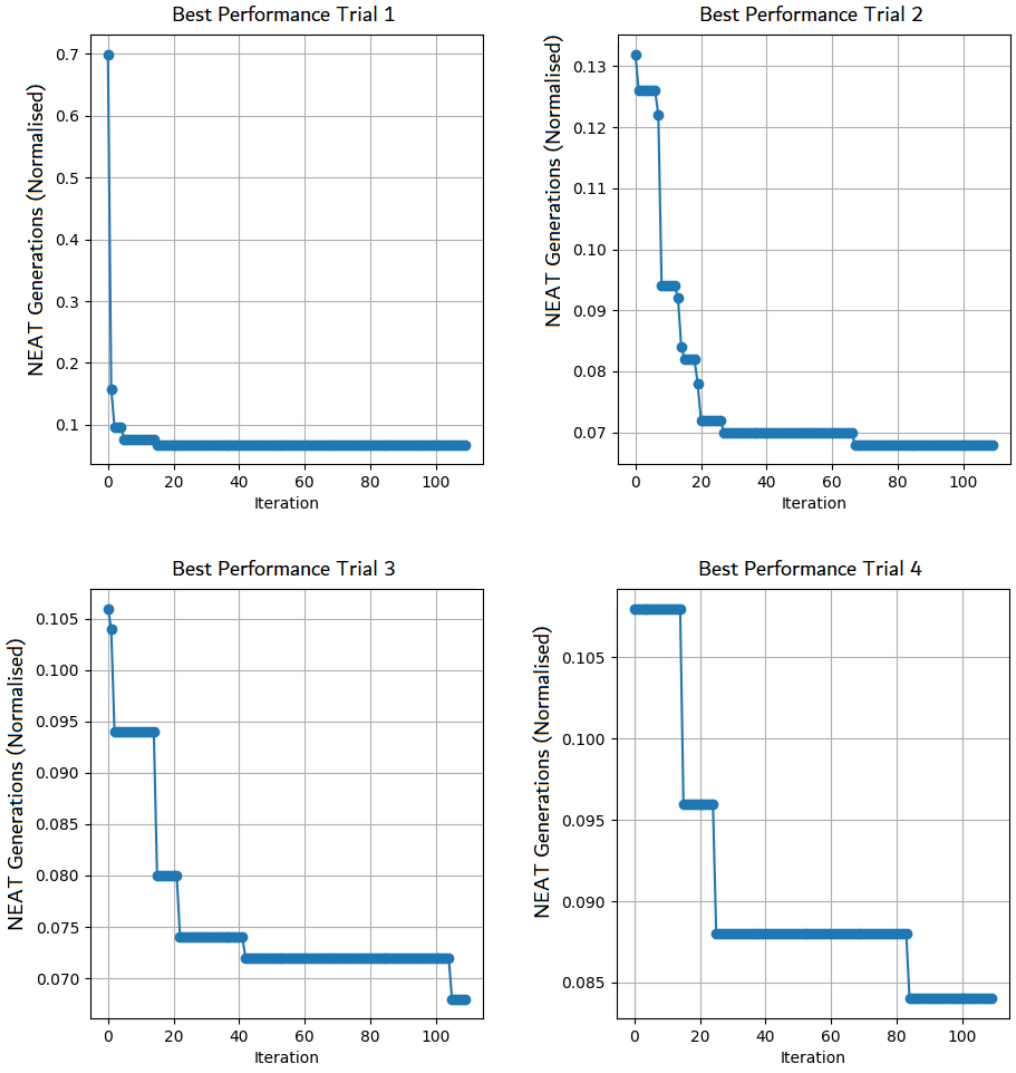


Figure 6.8: The number of NEAT generations is plotted against BO iterations, for each of the first four BO trials, illustrating the convergence of the BO hyperparameter tuning process to a set of hyperparameters that minimise the number of NEAT generations required to solve the CartPole task. The algorithm always converges to the same number 0.068 (which is 3.4 normalized using the $N := 50$ populations that were averaged over).

