Neuroevolution of Bayesian Neural Networks

Alberto Castelo Becerra

Master of Science in Advanced Artificial Intelligence

Universidad Nacional de Educación a Distancia (UNED)

June 2020

Supervisor:
Enrique J. Carmona Suárez

Table of Contents

List of Figures

List of Tables

Acknowledgment

First, I would like to express my gratitude to my supervisor Dr. Enrique J. Carmona Suárez for his support during the development of this Master Thesis. You provided valuable insight and help me push further on this topic.

I would like to thank everyone that I came across in my life and that helped me grow. I am specially grateful to my closest colleagues in University of Vigo, ADAC Lab at North Carolina State University, Gradiant and Nextail Labs. Sharing and discussing ideas with you was and still is a pleasure.

Then, I am grateful to my family: my sister, my parents and my grand-parents. I am specially grateful to my sister and father. To my sister because you are always there when needed despite being far away. To my father because I mostly learned from you the value of hard work. I can still remember you studying to get your engineering degree when I was kid. That was a recurrent memory while growing up and still is.

Finally, I am grateful to Lisa. My partner these years and I could not have asked for a better one. You made handling both working one thousand hours at a startup and working on this Masters really easy.

Abstract

Bayesian Neural Networks have been getting a lot of attention from the research community because they show promise on better handling uncertainty. This is specially relevant in applications where a false positive or negative can have a big impact such as in self-driving cars or medical diagnosis. However, they do not typically achieve the performance metrics of traditional Neural Networks as they still have many problems. Some of these problems are the issues that arise with the extra complexity of training a network that has extra parameters to model uncertainty. In this Master Thesis, we design, implement and validate a new algorithm called Bayesian-NEAT to both evolve architecture and parameters for feed-forward Bayesian Neural Networks. The algorithm is based on a previous evolutionary algorithm called NEAT (Neuroevolution of Augmenting Topologies) that is successfully applied to standard Neural Networks. The main contributions of this work are: (1) design and validation of Bayesian-NEAT as an approach for solving Supervised Learning problems of medium dimensionality and providing uncertainty estimates for individual predictions; (2) validate Bayesian-NEAT as highly robust algorithm against mislabeled data; and (3) design and implement first NEAT-based implementation of a feed-forward network in a Deep Learning framework.

Keywords

NEAT, Neural Architecture Search, Bayesian Neural Networks, Supervised Learning

Resumen

Las Redes Neuronales Bayesianas han atraído mucha atención últimamente por parte de la comunidad investigadora porque muestran resultados prometedores en la gestión de la incertidumbre en modelos predictivos. Esta gestión es especialmente relevante en aplicaciones donde un falso positivo o negativo pueda tener un gran impacto como en el caso de los vehículos autónomos o en la diagnosis médica. Sin embargo, este tipo de redes todavía no alcanzan el nivel de las métricas de las Redes Neuronales tradicionales. Alguna de las causas es la complejidad añadida al entrenar redes que tienen parámetros extra para modelar la incertidumbre. En esta Tesina, diseñamos, implementamos y validamos un nuevo algoritmo llamado NEAT Bayesiano para evolucionar tanto la arquitectura como los pesos en Redes Neuronales Bayesianas "de propogación hacia delante". El algoritmo está basado en un algoritmo evolutivo previo llamado NEAT (por sus siglas en Inglés, Neuroevolución de topologías en constante aumento) que ha sido aplicado con éxito a Redes Neuronales más tradicionales. Las contribuciones principales de este trabajo son: (1) el diseño y validación del NEAT Bayesiano como una aproximación para resolver problemas de Aprendizaje Supervisado en problemas de media dimensionalidad al mismo tiempo que se proporciona una estimación de la incertidumbre para cada ejemplo; (2) la validación del NEAT Bayesiano como un algoritmo altamente robusto contra conjuntos de datos con muchos ejemplos mal etiquetados; y (3) el diseño e implementación del primer algoritmo basado en NEAT para redes "de propogación hacia delante" en una librería especializada de Aprendizaje Profundo.

Palabras Clave

NEAT, Búsqueda de Arquitecturas de Neuronas, Redes Neuronales Bayesianas, Aprendizaje Supervisado

Chapter 1

Introduction

1.1 Context

Neural Networks have been achieving State-of-the-Art results for a decade now at a variety of problems, ranging from computer vision, text and speech processing (13). However, they still have some issues that do not allow its usage for all the use-cases where they could be applied. Some of the main issues related to Neural Networks are:

1. They need a large amount of data or examples to perform well. This requires a big effort to label every example, typically requiring an expert human to do it.

2. They are usually consumed as black-box models. This means that we cannot understand how the model generated a prediction or result. Furthermore, users do not know if they can trust the model for a specific input (15). That is, we do not know the uncertainty behind a prediction or result. In applications such as medical diagnosis or autonomous driving is important to understand when we can trust the autonomous system.

3. Architectures need to be manually crafted. Most interesting architectures (AlexNet, ResNet, etc.) have been manually crafted, creating a big overhead

during model training.

In this regard, Bayesian Neural Networks models are proposed in the literature as an alternative to standard Neural Networks to deal with the two first issues (17). First, standard Bayesian Models do not need a large amount of data to perform well. Second, instead of only having a point-estimate, Bayesian Models actually provide a probability distribution as result. This allows for further interpretability of the model as well as uncertainty estimation. However, they still have the problem that the model's architecture has to still be manually crafted.

Regarding the third issue mentioned above there is a long series of literature that addresses the topic of Neural Architecture Search (NAS), using approaches random from Bayesian Optimization, Random Search, and population-based Evolutionary Algorithms (8). However, most of those methods are applied to traditional neural networks. In this work, we propose an Evolutionary Algorithm, based in Neuroevolution of Augmenting Topologies (NEAT) (30), to address the issue of generating Bayesian Neural Networks that can solve Supervised Learning tasks.

1.2 Hypothesis and Objectives

In the previous section we discuss the importance of Bayesian Neural Networks and the problem of architecture search. In this section, we explain the hypothesis that motivates this work as well as we set research objectives that allow us to accept the hypothesis.

The hypothesis that motivates this work is: *"We can apply Evolutionary Algorithms to generate architectures for Bayesian Neural Networks that obtain competitive results in Supervised Learning tasks in a consistent manner while providing a measure of uncertainty about each prediction."*.

Based on this hypothesis we define the next objectives:

1. Design and implement a version of NEAT that can evolve Bayesian Neural Networks.
2. Design and implement a software system that facilitates research and experimentation.
3. Define supervised learning tasks and datasets to be used in experiments.
4. Design and implement benchmark methods for the proposed solution.
5. Validate the proposed algorithm in different datasets against benchmark methods.
6. Study an application where the proposed approach is obtains better results than benchmarks.

Finally, it is also worth discussing the limitations of this work. Despite Deep Neural Networks are the ones achieving State-of-the-Art results in many tasks, the study is limited to small and medium size neural networks due to the amount of computing available.

1.3 Structure

In this chapter we have provided context that motivates this work, stated the main hypothesis and define the objectives set to validate the hypothesis. Now we discuss how the rest of the document is structured.

Chapter 2 provides the background on the main concepts discussed in this work. The main topics in this chapter are background on Neural Networks, Bayesian Models, Neural Architecture search as well as Evolutionary Algorithms.

Chapter 3 discusses the proposed solution. This chapter is divided into two main subsections. While the first one deals with the software solution, the second one discusses the actual algorithm.

Chapter 4 contains the experiments, their most interesting results and a discussion on them. In this section, we discuss the algorithm's parametrization, benchmarking methods, datasets and how experiments were designed to validate the research hypothesis.

Chapter 5 discusses the conclusions and possible future research lines.

Chapter 2

Background

2.1 Neural Networks

A Neural Network is a graphical model build by composing different functions across different layers. These models, specifically, the deepest ones have revolutionized the research community and industry by being able to obtain great results in different domains such as Computer Vision, Natural Language Processing or Speech Recognition, among others. However, they still have some issues:

1. They tend to overfit the data. The research community has introduced several techniques to increase generalization such as Dropout, Stochastic Gradient Descent.

2. They need a lot of data in order to get high performance.

3. They are not interpretable nor explainable. It is not possible to know when they are sure about some prediction or when they are not.

The graphical structure of a neuron is shown in Figure 2.1 while the function mapping of a neuron is given by Equation 2.1. Some nomenclature:

1. $x_{k,i}$: It is the output of Neuron k at Layer i.

2. $Act(x)$: It is the activation function in a Neuron. Typical activation functions are: $Sigmoid(x)$, $Tanh(x)$, $RELU$, etc.

3. $w^i_{j,k}$: It is the connection weight between Neuron j of Layer $i-1$ and Neuron k of Layer i.

4. $b_{k,i}$: It is bias associated with Neuron k of Layer i.

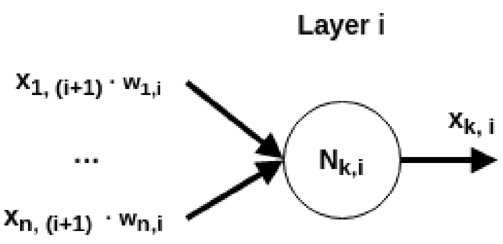


Figure 2.1: *Node in Neural Network.*

$$\begin{aligned} x_{k,i} = Act(\sum_{j}^{n} x_{j,i+1} \cdot w^i_{j,k} + b_{k,i}) \\ w_{j,i} \in \Re \\ b_{k,i} \in \Re \end{aligned} \tag{2.1}$$

Then, when a set of Neurons is connected, they form a Neural Network as shown in Figure 2.2. By composing each neuron's Equation 2.1, we can map each input $X = (x_1, ..., x_n)$ to an output $Y = (y_1, ..., y_m)$.

Typical applications where Neural Networks are used imply learning some data representation so that, when new data is presented to the input X, we can predict an output Y. In these cases, considering that the network structure or architecture is fixed and that we have selected an Activation function, we only need to know the Network's Weights and the Neuron Biases. These are the parameters that are learned during the training process of a Neural Network and they are real-valued numbers.

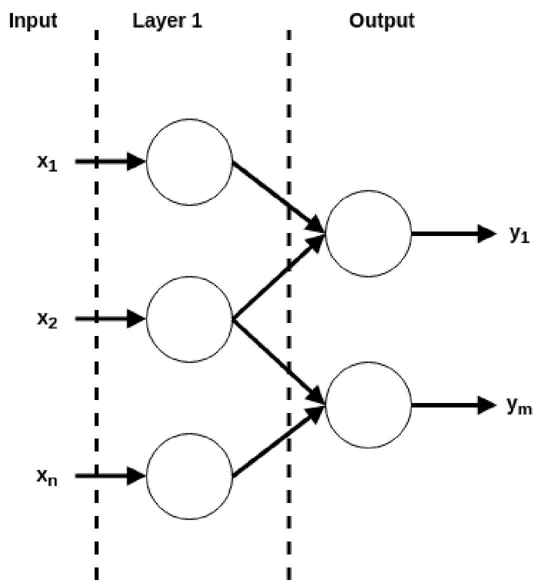


Figure 2.2: *Feed Forward Neural Network.*

2.2 Background on Bayesian Methods

In this section, we give a general overview of Bayesian Methods discussing how they work and how models learn. At the end, we provide an introduction to the specific learning strategy that is used in this work.

2.2.1 Introduction

Bayesian methods are called that way because they are based in Bayes Rule (see Equation 1).

$$p(\theta|D) = \frac{p(D|\theta) \cdot p(\theta)}{p(D)} = \frac{p(D|\theta) \cdot p(\theta)}{\int p(D|\theta) \cdot p(\theta) \cdot d\theta} \tag{1}$$

In such equation, θ refers to the model's parameters, while D refers to the observed variables or data. For instance, in a supervised learning problem, D relates to both the independent variable x and the dependent variable y. It is interesting to discuss what these terms mean (2):

1. $p(\theta|D)$: usually known as the Posterior, it is the conditional probability distribution of the parameters given what we know or observe, the data D.
2. $p(D|\theta)$: this component is known as the Likelihood probability distribution and is the conditional probability distribution of the data given the parameters.
3. $p(\theta)$: commonly known as model Priors. This probability distributions allow to inject expert knowledge into the model.
4. $p(D) = \int p(D|\theta) \cdot p(\theta) \cdot d\theta$: this probability distribution is commonly known as Evidence and is the probability of observing all combinations of input/output data (D) for all parameter configuration of θ. This term is typically intractable for problems of medium or large size and is the main obstacle towards being able to correctly apply Bayesian methods.

Then, Bayesian methods have four steps:

1. Modeling each term of the equation. Following Equation 1, we only need to define the Likelihood and the Priors because the denominator or Evidence can be extracted from those.
   (a) Modeling Likelihood ($p(D|\theta)$). The Likelihood can be any known distribution (Poisson, Binomial, etc.) with parameters θ that is acting as the generative process of our data. The Likelihood is not only linked to defined probability distributions and it can be as complex as we want. For instance, it could be modeled as a Probabilistic Neural Network with some weights and biases parameters.
   (b) Modeling Priors ($p(\theta)$). The Prior can be as well any kind of probability distribution, representing our initial belief on the values of θ. Typical distributions used are Gaussian, Exponential, etc.
2. Inference. This is the process of finding the Posterior $p(\theta|D)$. In order to solve this, we need to find the intractable denominator in Equation 1. Knowing the posterior means knowing the parameters of our model that will allow

us to make predictions on unseen data. Therefore, the Inference problem is equivalent to the Training problem in standard Machine Learning or Data Mining algorithms.

3. Validation of the model. That is making sure the model is accurate in terms of its posterior distribution.

4. Prediction. This is the final process when we are trying to use the inferred model to provide predictive distributions of the dependent variable $y*$ for a new unseen example with independent variables $x*$, given old data and the model parameters. This is commonly known as the Predictive Distribution or $p(y*|x*, D, \theta)$.

The second step of the process is typically the hardest and most time-consuming in Bayesian methods. In literature, we typically find three main approaches to solve the Inference Problem:

1. Directly solve the equation. Sometimes this is possible when the probability distributions used to model the problem have good mathematical properties. For instance, a Gaussian likelihood with Gaussian prior will return a Gaussian Posterior, simplifying the problem to just finding the two parameters that define a Gaussian distribution. Another occasion to directly calculate the posterior is when we have a very small search space where you can evaluate all the possible combinations of parameters.

2. Markov-Chain Monte-Carlo (MCMC) or Exact Inference methods (25). MCMC-based methods try to find the Posterior by defining a Markov-Chain of the generative process and sampling from it. The main advantage of this method is that if sampled enough times, it will converge to actual Posterior distribution. The problem is that sampling is slow, and because it relays on a Markov Chain, the process cannot be parallelized.

3. Variational Inference (VI) or Approximate Inference (4). VI methods treat inference as an optimization problem. The advantage of this method is that it

is much more scalable than MCMC and can train with much more data in a distributed fashion. The downside comes that we are actually approximating the posterior, not getting the exact one.

2.2.2 Variational Inference as Optimization

Variational Inference (VI) is an approximate bayesian inference technique (3; 4). Opposite to Monte-Carlo sampling techniques, VI treats the problem of inference as an optimization problem. That is, given a candidate Approximate Posterior Distribution $q_\phi(\theta)$ (often called Variational Distribution), its parameters ϕ are modified so that a divergence measure between both probability distributions (the real posterior and the approximation) is minimized as shown in Equation 2.2.

$$\min_{\phi} \quad D(q_\phi(\theta)||p(\theta|D)) \tag{2.2}$$

To make this problem tractable there are two important considerations: the equation used to measure divergence between both distributions and how complex is the defined Variational Distribution ($q_\phi(\theta)$). Regarding the selection of the divergence, the Kullback-Leibler (KL) Divergence is the one applied with most success because of its adequate mathematical properties. Equation 2.3 shows the KL Divergence definition. The KL divergence is an information theory measure of proximity between two distributions that is minimized when both distributions are equal. Despite being non-negative, it cannot be called a metric because it is asymmetric ($D_{KL}(q||p) \neq D_{KL}(p||q)$).

$$D_{KL}(q_\phi(\theta)||p(\theta|D)) = \int q_\phi(\theta) \cdot \log \frac{q_\phi(\theta)}{p(\theta|D)} \cdot d\theta \tag{2.3}$$

If we pay close attention to Equation 2.2, we can see that we are trying to find the parameters ϕ for which the KL Divergence is minimized. However, the whole purpose of the VI problem is to find the Posterior Distribution $p(\theta|D)$). How can we optimize ϕ if we cannot compute part of the cost function ($p(\theta|D)$))? This problem is solved by the definition of the KL Divergence. Bishop showed, applying Bayes

Rule to Equation 2.3, that the KL Divergence between these two distributions can be calculated in terms of the prior $p(\theta)$ and the log-likelihood $log(p(D|\theta))$, completely avoiding the intractable computation of the Evidence component $(p(D))$. This is shown in Equation 2.4.

$$
\begin{aligned}
D_{KL}(q_\phi(\theta)||p(\theta|D)) \propto \int q_\phi(\theta) \cdot \log \frac{q_\phi(\theta)}{p(\theta) \cdot p(D|\theta)} \cdot d\theta = \\
D_{KL}(q_\phi(\theta)||p(\theta)) - E_{q_\phi(\theta)}[\log(p(D|\theta)])
\end{aligned}
\tag{2.4}
$$

2.3 Bayesian Neural Networks

In this section, we introduce Bayesian Neural Networks and derive the cost functions that are used later in the proposed algorithm.

2.3.1 Introduction

Bayesian Neural Networks (BNN) are a special case of standard Neural Networks where each parameter follows a probability distribution (5; 22; 24). BNNs typically assume a Gaussian distribution for its parameters because great mathematical properties. In our study, we do it as well, because, as we discuss later, it allow us to increase the algorithm performance. Therefore, Equation 2.1 is transformed into Equation 2.5.

$$
\begin{aligned}
x_{k,i} = Act(\sum_{j}^{n} x_{j,i+1} \cdot w^i_{j,k} + b_{k,i}) \\
w^i_{j,k} \sim \mathcal{N}(\mu^i_{j,k}, (\sigma^i_{j,k})^2) \\
b_{k,i} \sim \mathcal{N}(\mu_{k,i}, \sigma^2_{k,i})
\end{aligned}
\tag{2.5}
$$

Bayesian Neural Networks have been receiving a lot of attention recently because they get the expressiveness of standard Neural Networks, allowing them to model more complex phenomena while bringing some of the features of traditional Bayesian methods. However, they still have some known problems (6; 11; 18):

1. BNN typically underperform on classification/regression metrics, especially in deeper models.

2. Most of the practitioner's tricks that are available for standard Deep Learning have not been clearly defined for BNNs.

3. Defining good priors is not solved. Bayesian methods highly relied on prior beliefs but there is not a clear strategy on how to define those priors in BNN yet. The typical approach is to just use a normal distribution.

4. How to define stochasticity level. There is not a clearly strategy on how to do this either despite there are some typical approaches:

   (a) Use a single σ or standard deviation for each parameter. This solution doubles the number of parameters of our model, and, therefore, it increases the complexity of the training process. This is the approach used in this work because of the nature of the proposed approach.

   (b) Consider the same σ for all the neurons in the same layer (5).

   (c) Hybrid networks. Assume that the layers closest to the input are deterministic while the ones closest to the output are stochastic (parameters follow a probability distribution). In this regard, some authors place a Gaussian Process at the end of the model to account for the uncertainty.