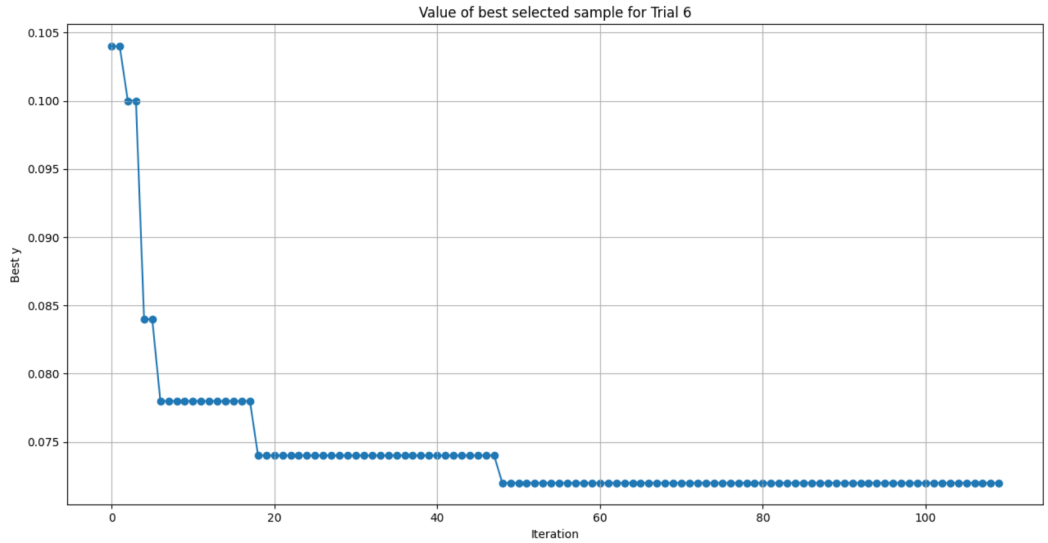


Figure 6.9: The number of NEAT generations is plotted against BO iterations, for BO Trial 6. The BO algorithm usually converges to the same number 0.068, but this time it converges to 0.072 (which is 3.6 normalized using the $N := 50$ populations that were averaged over).

Now that sixteen sets of NEAT hyperparameter values have been found for the CartPole MDP, the next experiment explores the use of these NEAT hyperparameter values on a new, previously unseen RL MDP.

6.4 Lunar Lander

In this experiment, the effectiveness of meta-transfer learning is examined. To this end, eight of the NEAT hyperparameter value sets found in the previous experiment are used to solve the third and final RL task, that is the Lunar Lander MDP. The first four where obtained using BO, and are the resulting hyperparameter sets from trials one through four. The final four of the eight hyperparameter sets were found using the random search algorithm, and are a result of trials nine through twelve. They are used to find a solution to the Lunar Lander MDP. In the Lunar Lander MDP, agents can control a spaceship, called a lander, using actions from the action set {*Fire Left Thruster, Fire Right Thruster, Fire center Thruster, Do Nothing*} at each time step. The goal is to gently land the lander between two flags on the moon's surface in the MDP. The rewards returned by the Lunar Lander MDP are slightly more complicated than the rewards returned by the CartPole MDP. The Lunar Lander rewards are [32]:

- +100 if the lander lands anywhere on the moon's surface.
- -100 if the lander crashes anywhere on the moon's surface.
- +100 to +140 for moving from the top of the environment to the bottom. This is a function of the fuel spent while landing, and whether or not a landed leg of the lander breaks contact with the bottom of the environment after making initial contact with the bottom of the environment.
- -3 each time the center thruster is fired. This penalty is intended to encourage agents to land the lander as quickly as possible, and save fuel.
- +10 for each of the two legs that makes contact with the moon's surface.

The MDP's state space consists of the x-axis and y-axis positional values of the lander, the x,y-axis lander velocity terms, lander angle, and angular velocity, left and right lander leg contact points (Boolean) [32]. Here the x,y-axis lander velocity terms are the velocity derivatives of the lander. [5]

[5] Whether or not any task can be posed as an MDP is up for debate, but including the derivatives in this way may help satisfy the Markov Property (introduced in the RL chapter) making the task easier to pose as an MDP.

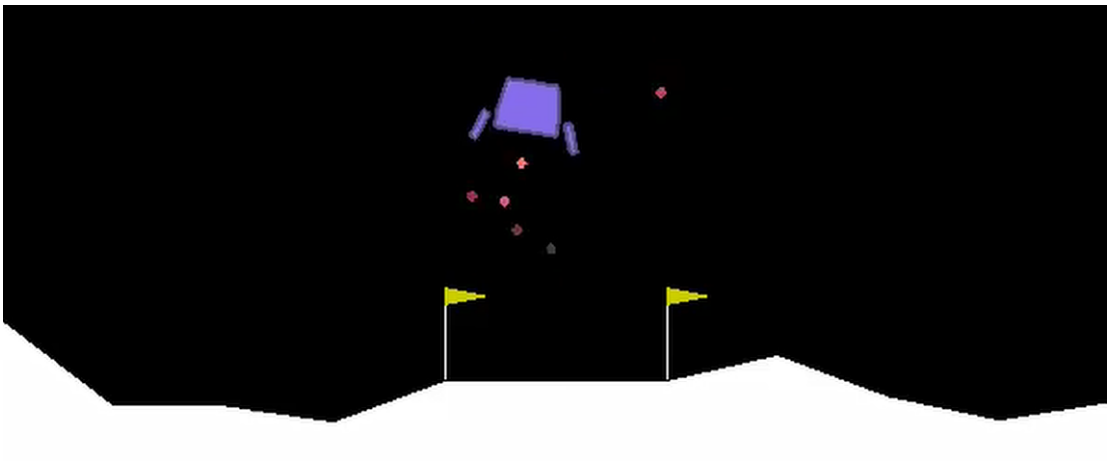


Figure 6.10: A screenshot from the Lunar Lander MDP used in this dissertation, built and maintained by OpenAI [32] on Github. The purple lander needs to be gently guided to land between the two yellow flags on the moon's surface.