2.3.2 Variational Inference in BNN

Recalling from Equation 2.4, the KL Divergence between the Variation Distribution and the Posterior is composed of two terms:

1. The expected log-likelihood $E_{q_\phi(\theta)}[\log(p(D|\theta)]$.

2. The KL Divergence between the Variational Distribution and the Priors $D_{KL}(q_\phi(\theta)||p(\theta))$

While the first term depends on the problem at hand, the second term only depends on the structure of the network and how many parameters it has. That is why the second term is usually called the Complexity Cost of the model. The complexity cost for a Bayesian Neural Network is shown in Equation 2.6 as the sum of each KL Divergence between each weight or bias i ($\mathcal{N}(\mu_i, \sigma_i^2)$) and its prior distribution belief ($\mathcal{N}(\mu_P, \sigma_P^2)$).

$$D_{KL}(q_\phi(\theta)||p(\theta)) = \sum_{i=1}^{N} D_{KL}(\mathcal{N}(\mu_i, \sigma_i^2)||\mathcal{N}(\mu_P, \sigma_P^2)) \tag{2.6}$$

On the other hand, the log-likelihood term depends on the problem at hand. As in this work, we handle both Classification and Regression tasks, we derive a cost function for each of these problems.

In the case of a classification problem, we can assume that the likelihood of expecting a Label follows a Bernoulli probability distribution, $p(D|\theta) \sim Bernouilli(p)$. Then, through a series of transformations, it can be derived that minimizing the log-likelihood is equivalent to minimizing the sum of the Cross-Entropy for all the possible labels. Equation 2.7 shows the Cross Entropy for a single label given the output of the Neural Network $f(X_i, \theta)$ for a single data point i.

$$CrossEntropy(y_i^c|X_i, \theta) = -(y_i \cdot log(f(X_i, \theta)) + (1 - y_i) \cdot log(1 - f(X_i, \theta))) \tag{2.7}$$

Then the log-likelihood for a classification problem can be expressed as in Equation 2.8, where C is the number of classes and N is the number of data points.

$$E_{q_\phi(\theta)}[\log(p(D|\theta)]) \propto \frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} CrossEntropy(y_i^c|X_i, \theta) \tag{2.8}$$

In the case of regression, we can assume that the likelihood $P(D|\theta)$ follows a Gaussian distribution. If that is the case, then, the log-likelihood is proportional to the Mean Square Error as shown in Equation 2.9

$$E_{q_\phi(\theta)}[\log(p(D|\theta)]) \propto \frac{1}{N}\sum_{i=1}^{N}(Y_i - f(X_i,\theta))^2 \tag{2.9}$$

2.4 Background on Neuro-Evolution

This section starts with a general description on Neural Architecture Search methods, then a discussion on how evolutionary algorithms work and, finally, an overview on NEAT.

2.4.1 Background on Neural Architecture Search

Neural Architecture Search (NAS) is the process of automating Neural Networks' architecture design (8) and it is required stepping stone in the path to AutoML (16). AutoML looks for full automation of the Machine Learning process, from feature engineering to architecture design. In NAS, three attributes that allow classifying different algorithms:

1. Search space. The search space defines the set of networks that can be represented. While some approaches allow a level of granularity that goes to each connection, other approaches use higher level abstractions (building-blocks).

2. Search strategy. The search strategy defines how the search space is explored. In the literature, it is common to see four types of approaches:

    (a) Evolutionary Algorithms (30). This approach relies on evolving individuals of a population across generations. Each individual represents a network and they can reproduce and mutate as in nature.

    (b) Bayesian Optimization (1). In this problem, architecture search is considered as hyper-parameter optimization process where the sequence of parameters studied are defined in a bayesian manner. That is, previous executions are used as prior belief that help define a new search direction.

(c) Reinforcement Learning (8). In this case, the agent action is to generate neural networks getting rewards proportionally to the performance of the network in validation set.

3. Performance estimation strategy. This refers to the strategy used to measure the potential of each architecture. A Naive approach is typically to train and validate each of the architectures. However, this is computationally very expensive. Gaier et al (10) question the importance of the actual weight values compared to the architecture. For that they introduced Weight Agnostic Neural Networks (WANN), a search method based on NEAT that can perform some tasks without learning the weight and biases of the Neural Network. That is, is able to estimate performance without training the network.

As this work focus on the first of the discussed search strategies, below we discuss the principles used in Evolutionary Algorithms.

2.4.2 Background on Evolutionary Algorithms

Evolutionary Algorithms (EAs) are called that way because they are inspired by biology and, especially, by Darwinian evolution. As today's species evolved from very simple organisms, this branch of Artificial Intelligence leverage similar concepts to evolve individuals so that they are able to solve a problem (7). In EAs, solving a problem means finding the fittest individual. That is, the individual that has better adapted to its environment (problem) and has more chances to survive. The fitness can be modeled mathematically as the function that needs to be maximized. The evolutionary process is an iterative process and the one of generating individuals with better fitness. It usually depends on the following concepts:

1. Representation. It defines the search space or the set of possible individuals.

Moreover, EAs take nomenclature from biology as well. In this regard, it is worth to define several concepts that are used in this work:

  (a) Genome. It is the set of possible genes that are allowed by the selected representation.
  (b) Genotype or Chromosome. It is an instantiation of the genome. That is, an individual with specific genes' values.
  (c) Phenotype. It is an object in the context of the problem at hand (for instance, a Neural Network) and its complete definition is encoded in a Genotype. While the problem deals with Phenotypes, the EAs work with its Genotype counterpart. Therefore, a mapping between these two needs to be defined.

2. Fitness Function. It the maximization objective of the evolutionary search. The fittest individual is the one with the highest fitness value.
3. Population. It is the set of individuals alive at a specific generation or iteration.
4. Initialization. It is the process of randomly initializing a population at the beginning of the evolutionary process.
5. Recombination. It is the process of merging attributes or genes from parents. In this regard, there are many strategies on how to select parents to breed and on how to recombined those parents to create a new individual.
6. Mutation. It consists on randomly modifying some genes to create a new altered individual.
7. Survivor selection. At the end of each iteration, the algorithm decides what are the individuals that survive to the next generation or iteration.

In summary, there are two main processes in most EAs:

1. Generate candidate solutions. This stage focus on evolving new individuals through recombination or mutation.

2. Evaluate the fitness of each candidate. The fitness of each individual is calculated depending on the task at hand. For instance, it could be the accuracy in Supervised Learning problem or the rewards obtain performing a Reinforcement Learning task.

2.4.3 Neuro-Evolution of Augmenting Topologies

One of the main contributions to the field of Neuroevolution, which also serves as a starting point for the research in this work, comes from the Neuroevolution of Augmenting Topologies (NEAT) by Stanley et al (27; 30). NEAT introduces some innovative concepts to the search problem of finding both the right structure and weights for a Neural Network for a specific task. NEAT is able to solve problems by starting with a very simple structure and increasing complexity as needed. This process is known as complexification and it allows to reduce the dimensionality of the search space of the networks' weights. Previous solutions were not able to do both things at the same time. Either they optimized the weights of a given network or they created new structures whose optimal weights were found later on. So this became the most innovative contribution of NEAT. Despite almost two decades have already passed since the invention of NEAT, it is still an algorithm that is used currently as a Neuro-Evolution benchmark.

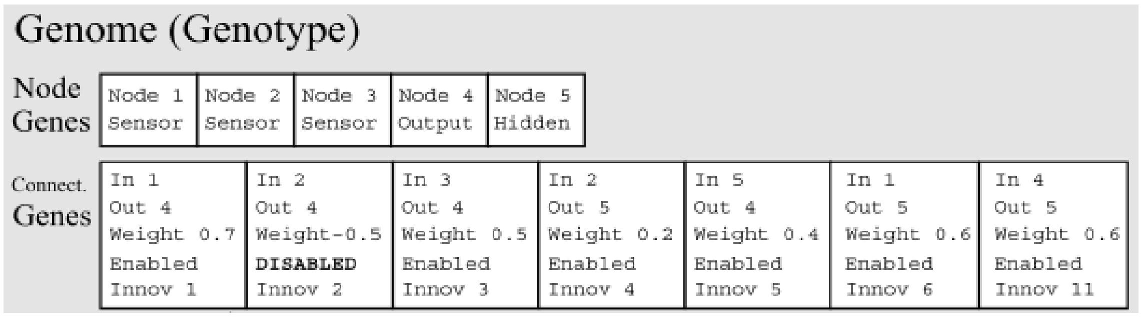
How does it work?

As any EA, NEAT also can be divided into two main phases: individuals' generation and individuals' fitness assessment. While the second phase depends more on the task at hand and its cost function, the first phase is worth explaining more in detail.

NEAT uses Direct Encoding to map individuals' genotype to the actual phenotype. The genotype of NEAT is shown in Figure 2.3. NEAT distinguishes two types of Genes:

1. Nodes: they can be input, hidden or output. Input and output nodes always exist in all networks and cannot be removed. Hidden and Output nodes contain parameters that can be evolved. In the original NEAT, the Nodes do not have any learned parameter.

2. Connections: they are defined by the Source and Destination Node. They only contain one evolving parameter, the weight of that connection. Furthermore, they can also be disabled so it is not considered in the actual network and have some *Innovation Number* that allows to correlate genes.



Genome (Genotype)

Node Genes

Node 1	Node 2	Node 3	Node 4	Node 5
Sensor	Sensor	Sensor	Output	Hidden

Connect. Genes

In 1	In 2	In 3	In 2	In 5	In 1	In 4
Out 4	Out 4	Out 4	Out 5	Out 4	Out 5	Out 5
Weight 0.7	Weight-0.5	Weight 0.5	Weight 0.2	Weight 0.4	Weight 0.6	Weight 0.6
Enabled	**DISABLED**	Enabled	Enabled	Enabled	Enabled	Enabled
Innov 1	Innov 2	Innov 3	Innov 4	Innov 5	Innov 6	Innov 11

Figure 2.3: *NEAT Genotype*

While the generative phase of NEAT operates with the Genotype, the evaluation phase works with the actual Neural Network that is produced, the Phenotype. Figure 2.4 shows the structure produced by the previous Genotype.