6.4.1 Lunar Lander Experiment

The Lunar Lander MDP takes significantly longer to simulate and solve using NEAT than the previous MDPs, making hyperparameter tuning extremely time-consuming. In order to solve the Lunar Lander RL task, eight sets of NEAT hyperparameter values found using BO and random search, from the CartPole trials in experiment 2, are used to initialise NEAT populations to solve the Lunar Lander MDP. For clarity, BO is not used in experiment 3, but NEAT hyperparameter values found using BO for each of the eight hyperparameter value sets found using BO in experiment 2 are used here as fixed hyperparameter values. The results are then stored to be compared with ten Lunar Lander attempts using random NEAT hyperparameters values[6]. The four NEAT hyperparameters in question are, once again: *prob_add_connection, prob_delete_connection, prob_add_node, prob_delete_node.*

In order to compare different NEAT agents, the NEAT fitness function used for this experiment is the cumulative reward returned episodically by the Lunar Lander MDP. The MDP is considered solved if a cumulative reward of 200, or more, averaged over 100 consecutive episodes is attained. For this experiment, the maximum number of NEAT generations permitted is 250, which if reached, then the NEAT population is deemed not to have converged to a solution. This is 100 more generations than the amount permitted in the previous experiment due to the

[6]Note that the hyperparameter sets found using random search during CartPole optimisation are the 5^{th} to 8^{th} hyperparameter sets used in this experiment, shown in the table above, which are different from the eight random hyperparameter sets generated for the baseline to compare against.

increased complexity of this task.

Lunar Lander Experiment Parameters

Parameter	Value
T:	Lunar Lander
$\|P\|$:	250
$Steps_{max}$:	1000
$Gens_{max}$:	500
$Genome_{runs}$:	10

Table 6.10: Experiment parameters for experiment 3

The goal is simply to evaluate the eight NEAT hyperparameter sets found in experiment 2 on a new task, and not to optimise NEAT hyperparameters for the Lunar Lander MDP. This means that the experiment parameters N and $Trials_{max}$ are not used in this experiment.

NEAT takes much longer to solve the Lunar Lander MDP than the CartPole MDP. For this reason, $Gens_{max}$ is set to 500 and each NEAT population is initialised with 250 agents in order to help explore more of the solution space.

6.4.2 Lunar Lander Experiment Results

Parameters from Trial #	# of Generations	Converged
1	147	Yes
2	111	Yes
3	215	Yes
4	249	No
5	215	Yes
6	155	Yes
7	202	Yes
8	138	Yes

Table 6.11: Results from meta-transfer learning in the Lunar Lander experiment. All but one of the hyperparameter sets found from experiment 2 used here (in experiment 3) resulted in convergence to a solution. The first four of the eight hyperparameter sets used here were found using BO and the other four were found using random search in experiment 2. Interestingly the hyperparameters found in Trial 3 from experiment 2 did not lead to convergence when transferred to the Lunar Lander task.

The performance of the eight NEAT hyperparameter value sets (Four found using BO and four found using random search) is shown in Table 9, and the effectiveness of each of the NEAT hyperparameter value sets is ultimately judged by whether or not the NEAT populations initialised using the hyperparameter values converged to a solution to the Lunar Lander MDP. All but one of the eight found NEAT hyperparameter value sets lead to convergence. The set of NEAT hyperparameter values that didn't lead to convergence was the set found during Trial 4 of experiment 2, the last trial that made use of BO. Interestingly, the NEAT hyperparameter value sets found using random search all lead to convergence when used on the Lunar Lander task. This could indicate that if the goal is to use the hyperparameter values on a different task (meta-transfer learning), then BO could lead to worse results than random search, possibly due to overfitting.

The results in Table 9 are significant because finding solutions to the Lunar Lander MDP using NEAT is a difficult task, and convergence is highly dependant on NEAT's hyperparameter values. To illustrate this, Table 6.12 shows the results from ten attempts at the same task using randomly generated parameters.

In Table 6.11, the randomly generated NEAT hyperparameters were not part of the

Random Parameters #	# of Generations	Converged
1	249	No
2	249	No
3	194	Yes
4	249	No
5	249	No
6	249	No
7	249	No
8	249	No

Table 6.12: Results showing the performance of eight randomly generated NEAT hyperparameter value sets, of which only one of the eight sets lead to convergence in the Lunar Lander MDP. This indicates that the meta-transfer learning is significantly better than performing a random search directly in the Lunar Lander hyperparameter space.

Chapter 7

Conclusion

There is a high computational cost associated with solving machine learning problems, and machine learning algorithms often have a large number of tunable parameters that must be learned from experience. This is especially true of the reinforcement learning algorithm NEAT, which has more than 50 hyperparameters. The goal of learning hyperparameters is to find a combination of hyperparameters values that maximise the efficacy and accuracy of a machine learning algorithm and a poor choices for these hyperparameter values leads to slow or incomplete learning while good choices lead to quick, efficient learning. In particular we explore the issue of selecting hyperparameter values for the reinforcement learning algorithm NEAT, and employ the optimisation algorithm Bayesian Optimisation to do so. *Bayesian Optimisation* (BO) is a form of *Surrogate Optimization*. Surrogate Optimization uses an inexpensive to evaluate function to approximate an expensive to evaluate objective function.

The initial goal of this work was to experiment with and compare the two hyperparameter tuning algorithms, BO and random search, against each other. BO was initially thought to be the superior algorithm (in experiment 1), but as the complexity of the problem grew, so did the time taken per BO iteration. This is evident in experiment 2 when a $4d$ GP prior needs to be fit at each BO iteration. Ultimately there is very little benefit to using BO over random search (At worst a 0% improvement, and at best an 8.82% improvement.), and in this higher dimensional case (experiment 2), BO was far slower than random search, for the same amount of iterations.

Far more significant is the fact that, in experiment 3, hyperparameter values found

while tuning the NEAT hyperparameters for one RL task (CartPole) were effective at reducing the number of generations to convergence for another, completely separate, RL task (Lunar Lander). This indicates that there is some general “usefulness" of hyperparameter value sets, which generalise to new, unseen, tasks.

It should be noted that, while the Lunar Lander MDP is more complicated than the CartPole MDP from experiment 3 and 2 respectively[1], the state spaces in the RL MDPs both have state sequences influenced by simulated gravity. Their action spaces are also similar in that agents have control over the movement of an object within each MDP. This similarity could explain why the hyperparameter value sets found while solving one task, helped solve the other task.

Despite this similarity, the Lunar Lander task is objectively harder to simulate and solve, using NEAT, than the CartPole task. The fact that hyperparameter values found while solving the simpler task help solve the harder task illustrates the usefulness of meta-transfer learning for generalising machine learning frameworks to harder problems in similar domains.

The algorithms used to tune NEAT's hyperparameter themselves had hyperparameters that need to be tuned. This is known as the “Turtles all the way down” dilemma, where a proposed solution to a problem has, itself, the same problem. With this in mind, typically an ANN's topology parameters are considered hyperparameters, but NEAT has incorporated topology parameters such as node architecture and connection architecture parameters into its evolutionary process (Hence the name, NeuroEvolution of Augmenting Topologies). It would be interesting to find a way of recursively tuning the mutation probability hyperparameters as model parameters (not algorithm hyperparameters) in the NEAT evolutionary process, effectively getting rid of the turtles all the way down dilemma. This would make for interesting future work.

All code used for this work is available in the repository:

- `https://github.com/maaxnaax/MSc_code`

[1]The Lunar Lander MDP has a 5 dimensional action space, while the CartPole MDP has a 3 dimensional action space. The Lunar Lander MDP has an 8 dimensional observation space while the CartPole MDP has a 4 dimensional observation space.

Chapter 8

Future Work

8.1 Discrete BO for hyperparameter Tuning

This work explored the effectiveness of tuning of continuous-valued NEAT hyperparameters. It would be interesting to extend BO hyperparameter tuning to discrete-valued NEAT hyperparameters. The paper “Dealing with Integer-valued Variables in Bayesian Optimization with Gaussian Processes"[33] proposes using a modified GP kernel that uses rounding to effectively squash all input values near each other to have the same output, meaning that discrete functions can be approximated.

8.2 Using Multiple hyperparameter Values

Since there are multiple NEAT agents in a NEAT population, one could use the average of the top k hyperparameter sets from a NEAT population instead of the best hyperparameter set found for meta-transfer learning, meaning that the likelihood of overfitting may decrease, increasing the transferability of the NEAT hyperparameter value sets.