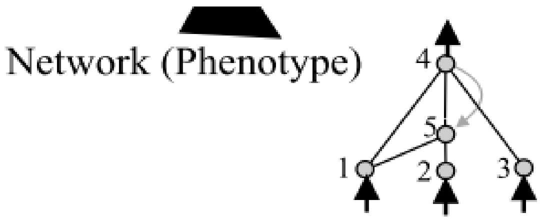


Figure 2.4: *NEAT Phenotype*

One of the most important evolution operators used in NEAT is mutation and it affects both genes' parameters (e.g.: weights) and structure. The mutation on numerical or categorical parameters is done as in any other evolutionary system:

1. Numerical parameters are modified using a random sample from a probability distribution.

2. Categorical parameters mutate by random selection from a pool of values.

Regarding structure mutation, the original NEAT work only proposes the use of mutation that adds complexity to the network: either add a new node or add a new connection.

While mutating only genes' parameters is straightforward as in any EA, mutating structure creates some extra problems when performing crossover among individuals. This is because different individuals can have very different structures and creating a new individual from two completely different structures typically translates into a new individual of poor quality or fitness. One of these problems that is well-known in the literature is the competing conventions problem. In order to solve this issue is why NEAT introduces *Historical Markings* or *Innovation Numbers*: to track genes through history and infer if individuals with similar genes can breed or not.

Even though NEAT provides a mechanism for crossover between individuals with different structure, there is another problem. More complex networks when mating will generate a network that is not completely fine-tuned for the problem. That is, it will start with a low fitness that in some cases will be lower than the fitness of less complex networks that have already evolved. As in evolution, only the strongest individuals survive, this means that the new networks will not be able to develop its full potential. In order to fix this issue, NEAT introduces another interesting mechanism called Speciation. This refers to grouping individuals into species based on how similar their genomes are. When a new specie is detected, it is allowed to evolve for some generations to see if their fitness increases. After those initial generations, it will compete against the others for survival.

Finally, future versions of NEAT (Hyper-NEAT) use Indirect Encoding to create bigger networks (28; 29). However, they are not addressed in this work.

Chapter 3

Proposed Solution

3.1 Software Solution

Given the research nature of this project, the software system built has that into consideration. Therefore, we set the following requirement: *The system should allow for fast prototyping and experimentation.* All software decisions were made taking this into account.

3.1.1 Software Development Patterns

In any software project it is important to follow good software development patterns because as the complexity grows, the project can become difficult to handle (20). In our case, the project takes advantage of the following:

1. Automatic Tests. Automatic tests are important because they programmatically define the behavior of a specific software module or submodule. Therefore, when something changes checking that system still works does not have any extra overhead for the developer. This is definitely a pattern that helps achieve faster experimentation as we can quickly see how changing a small piece affects the system. In this regard, this project implemented more than

100 automatic tests to ensure the algorithm's quality and robustness. Moreover, we can differentiate between two types of automatic tests in this project:

   (a) Unit Tests: used for validating the behavior of certain individual components of the system.

   (b) Integration Tests: used for validating the interaction between several components of the system.

2. Versioning Control. Being able to store the state of the code at some point allows to experiment faster as, in case of introducing any error, we can quickly go back to a previous stable version.

3. Modularity. Following the design principle of "Divide and Conquer", creating well isolated modules with individual responsibilities helps breakdown the complexity and ensure a robust software system.

4. Logging. Logs are an important part of any software system as they help to know what happened at some point.

3.1.2 Software Architecture

Despite Section 3.2 thoroughly explains the algorithm behind this work, there are a lot of other modules and components that were either developed or integrated to allow for fast prototyping and testing of the solution. Figure 3.1 shows a high level System's Diagram with the algorithm solution at the core of the system.

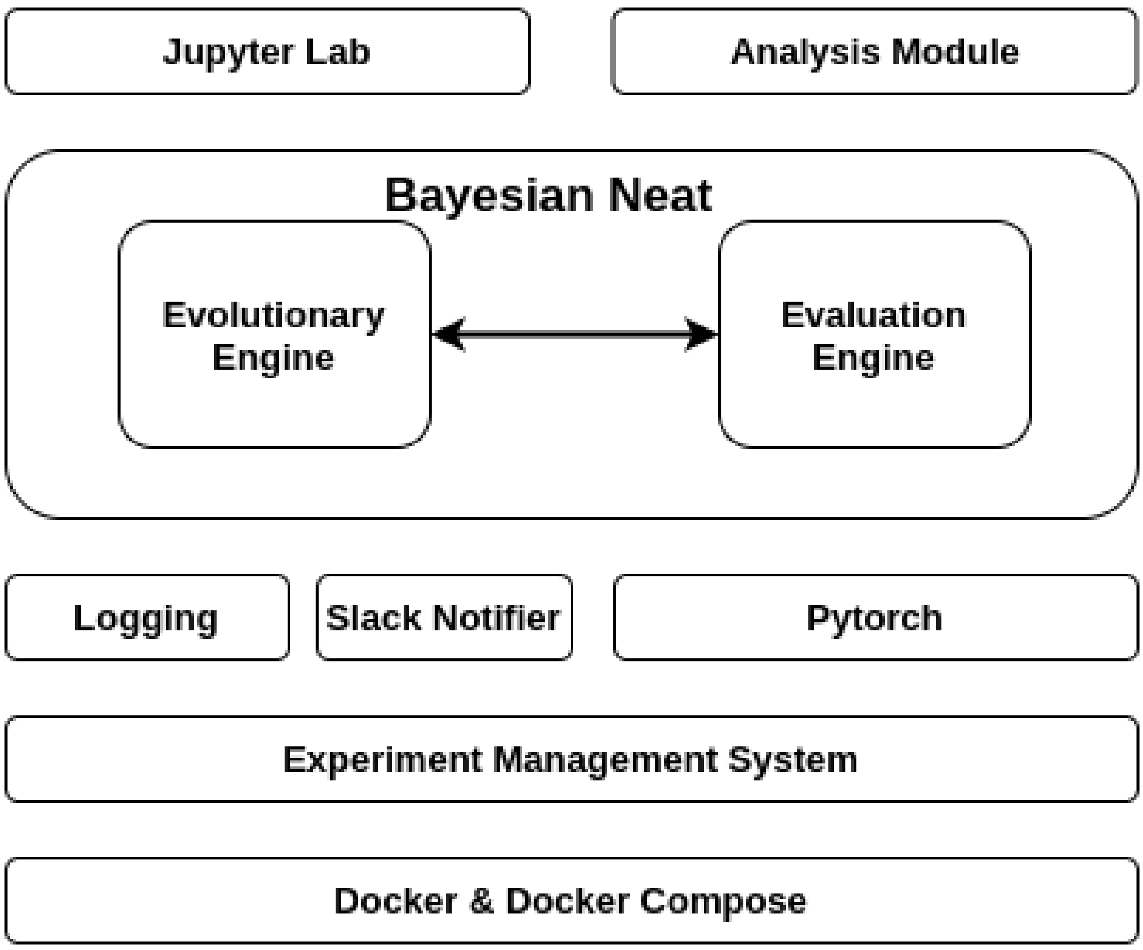


Figure 3.1: *System Architecture's Diagram.*

It is worth mentioning that all developments in this work were done in Python[1]. The reason is because of its library ecosystem (Numpy, Pandas, Pytorch, Numba, etc.) and because it is a language that is fast to write. The downside of this is that Python is a slow programming language even though. Starting from the bottom up of this diagram, next, we describe each of these subcomponents in detail.

[1]Python: https://www.python.org/

Docker Docker-Compose Orchestration

Docker[2] and Docker-Compose[3] are used to orchestrate the Bayesian-NEAT Research System. Docker is a technology that allows to create isolated environments (called containers) to run applications without interference from the underlying Operating System or other software dependencies installed on a computer. By defining a *Dockerfile* or specification of software dependencies to install, Docker ensures reproducibility of anything that runs inside of this container. Therefore, it is highly used for deploying applications to the cloud. In this project, there are two containers: the Bayesian-NEAT container that includes the code used to run the algorithm and the Minio container that is used as storage for experiment results. Since there are two containers, we use Docker-Compose to define how those two containers interact with each other and with the outside world. The diagram of how the system is orchestrated is shown in Figure 3.2. The shared volume block indicates that storage is shared between the host operating system and the Docker containers.

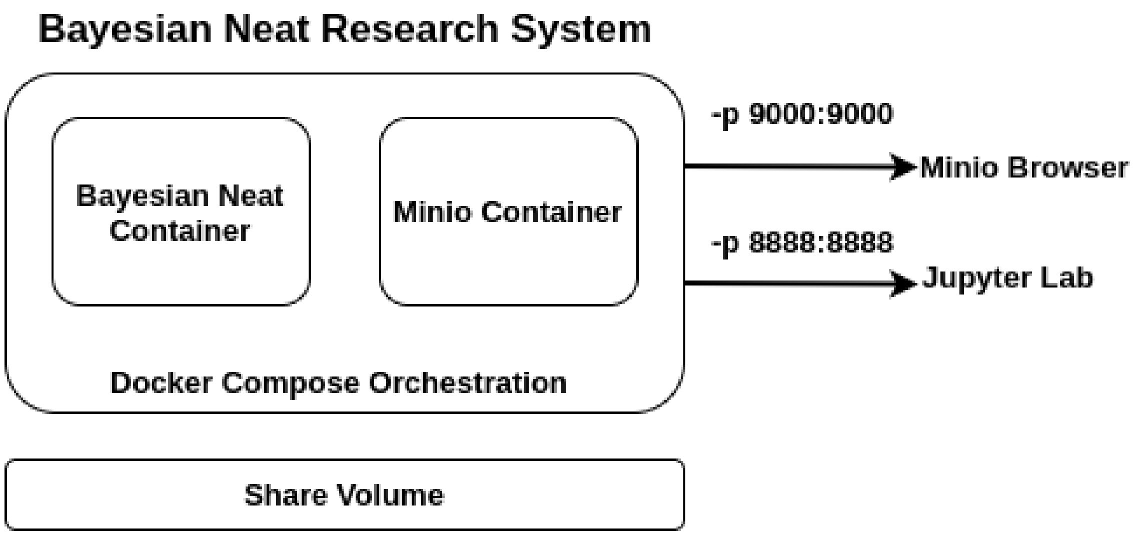


Figure 3.2: *Bayesian-NEAT Research System.*

In the main block, we see the orchestration previously discussed with two con-

[2]Docker: https://docs.docker.com/
[3]Docker: https://docs.docker.com/compose/

tainers. Finally, the two arrows pointing to the right are networking ports enabled outside the containers to interact with both containers:

1. Minio Browser (available at $localhost : 9000$) offers a window to navigate between the objects.
2. Jupyter Lab (available at $localhost : 8888$) offers access to a Jupyter Lab that contains the software installed. This service is primarily used for study the results of the algorithm.

Experiment Management System

Given the research nature of this project, it is important to correctly handle experiments data and metadata so that results can be interpreted correctly. This is especially important in this case as code keeps evolving after new issues or opportunities are discovered through the research process. Considering this, the Experiment Management System handles the following information:

1. Reports. Each execution is different, to be able to analyze it, it is important to save as much data as possible. It is worth mentioning, that all reports are stored in JSON format. Therefore, all classes need to implement a serialization process to dump its data to a file but also they need to implement a deserialization process to recover an object from its JSON definition. This is some the data and metadata saved in each report:
    (a) Execution Metadata.
    (b) Best genotypes.
    (c) Classification Metrics.
    (d) Evolution Metadata.
    (e) Start and Finish Time.
    (f) Parameters Configuration.

2. Logs. For each execution, the logs are also saved. This allows to have traceability for each execution.

Because the experiments' data is going to be used later to analyze results it is important to discuss how executions are saved. Each execution has the following metadata that allows to group interesting executions together to facilitate analysis:

1. Execution ID: a unique identifier for each execution.
2. Correlation ID: string ID that helps group together executions that are similar. For instance, we typically run the same experiment with the same parameters several times. Assigning the same correlation ID to those executions helps us to better retrieve later that information to be analyzed.
3. Dataset: the name of the dataset used for the execution. It has a role similar to the Correlation ID but while the previous one is used to group executions with the same configuration, this one is used to group executions using the same dataset.

Finally, the Experiment Management System is implemented on top of Minio[4] as a persistance layer. Minio is an Object Store designed for scalable cloud applications and it implements the same interface as Amazon Web Services (AWS) S3 service[5]. Because they share the same interface, it is easy to deploy Bayesian NEAT Research System in AWS's cloud if more computational resources are needed.

Logging

As discussed before, logging is important in any software system to add traceability of what happens. In this system we implemented a logging system of four levels:

1. Error. Logs that register an error in the system.

[4]Minio: https://min.io/

[5]AWS S3: https://aws.amazon.com/s3/

2. Info. Logs that register typical behavior of the system.
3. Debug. Logs that register extra data to debug the system.
4. Time. Logs that register the time a function takes to run.

Slack Notifier

Because experiments can take a few hours, in order to be able see results from anywhere at any time, an integration with Slack[6] was added. Slack is a workspace typically use by teams to collaborate. It has both a mobile phone app and a web app. This integration takes the results of recently finished experiments and sends them to a channel in a workspace where they can be observed by the users of that workspace.

Pytorch

Pytorch[7] is an open-source Deep Learning framework used to perform training and inference in neural networks. It is one of the two main frameworks together with TensorFlow. Despite both frameworks are similarly fast and both provide bindings to Python, the main reason why we select Pytorch over Tensorflow for this project is because there is no need to compile the networks before doing inference. While Pytorch is able to directly perform inference on a defined network, Tensorflow adds a compiling step where the network is optimized[8]. This would be okay if we were to used the same network over and over again. However, in our problem, networks are constantly changing and this compiling time would create a sensible overhead.

It is worth mentioning that there are other implementations of NEAT in a Deep Learning Framework such as Pytorch-NEAT[9] but not for Feed-forward Neural Net-

[6]Slack: https://slack.com/

[7]Pytorch: https://pytorch.org/

[8]Tensorflow 2.0 recently introduced the dynamic graphs features that does not require to compile the networks' graph. However, this wasn't available at the start of the project.

[9]Pytorch-NEAT: https://github.com/uber-research/PyTorch-NEAT

works with direct encoding.

Analysis Module

The analysis module consist of a series of procedures to easily read the reports generated by a set of correlated executions, extract interesting data and calculate the most important metrics statistics. They also have specific plotting functions that can be reused for different experiments.

Jupyter Lab

Jupyter Lab[10] is a web interface with Jupyter Notebooks. In this project, Jupyter Lab is used to both launch experiments and, especially, analyzing them. The interactive philosophy of notebooks allow to analyze results quickly and also generate graphs and plots to visually study results.

3.2 Algorithmic Solution: Bayesian NEAT

The approach described in this section consist on automatically growing Bayesian Neural Networks for Supervised Machine Learning tasks. As discussed before, inference in BNN can be transformed into an optimization problem through Variational Inference. Together with the ability of NEAT to grow increasingly complex topologies, we have developed a new approach to grow Bayesian Neural Networks. In the following sections, we break down the three fundamental problems in EAs and explain in detail how the algorithm works: Encoding, Evolution and Evaluation of BNN.

[10]Jupyter Lab: https://jupyterlab.readthedocs.io/

3.2.1 Encoding of Bayesian Neural Networks

As described in section 2.3, BNNs are defined by some structure of nodes and connections that is equivalent to normal Neural Networks. However, the weights and biases, in connections and nodes respectively, are not point-estimates but distribution-estimates. That is, parameters follow a probability distribution. Furthermore, a Gaussian Distribution is typically used (4) as the probability distribution because of its well known mathematical properties and because they are defined uniquely by its two main parameters: the mean and the variance. Therefore, in our work, we use Gaussians to model both the probability distributions representing weights and biases (Equation 3.1).

$$\begin{aligned} w_{i,j} &\sim \mathcal{N}(\mu_{i,j}, (\sigma_{i,j})^2) \\ b_i &\sim \mathcal{N}(\mu_i, \sigma_i^2) \end{aligned} \tag{3.1}$$

The genotype of the proposed solution composed of two types of genes as in NEAT.

1. Node Genes. These genes contain the information associated with the nodes of the BNN.
   (a) Node Key: an integer that uniquely identifies the node.
      i. Input nodes use negative integers from -1 to n_{input}.
      ii. Output nodes use non-negative integers from 0 to $n_{output} - 1$.
      iii. Hidden nodes use positive integers from n_{output} to $n_{output} + n_{hidden} - 1$.
   (b) Bias Mean (μ_i): the mean for Neuron i.
   (c) Bias Standard Deviation (σ_i): the standard deviation for Neuron i.
   (d) Activation Function: the function that is applied to the linear combination of the node's input.
2. Connection Genes. These genes contain the information associated with the links of the BNN.

(a) Connection Key: a pair of integers that defines the link between two neurons or nodes. First, item of the pair contains the key of the origin node while the second item contains the key of the destination node.

(b) Weight Mean ($\mu_{i,j}$): the mean for the connection weight between neuron i and j.

(c) Weight Standard Deviation ($\sigma_{i,j}$)): the standard deviation for the connection weight between neuron i and j.

In order to better understand the encoding used, we discuss the simple example shown in Figures 3.3 and 3.4. The Genotype in Figure 3.3 consist on a list of 3 nodes (one output node and two hidden nodes) and 6 connections. In the genotype, the input nodes do not need to be specified because they don't have associated parameters nor apply a function to the input. The connections, apart from the weight parameters, also know the source and destination node. The network that results from that encoding is shown in Figure 3.4.