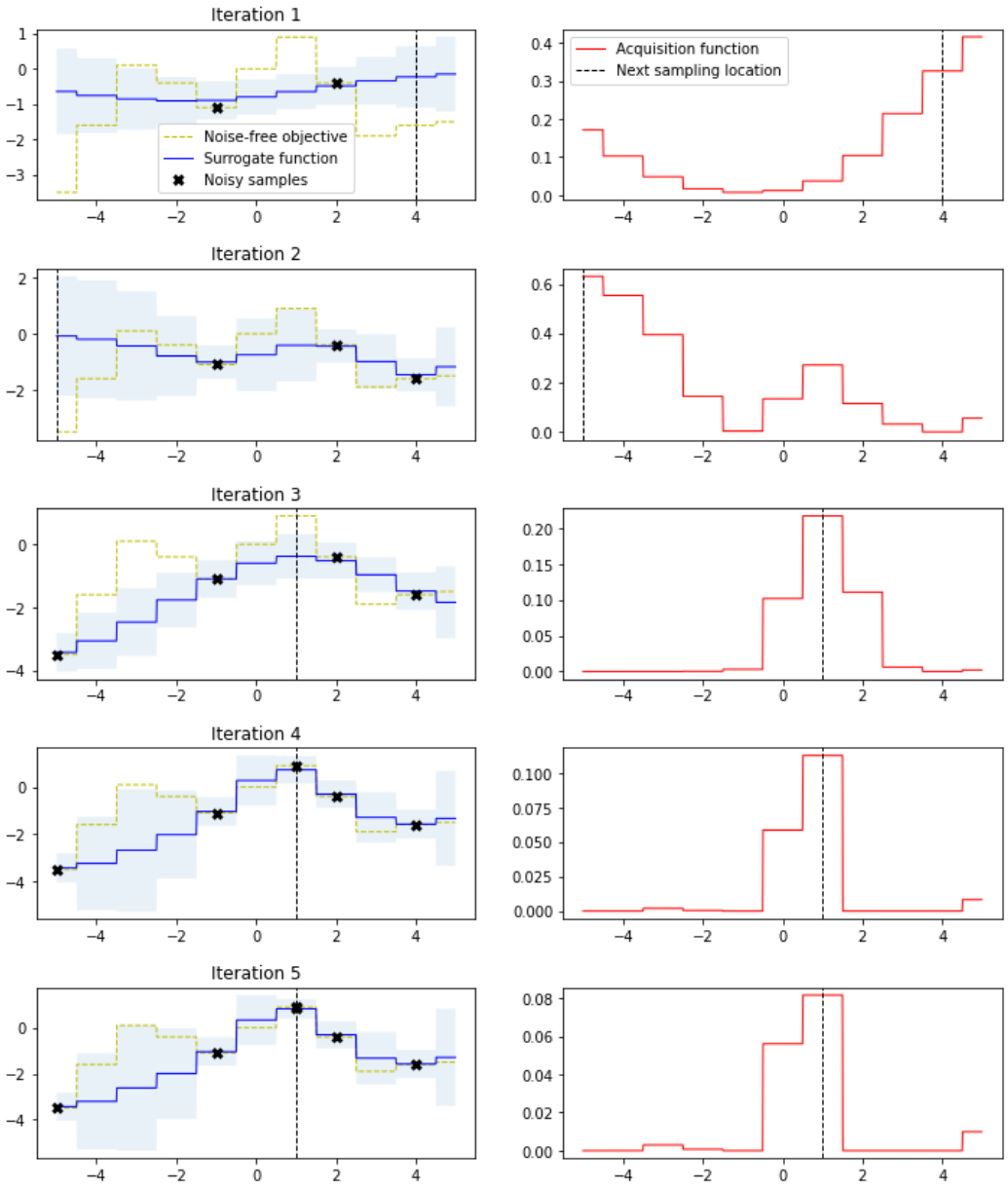


Figure 8.1: A plot generated by recreating the paper on an objective function similar to the objective function from the GP section, that is with two local optima, but only one global optimum. The discrete BO algorithm is able to find the global maxima within three iterations of BO.

Chapter 9

Appendix

9.1 Pseudocode for Experiment 1

Algorithm 3 The Procedure (function) used to evaluate a single NEAT hyperparameter on the XOR problem.

```
1: procedure TRAIN_NEAT_XOR(a)
2:     i ← 0
3:     L_g ← empty list (of NEAT generations until convergence)
4:     for i < 20 do
5:         P ← NEAT Population
6:         P.enabled_mutate_rate ← a
7:         T ← XOR MDP
8:         Evolve P until T is solved
9:         g_i ← Number of P generations until T was solved
10:        L_g.insert(g_i)
11:        i ← i + 1
12:    end for
13:    mean_gens ← mean(L_g)
14:    Return mean_gens
15: end procedure
```

The procedure (function) in the algorithm above takes a single value, a, as input. In the for loop a NEAT population is created and the NEAT hyperparameter *enabled_mutate_rate* is set to value a. The XOR task is solved, and the number of NEAT generations, until it is solved, is recorded in list L_g. The for loop is repeated ten times and the list L_g is appended each time. The mean, of the ten NEAT generations, until the XOR task is solved, is then used

as the value to be returned by the procedure.

Algorithm 4 A single Trial to tune the NEAT hyperparameter, *enabled_mutate_rate*, using random search.

$\mathcal{D}_{1R} \leftarrow \varnothing$
$i \leftarrow 0$
while $|\mathcal{D}_{1R}| < 12$ and $i < 12$ **do**
    $a \leftarrow$ sample from $\mathcal{U}[0, 1]$
    $mean\_gens \leftarrow$ TRAIN_NEAT(a)
    Add element $(\{a\}, mean\_gens)$ to $\mathcal{D}_{1R}$
    $i \leftarrow i + 1$
end while

In the algorithm above, twelve iterations of random search hyperparameter tuning are performed in order to find a suitable value for the NEAT hyperparameter *enabled_mutate_rate*. The mean number of generations over ten full runs with the same value for the NEAT hyperparameter is used as the metric by which to judge the effectiveness of a hyperparameter value.

Algorithm 5 A single Trial to tune the NEAT hyperparameter, *enabled_mutate_rate*, using BO.

$Par \leftarrow \{[0, 1]\}$, the 1d NEAT hyperparameter search space
$GP \leftarrow$ GP prior over Par
$i \leftarrow 1$
$\mathcal{D}_{1BO} \leftarrow \varnothing$
while $|\mathcal{D}_{1BO}| < 2$ and $i \leq 2$ **do**
    $a \leftarrow$ sample from $\mathcal{U}[0, 1]$
    $mean\_gens \leftarrow$ TRAIN_NEAT(a)
    Add element $(\{a\}, mean\_gens)$ to $\mathcal{D}_{1BO}$
    $i \leftarrow i + 1$
end while
Condition GP on $\mathcal{D}_{1BO}$
Compute $\alpha_2^{\text{EI}}(Par)$
$k = 1$
while $k \leq 10$ **do**
    Compute $\alpha_i^{\text{EI}}(x)$
    $\{a, b, c, d\}_\alpha \leftarrow \text{argmax}_{x \in \mathbf{X}} \alpha_i^{\text{EI}}(Par)$
    $mean\_gens \leftarrow$ TRAIN_NEAT($\{a, b, c, d\}_\alpha$)
    Add element $(\{a\}_\alpha, mean\_gens))$ to $\mathcal{D}_{2BO}$
    Condition GP on $\mathcal{D}_{2BO}$
    $i \leftarrow i + 1$
end while

In the algorithm above, twelve iterations of Bayesian Optimisation hyperparameter tuning are performed in order to find a suitable value for the NEAT hyperparameter *enabled_mutate_rate*. as before, the mean number of generations over ten full runs with the same value for the NEAT

hyperparameter is used as the metric by which to judge the effectiveness of a hyperparameter value.

9.2 Pseudocode for Experiment 2

The algorithms in this section, the algorithms for experiment 2, are similar to the algorithms for experiment 1 above, except that instead of tuning a single NEAT hyperparameter, four NEAT hyperparameters - namely $prob\_add\_connection, prob\_delete\_connection, prob\_add\_node, prob\_delete\_node$ - are simultaneously tuned.

Algorithm 6 The Procedure (function) used to evaluate a set of NEAT hyperparameters for the CartPole RL problem.

procedure TRAIN_NEAT(a, b, c, d)
    $i \leftarrow 0$
    $L_g \leftarrow$ *empty list*
    for $i < 50$ **do**
        $P \leftarrow NEAT\ Population$
        $P.prob\_add\_connection \leftarrow a$
        $P.prob\_delete\_connection \leftarrow b$
        $P.prob\_add\_node \leftarrow c$
        $P.prob\_delete\_node \leftarrow d$
        $T \leftarrow CartPole\ MDP$
        Evolve P until T is solved
        $g_i \leftarrow$ Number of P generations until T was solved
        $L_g.insert(g_i)$
        $i \leftarrow i + 1$
    end for
    $mean\_gens \leftarrow \text{mean}(L_g)$
    Return $mean\_gens$
end procedure

Algorithm 7 A single Trial to tune NEAT hyperparameters using BO.