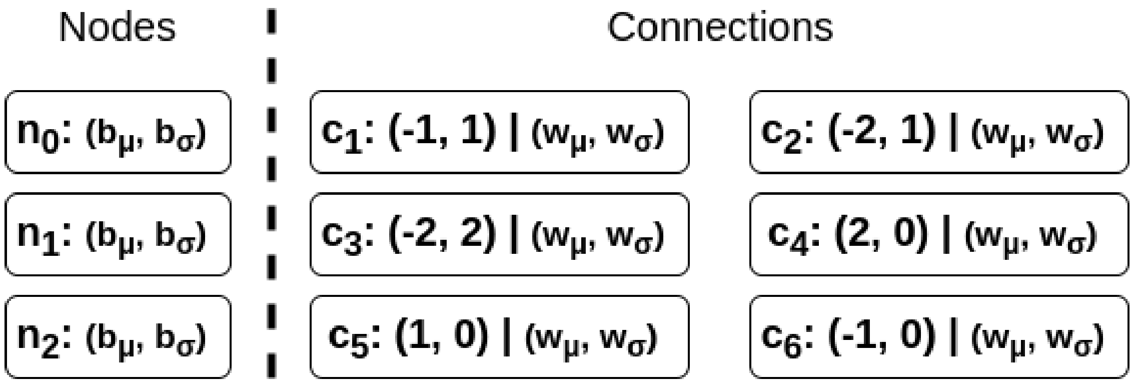


Figure 3.3: *Genotype of a example Bayesian Neural Network.*

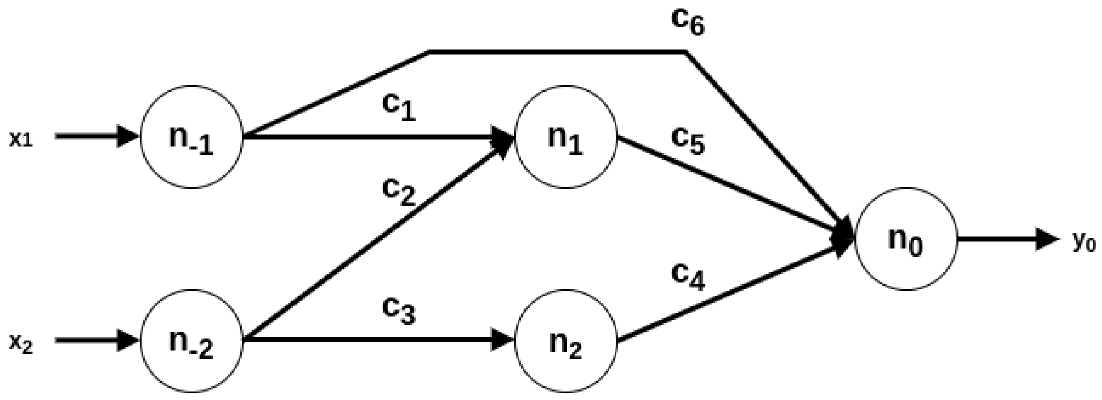


Figure 3.4: *Phenotype of a example Bayesian Neural Network.*

Despite bias and weight associated parameters are the ones that are typically learned during Neural Network training, there are other factors that define the behavior of the model such as the Activation Function used at each node. However, for sake of simplicity, in this work, we consider the Activation Function to be a given parameter and is not allowed to be optimized.

Finally, despite the direct encoding strategy explained above allows to define any kind of graph structure for the network, the generative operators restrict the graph to be a Direct Acyclic Graph (DAG) or a Bayesian Feed-Forward Neural Network. Therefore, Bayesian Recurrent Neural Networks cannot be generated by the proposed solution.

3.2.2 Evaluation of Bayesian Neural Networks

While the generative phase of the algorithmic solution is in charge of creating new BNNs that are able to solve the task, the evaluation aims at measuring the fitness of each of the BNN of the population at the problem at hand. In this section, we discuss the most important details of our evaluation phase. First, we discuss the fitness functions used for different types of problems. Then, we discuss how we build a BNN that can be evaluated using our software solution.

Fitness Function for Classification and Regression Problems

Section 2.3.2 defines different cost functions depending on the problem. That is, whether it is a Classification or Regression problem. On the other hand, they are measuring the loss of fitness so in order to translate to a framework where we want to maximize the BNN's fitness, we apply a sign mutation. Equations 3.2 and 3.3 show the fitness functions for Classification and Regression problems, respectively, used in this work. In these equations, N is the number of examples in the dataset that are being evaluated and M is the number of biases or connection weights of the network. Furthermore, $f(X_i, \theta)$ is the output of the network given an input X_i and parameters θ.

$$\begin{aligned} Fitness_C = -(\frac{1}{N}\sum_{i=1}^{N} CrossEntropy(y_i^c|f(X_i,\theta)) - \\ \beta \cdot \sum_{j=1}^{M} D_{KL}(\mathcal{N}(\mu_j,\sigma_j^2)||\mathcal{N}(\mu_P,\sigma_P^2))) \end{aligned} \tag{3.2}$$

$$\begin{aligned} Fitness_R = -(\frac{1}{N}\sum_{i=1}^{N}(Y_i - f(X_i,\theta))^2 - \\ \beta \cdot \sum_{j=1}^{M} D_{KL}(\mathcal{N}(\mu_j,\sigma_j^2)||\mathcal{N}(\mu_P,\sigma_P^2))) \end{aligned} \tag{3.3}$$

Furthermore, both equations contain the parameter β which defines how much the second term of the equations weights. Finding the right β parameter can be a substitute of finding the right prior. Basically, if we fix the prior of all the parameters to $\mathcal{N}(0.0, 1.0)$, we only need to play with the β parameter to find a good balance between the log-likelihood on the dataset and the complexity cost of the network.

Genotype to Phenotype mapping

In the proposed solution, a genotype is an object that stores its connections and nodes in hash tables. However, in order to evaluate the genotype, we need to create the equivalent Bayesian Neural Network. That is, we need to build a computation graph that allows us to get a stochastic output Y based on some input X and some parameters θ, $Y = f(X, \theta)$. Then, for each example x_i the output of the network is going to provide a list of $n\_samples$ y_i that will allow us to characterize the predictive distribution. The output network is a model that can be evaluated using a Deep Learning framework (Pytorch). This Genotype-to-Network mapping process is not a straight forward process due to the requirement that there can be multi-hop connections. That is, connections can jump through layers. This is the main difference with standard fully connected networks.

With that in mind, we have designed a network built by layers that can handle those requirements. Figure 3.5 shows the diagram of a stochastic network with 1 hidden layer that is being generated, where SLL stands for Stochastic Linear Layer and Act refers to a layer that applies an Activation Function to each of its inputs and it can be considered as the Activation Layer.

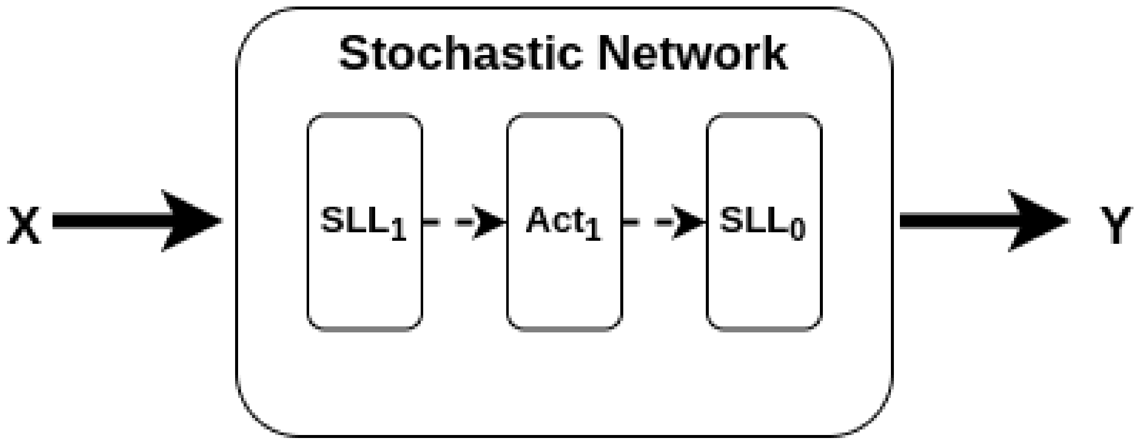


Figure 3.5: *Stochastic Network with 1 hidden layer.*

Furthermore, we have created a series of abstractions that help to encapsulate the different procedures required to map the genotype to a network as well as the behavior of the network itself. Figure 3.6 shows a diagram with all the components required to map a genotype to a network. Basically, there are two main components: the Genotype-to-Layers and the Stochastic-Network-Builder.

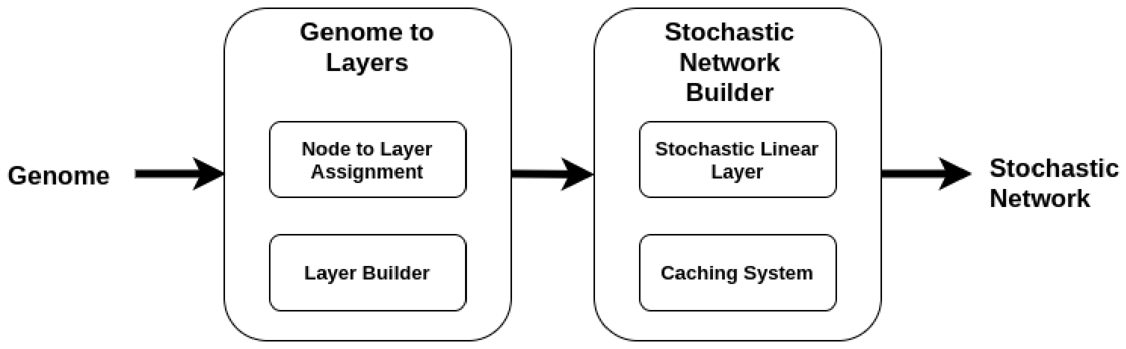


Figure 3.6: *Genotype to Network Diagram.*

For sake of understanding, we explain how these components work starting from the output and describing the system backwards. As said before, the output of this mapping is a Stochastic Network that we can use, together with a fitness function, to evaluate an individual.

Stochastic-Network-Builder Component

This component builds a Stochastic Network given the information for each layer that is built in the previous stage. The following is a capture of the Python code used to implement the forward pass of the Stochastic Network. Here we can see two parts that are not typical of standard Neural Networks. The first one is the use of cache in the middle of the forward pass and the second one is the Stochastic Linear Layer. We describe those two pieces below.

```
def forward(self, x):
    kl_qw_pw = 0.0
    start_index = self.n_layers - 1
    for i in range(start_index, -1, -1):
        # cache needed values from past layers
        for index_to_cache in self.layers[i].indices_of_nodes_to_cache:
            self._cache[(i, index_to_cache)] = \
            x.index_select(1, torch.LongTensor((index_to_cache,)))

        # append needed values for next layers
```

```
            chunks = [x]
            for index_needed in self.layers[i].indices_of_needed_nodes:
                chunks.append(self._cache[index_needed])
            x = torch.cat(chunks, 1)

            # apply Stochastic Linear Layer
            x, kl_qw_pw_layer = getattr(self, f'layer_{i}')(x)
            kl_qw_pw += kl_qw_pw_layer

            # apply Activation Layer
            if i > 0:
                x = getattr(self, f'activation_{i}')(x)
        return x, kl_qw_pw
```

Stochastic Linear Layer
This subcomponent is the one that performs all the sampling and where the stochasticity of the network comes from. In standard Neural Networks, there is what is called a Linear Layer. That is a fully connected layer that applies the algebraic Equation 3.4.

$$X_{out} = W^T \cdot X_{in} + B \tag{3.4}$$

In the Bayesian case, we can derive a similar equation. However, in our case, the matrices will be sparse, depending on the number of connections. This is not a problem as there is research for efficient sparce matrix multiplication for deep learning (14). As discussed before, we are sampling from the weights and biases distributions. However, this is highly inefficient. By applying Equation 3.5 we are able to optimize the sample efficiency of the system since we can create a single matrix that sample from $\mathcal{N}(0,1)$ more efficiently than sampling each parameter individually. Such transformation is possible because of the good mathematical properties of Gaussian distributions. This equation is the transformation that is applied at a Stochastic Linear Layer.

$$X_{out} = W_{\mu}^{T} \cdot X_{in} + (W_{\sigma}^{T} \cdot X_{in}) \cdot \mathcal{N}(0,1) + B_{\mu} + B_{\sigma} \cdot \mathcal{N}(0,1) \tag{3.5}$$

Cache System

As discussed before, our generative part of the algorithm allows to create individuals with connections that jump through layers. Let's consider the example in Figure 3.4, we have a multi-hop connection from Node -1 to Node 0. While Node 0 belongs to Layer 0 (output layer), Node -1 belongs to Layer 2 (input layer). The caching system in the Stochastic Network allows us to save the output of Node -1, while computing the forward pass on the hidden layer, so it can be used again in the forward pass of the output layer.

Genotype-To-Layer Component

This component uses the genotype to build the parameters of each layer. First, we start by assigning each node to a layer. Then, once we have our layers defined, we the parameters for each of the layers.

Node-to-Layer-Assignment

The output of this procedure is to uniquely map each node to a layer. It works as follows:

1. Build the output layer (layer(i)). Given the number of outputs we can look-up the bias distributions associated with the output layer.

2. Recursively build the other Layer(i-1):

   (a) Find the nodes in Layer(i-1). This is done by examining the connections between nodes that go to the Layer(i).

   (b) Get bias distribution for Layer(i-1).

   (c) Get weight distributions for connections between Layer(i-1) to Layer(i).

   (d) Check whether we are at the Input Layer. We can check this by looking at the sign of the nodes' keys. If they are negative, that means that they are input nodes. Therefore, we stop the recursion.

The bias and weights parameters are given in matrices. In the case that the parameter for a position in the matrix does not exist, then it is a zero. Working with matrices makes the computation easier as there is specialized hardware (GPU) that can accelerate operations.

3.2.3 Evolution of Bayesian Neural Networks

As said before, this work is based on the NEAT algorithm (30). In terms of software, we also relied on some initial implementation of the evolutionary part of NEAT (NEAT-Python[11]). However, there are some fundamental changes that were introduced to allow encoding BNN as well as other algorithmic details that are different from original NEAT. Figure 3.7 shows a high level diagram of the evolutionary process that populations undergo through many generations. Since genotype evaluation is already explained in Section 3.2.2, in this section, we explain the Initialization, Speciation and Reproduction of populations of BNN.

[11]NEAT-Python: https://neat-python.readthedocs.io/

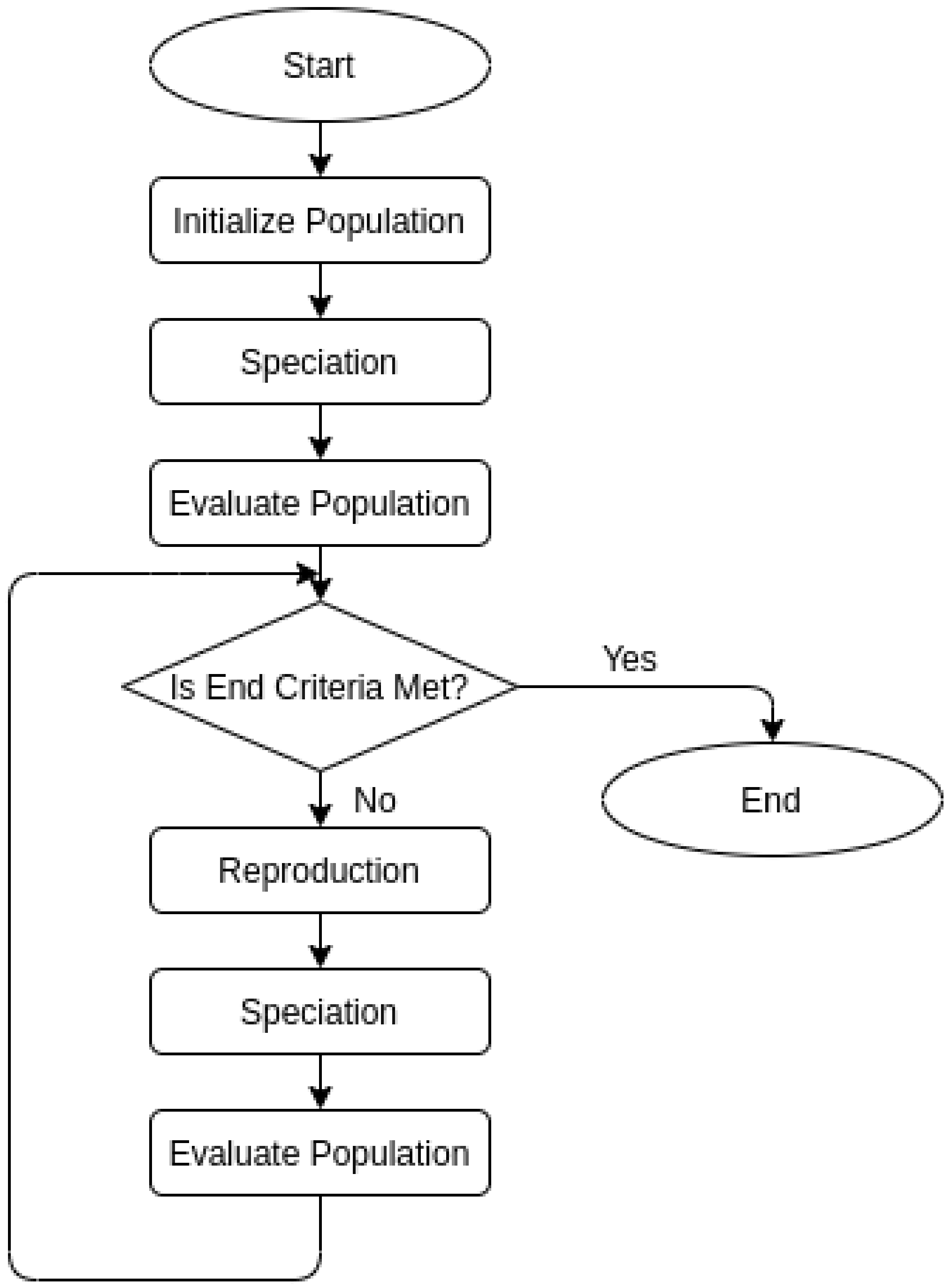


Figure 3.7: *Flowchart of Bayesian-NEAT.*

Population Initialization

Given the size of the population as a parameter (*pop_size*), the algorithm automatically generates an initial set of genotypes using a random initialization approach. For each genotype, first, it generates the output nodes based on the number of outputs (n_output). Then, it generates hidden nodes, if configured using the parameter $n\_initial\_hidden\_neurons$. Finally, it generates the connections.

Generating the connections is the more complex part of each genotype initialization and depends on the following configuration parameters, $is\_initial\_fully\_connected$, $initial\_nodes\_sample$ and $n\_initial\_hidden\_neurons$. Now let's review how the different combination of parameters affect the initial connections:

1. Fully Connected and some Hidden Neurons. In this case, the algorithm creates all possible connections between the Input Layer and the Hidden Layer and between the Hidden Layer and the Output Layer. Therefore the number of initial connections is $n\_input \cdot n\_initial\_hidden\_neurons + n\_output \cdot n\_initial\_hidden\_neurons$.

2. Fully Connected and no Hidden Neurons. In this case, the algorithm creates all possible connections between the Input Layer the Output Layer. Therefore the number of initial connections is $n\_input \cdot n\_output$.

3. Not Fully Connected and Hidden Neurons. In this case, the algorithm creates $initial\_nodes\_sample$ random connections between the Input Layer the Hidden Layer and all possible connections between the Hidden Layer and the Output Layer. Therefore the number of initial connections is $initial\_nodes\_sample \cdot n\_initial\_hidden\_neurons + n\_output \cdot n\_initial\_hidden\_neurons$.

4. Not Fully Connected and no Hidden Neurons. In this case, the algorithm creates $initial\_nodes\_sample$ random connections between the Input Layer the Output Layer. Therefore the number of initial connections is $initial\_nodes\_sample \cdot n\_output$.

The connection generation approach aims at ensuring that we have at least one path from input to all outputs. That is, all outputs will have an associated value that is a function of the input, although not necessarily all the input dimensions will affect each output. This is especially important in high dimensional problems, where if one connection is deleted soon, it might turn out in not being able to generate a signal for one of the outputs.

Speciation

The speciation mechanism allows to segmentate genotypes based on a distance measure between genotypes, so that different species are given the opportunity to develop themselves for some generations before they die. The ad-hoc distance measure used in the solution takes into account both the topology differences as well as the weight and biases differences.