$Par \leftarrow \{[0,1],[0,1],[0,1],[0,1]\}$, NEAT hyperparameter search space
$GP \leftarrow$ GP prior over Par
$i \leftarrow 1$
$\mathcal{D}_{2BO} \leftarrow \varnothing$
while $|\mathcal{D}_{2BO}| < 10$ and $i \leq 10$ **do**
    $a \leftarrow$ sample from $\mathcal{U}[0,1]$
    $b \leftarrow$ sample from $\mathcal{U}[0,1]$
    $c \leftarrow$ sample from $\mathcal{U}[0,1]$
    $d \leftarrow$ sample from $\mathcal{U}[0,1]$
    $mean\_gens \leftarrow$ TRAIN_NEAT(a,b,c,d)
    Add element $(\{a,b,c,d\}, mean\_gens)$ to $\mathcal{D}_{2BO}$
    $i \leftarrow i+1$
end while
Condition GP on $\mathcal{D}_{2BO}$
Compute $\alpha_{10}^{\mathrm{EI}}(Par)$
$k = 0$
while $k < 250$ **do**
    Compute $\alpha_i^{\mathrm{EI}}(x)$
    $\{a,b,c,d\}_\alpha \leftarrow \mathrm{argmax}_{x \in \mathbf{X}}\, \alpha_i^{\mathrm{EI}}(Par)$
    $mean\_gens \leftarrow$ TRAIN_NEAT$(\{a,b,c,d\}_\alpha)$
    Add element $(\{a,b,c,d\}_\alpha, mean\_gens))$ to $\mathcal{D}_{2BO}$
    Condition GP on $\mathcal{D}_{2BO}$
    $i \leftarrow i+1$
end while

9.3 Random

Algorithm 8 A single Trial to tune NEAT hyperparameters using Random Search.

$\mathcal{D}_{2R} \leftarrow \varnothing$
$i \leftarrow 1$
while $|\mathcal{D}_{2R}| < 110$ and $i \leq 110$ **do**
    $a \leftarrow$ sample from $\mathcal{U}[0, 1]$
    $b \leftarrow$ sample from $\mathcal{U}[0, 1]$
    $c \leftarrow$ sample from $\mathcal{U}[0, 1]$
    $d \leftarrow$ sample from $\mathcal{U}[0, 1]$
    $mean\_gens \leftarrow$ TRAIN_NEAT(a, b, c, d)
    Add element $(\{a, b, c, d\}, mean\_gens)$ to $\mathcal{D}_{2R}$
    $i \leftarrow i + 1$
end while

9.4 Pseudocode for Experiment 3

The purpose of the algorithms in this section is to evaluate the effectiveness of NEAT hyperparameter value sets, found in the previous experiment when they are used on a new unseen RL task.

In the algorithm below, the eight hyperparameter value sets found in experiment 2 are used as the hyperparameters sets to initialise eight NEAT populations with the goal of solving the current task at hand, namely the Lunar Lander MDP.

Algorithm 9 Evaluating the eight NEAT hyperparameter sets found in experiment 2 on a new MDP, namely Lunar Lander.

$\mathcal{D}_3 \leftarrow \varnothing$
$Hyp \leftarrow$ List of all found sets of hyperparameters from experiment 2
for each $\{a, b, c, d\}$ in Hyp **do**
    $P \leftarrow NEAT\ Population$
    $P.prob\_add\_connection \leftarrow a$
    $P.prob\_delete\_connection \leftarrow b$
    $P.prob\_add\_node \leftarrow c$
    $P.prob\_delete\_node \leftarrow d$
    $T \leftarrow Lunar\ Lander\ MDP$
    Evolve P until T is solved
    $mean\_gens \leftarrow$ Number of P generations until T was solved
    Add element $(\{a, b, c, d\}_\alpha, mean\_gens))$ to $\mathcal{D}_3$
end for

In the algorithm below, eight NEAT hyperparameter value sets are randomly generated and evaluated. This is effectively a random search, but only for eight iterations, which is not enough iterations to yield a result. The eight NEAT hyperparameter sets are generated to serve as a benchmark to measure the performance of the meta-transfer learning against.

Algorithm 10 Evaluating eight randomly generated NEAT hyperparameter sets on the Lunar Lander MDP.

$\mathcal{D}_{3R} \leftarrow \varnothing$
$i \leftarrow 1$
while $i \leq 8$ **do**
    $P \leftarrow NEAT\ Population$
    $a \leftarrow$ sample from $\mathcal{U}[0, 1]$
    $b \leftarrow$ sample from $\mathcal{U}[0, 1]$
    $c \leftarrow$ sample from $\mathcal{U}[0, 1]$
    $d \leftarrow$ sample from $\mathcal{U}[0, 1]$
    $T \leftarrow Lunar\ Lander\ MDP$
    Evolve P until T is solved
    $mean\_gens \leftarrow$ Number of P generations until T was solved
    Add element $(\{a, b, c, d\}_\alpha, mean\_gens))$ to $\mathcal{D}_{3R}$
end while

Chapter 10

Bibliography

Bibliography

[1] A. Samuel. *Some Studies in Machine Learning Using the Game of Checkers.* IBM Journal of Research and Development, Vol.3, 210 - 229, 1959.

[2] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* The MIT Press, Cambridge, Massachusetts, Vol.2, 2018.

[3] K. Wang, X. Gao, Y. Zhao, X. Li, D. Dou, C. Xu. *Pay Attention to Features, Transfer Learn Faster CNNs.* International Conference on Learning Representations (ICLR), 2020

[4] M. Krasser. Bayesian machine learning notebooks. Webpage, updated 2020. `https://github.com/krasserm/bayesian-machine-learning`

[5] P. Fraziern. *A Tutorial on Bayesian Optimization.* arXiv: 1807.02811, 2018.

[6] M. Ebden. *Gaussian Processes for Regression: A Quick Introduction.* arXiv: 1505.02965v2, 2015.

[7] C. Rasmussen and C. Williams. *Gaussian Processes for Machine Learning.* Massachusetts Institute of Technology.MIT Press, ISBN 026218253X, 1 - 32, 2006.

[8] R. Richardson and M. Osborne and D. Howey. *Gaussian process regression for forecasting battery state of health.* Addison-Wesley, Reading, Massachusetts, 1993. Journal of Power Sources, 357: 209 - 219, 2017.

[9] R. Ott and M. Longnecker. An Introduction to Statistical Methods and Data Analysis 7th Edition, Cengage Learning, 2015

[10] V. Nguyen, S. Gupta, S. Rana, C. Li, and S. Venkatesh. *Regret expected improvement over the best-observed value and stopping condition.* Asian Conference on Machine Learning, 279–294, 2017.

[11] R. Lorenz, R. Monti, I. Violante, A. Faisal, C. Anagnostopoulos, R. Leech, and G. Montana. *Stopping criteria for boosting automatic experimental design using real-time fmri with bayesian optimization.* arXiv:1511.07827, 2015.

[12] K. Stanley and R. Miikkulainen. *Evolving Neural Networks through Augmenting Topologies* . Evolutionary Computation vol.10(2), 99-127, 2002.

[13] K. Stanley and R. Miikkulainen *Efficient Evolution of Neural Network Topologies.* Proceedings of the 2002 Congress on Evolutionary Computation vol.2, 1757-1762, 2002.

[14] I. Goodfellow, F. Bach, Y. Bengio and A. Courville. *Deep Learning (Adaptive Computation and Machine Learning series).* 2020

[15] G. Cybenko *Approximation by superpositions of a sigmoidal function.* Mathematics of Control, Signals and Systems vol.2, 303–314, 1989.

[16] B. Hanin *Universal Function Approximation by Deep Neural Nets with Bounded Width and ReLU Activations.* 2019.

[17] K. Funahashi. *On the approximate realization of continuous mappings by neural networks.* Neural Networks vol.2, 183–192, 1989.

[18] K. Hornik, M. Stinchcombe and H. White. *Multi-layer feedforward networks are universal approximators.* Neural Networks vol.2, 359–366, 1989.

[19] H. Wang and B. Raj *On the Origin of Deep Learning.* 2017.

[20] D. Kingma and J. Ba. *ADAM: A Method for Stochastic Optimization.* 2015

[21] D. Montana, and L. Davis. *Training feedforward neural networks using genetic algorithms.* Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 762–767, 1989

[22] J. Koza, F. Bennett III and D. Andre. *Automated Design Of Both The Topology And Sizing Of Analog Electrical Circuits Using Genetic Programming.* 1998.

[23] K. Stanley, D. D'Ambrosio, and J. Gauci. *A hypercubebased encoding for evolving large-scale neural networks.* Artificial life 15, Vol.2, 185–212, 2009

[24] R. Louis, and D. Yu. *A study of the exploration/exploitation trade-off in reinforcement learning Applied to autonomous driving.* 2019

[25] M. Hausknecht, J. Lehman, R. Miikkulainen, and P. Stone. *A NeuroEvolution Approach to General Atari Game Playing.* 2014

[26] A. Zheng. *Evaluating Machine Learning Models*. O'Reilly Media, Inc. 2015.

[27] E. Lin, Q. Chen, X. Qi. *Deep Reinforcement Learning for Imbalanced Classification.* 2019

[28] I. Omelianenko *Hands-On NeuroEvolution with Python*. Packt Publishing, 2019.

[29] Open AI Lunar Lander-V2 Webpage, updated 2020. `https://gym.openai.com/envs/LunarLander-v2/`

[30] A. Barto, R. Sutton and C. Anderson, *Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problem*. IEEE Transactions on Systems, Man, and Cybernetics, 1983.

[31] Open AI CartPole-v0 webpage, updated 2020. `https://gym.openai.com/envs/CartPole-v0/`

[32] Open AI Lunar Lander-V2 Github Webpage, updated 2020. `https://github.com/openai/gym/blob/master/gym/envs/box2d/lunar_lander.py`

[33] E. Garrido-Merchan and D. hernandez-Lobato. *Dealing with Integer-valued Variables in Bayesian Optimization with Gaussian Processes*. 2017.