A special distinction between original NEAT and the one used in this work is how the number of species is decided. While in the original NEAT, the number of species depends on the actual distances between genotypes, in this work we have implemented a new method to keep the number of species stable. That is, given a fix number of species ($n\_species$), it always keeps a constant number of species. In our tests we found the original approach to be quite unstable and dependent on some distance thresholds defined.

Reproduction

In the case of NEAT, Reproduction step includes both Recombination and Mutation. At the beginning, of each generation, the number of desired off-springs per specie is estimated considering the overall fitness of each specie. Then, reproduction happens independently for each specie.

In order to breed, first, parents need to be selected. NEAT implements a random tournament between the best parents (those with higher fitness value). Then, parents are chosen randomly from that pool and breed to generate a new off-spring. Here is where recombination or crossover happens. This process consists on inheriting connections and nodes that can also be crossed-over between parents. After this process, a new phenotype is created.

Now, this newborn phenotype is mutated. First, an architectural mutation is applied with a certain probability. Architectural mutation consist on either adding or removing genes (connections or nodes). Given the constraints that we impose in the networks, when mutating architecture a feasibility check is implemented. Architectural mutation depends on several parameters that define the probability of one of the following events happening:

1. Create a new node (*add_node_prob*). This mutation works as selecting a connection and breaking into a new connection and a new node.
2. Delete a node (*delete_node_prob*). First, the mutation operator needs to find a set of candidate nodes that can be removed without affecting the network constraints. Then, it randomly selects one.
3. Create a new connection (*add_node_prob*). This operator creates a valid pool of possible origin and destination nodes. Then, it needs to verify that a connection to them is not adding a cycle into the graph (this would make the network recurrent).
4. Delete a connection (*delete_connection_prob*). Select a random connection and delete it, guaranteeing that we still have a path from input to each output.

Finally, the parameter values of each gene are also mutated with a certain probability (*mutate_rate*) and intensity (*mutate_power*).

3.2.4 Fine Tuning

Based on initial experiments, we detect that the algorithm is not competitive in some datasets with Backpropagation-based methods. In order to improve on these results, we add a final fine-tuning stage to further optimize the fitness while fixing the architecture. Given that the fitness function and the architecture is differentiable, we leverage Pytorch's Autograd[12] module to calculate gradients of the parameters

[12]https://pytorch.org/docs/stable/autograd.html

and apply Stochastic Gradient Descent (13) on them. Furthermore, Early-Stopping technique is used while training each network to obtain the best network on the validation set.

The fine-tuning stage works as follows:

1. Select the fittest genotype per specie.
2. Train each of these genotypes for $epochs\_fine\_tuning$ epochs:
   (a) Each epoch updates parameters using the training set and calculates the loss using the validation set. If the validation loss improves, the network parameters are saved.
   (b) Training continues until all $epochs\_fine\_tuning$ are done or the network has not found a better network in 200 epochs.
   (c) The network with the best validation loss is returned.
3. The best network is the one with the best validation loss from all the species.

3.2.5 Algorithm Parametrization

Appendix B describes the possible parametrization of Bayesian-NEAT algorithm as well as gives some good default values. Experiments are run with this parameters unless specified otherwise.

Chapter 4

Experiments and Results

In this chapter, we describe the experiments carried out, as well as discuss the most important results obtained.

4.1 Experiments Design

The objective of these experiments is to quantitative and qualitative validate the proposed solution in Supervised Classification Learning tasks. Despite that the proposed solution can also be used in regression tasks, for simplicity, in these experiments we focus only on classification tasks. In order to do so, we ask the following questions:

1. Is Bayesian-NEAT able to solve Supervised Classification Learning tasks?
2. How does Bayesian-NEAT perform in terms of classification metrics with respect to benchmarks?
3. Is there an use-case where Bayesian-NEAT can beat benchmarks?
4. Is Bayesian-NEAT able to estimate the uncertainty in the prediction of individual data-points?

4.1.1 Datasets

We used several datasets, shown in Table 4.1, to demonstrate the use of the proposed approach. The reason for using several datasets is to be able to assess more confidently how the algorithm's behavior could translate to new unseen datasets. In order to select datasets, we focused on having diversity based on three criteria: number of examples, input dimension's size and output dimension's size.

Table 4.1: *Classification Datasets*

Name	Number Examples	Number Attributes	Number Classes
Iris (23)	150	4	3
Wine (31)	178	13	3
Breast Cancer (21)	569	30	2
Titanic (9)	1045	6	2
SpamBase (19)	4601	57	2
MNIST (32)	70000	64	10

It is worth mentioning that datasets are split into train and test sets with proportions 75 % and 25 % respectively. Furthermore, for training networks using Backpropagation, we also use 20 % of the training set as validation set. Furthermore, MNIST images were downsampled from 32x32 pixels to 8x8 for the sake of computation performance.

4.1.2 Algorithm Parametrization

For all the experiments we used the default values in Table B.1 unless specified otherwise. It is worth mentioning, how we select the β. This is an important parameter as it defines the weighting factor between the log-likelihood and complexity cost in the multi-objective fitness function expressed in Equation 2.4. In order to define a proper value, we need to consider how both components of the fitness function work:

1. The log-likelihood is an estimate of getting the output right. That is, it helps to get better results in terms of classification metrics.

2. The complexity cost is a penalty on the parameters being different from their prior belief and it tends to grow with the number of parameters or size of the network.

Therefore, if β is too high the Bayesian-NEAT is not incentivized to grow its architecture. On the other hand, if too small there is not any penalty on the architecture. Because the complexity cost term grows with the network and the log-likelihood tends to decrease as weights are optimized, we decide to select a β such that the complexity cost represents around 5 % of the total cost function. Despite defining this parameter could be further investigated, we found that this criteria has worked well in practice. The final values for each dataset are shown in Table 4.2.

Table 4.2: β *by Dataset.*

Name	β
Iris	0.0001
Wine	0.00008
Breast Cancer	0.00005
Titanic	0.0001
SpamBase	0.00001
MNIST	0.000005

4.1.3 Benchmarks

In order to validate Bayesian-NEAT, we compare it with more traditional methods, that serve as benchmarks. We also add a naive benchmark that is used in the first experiment. The benchmarking methods are:

1. Random-Guessing (RG).

2. Multi-Layer Perceptron (MLP).
3. Bayesian Multi-Layer Perceptron (B-MLP).
4. NEAT.

The reason for using those methods is to be able to compare two effects:

1. Standard vs. Bayesian networks. In this case, we compare how standard networks (MLP and NEAT) compare with Bayesian NEAT. In this regard, we do not expect Bayesian methods to over-perform standard networks (6). However, it is interesting to see how far they are from them.
2. Traditional search vs. Evolutionary search. While in traditional search architecture and weights learning is decoupled, in Neuro-Evolution both architecture and weights can be evolved together. In this comparison, we wish to establish some relationship between architecture and classification performance.

Multi-Layer Perceptron

The MLP is trained using Back-Propagation to train the weights and a naive Grid Search approach to find the best architecture. The method searches for the best combination of architecture parameters:

1. Number of Hidden Layers.
2. Number of Nodes per Layer.

Each execution in that search trains the network with those hyperparameters using both train and validation sets. The search method keeps the network that performs better in the validation set. Finally, metrics are calculated using the test set.

Bayesian Multi-Layer Perceptron

In this case, the weights' training and the architecture's search follows the same approach as in the MLP method. The only difference is that, in this case, the network is using the Stochastic-Layer that we built for Bayesian-NEAT. Therefore, it is also able to provide a stochastic output. It's worth mentioning that training this network with BackPropagation is possible due to using Variational Inference because both loss and network can be differentiated.

NEAT

This algorithm consist on NEAT for standard neural networks. In terms of implementation is the same as Bayesian-NEAT but with some tweaks. First, the complexity cost is fixed to zero ($\beta = 0.0$). This makes the network optimize for just the log-likelihood, or Cross-Entropy loss in classification problems. Second, we fix the Standard Deviation of the biases and connections to be zero ($fix\_std = True$). That is, standard deviation is always zero and, therefore, the Stochastic Layer becomes a deterministic one. This allows us to make the comparison of Standard vs. Bayesian networks using the same evolutionary search approach and without the need of further development. In terms of implementation, it required additional work so that both backpropagation and mutation does not update the standard deviation parameters.

Random Guessing

Random Guessing (RG) consist on predicting a class by just sampling from the distribution of classes. To calculate classification metrics for the RG method for each dataset, we directly take the ratio given by the number of classes. Despite this method can be inaccurate (due to class imbalance), we think that is close enough for our comparison purpose, especially after seen that Bayesian-NEAT performs clearly better than them. We use this benchmark to prove that our proposed algorithm is actually able to learn.

4.1.4 Data Processing

Both benchmarks and our proposed solution require some minimal data pre-processing and post-processing in order to work. Regarding pre-processing, we applied the following transformations as recommended by (12):

1. Attribute Encoding. Some of datasets have some categorical attributes that need to be transformed to numeric values. For that, we have applied One-Hot Encoding method.

2. Attribute Normalization. In neural networks it is needed to remove scale of the input data and normalized around zero.

3. Data Repetition. This transformation only applies to Bayesian methods (B-MLP and B-NEAT). In order to generate a robust probability distribution at the end of the softmax layer, many samples are needed. In order to get many samples, both input tensor X and output tensor Y need to be repeated $n\_samples$ times.

Because Bayesian methods require data repetition at the input, at the output they require a post-processing step to convert the output distribution into a prediction. In order to give a prediction, a BNN relies on the expected value of such distribution. That is, the mean.

4.1.5 Metrics

In order to measure how the algorithms perform in the classification tasks we use the following metrics:

1. Classification Accuracy. It is the proportion of correct predictions with respect to the total number of cases. It is typically shown as a percentage.

$$Accuracy = \frac{Number of Correct Predictions}{Total Number of Predictions} \cdot 100\% \tag{4.1}$$

2. Precision. Intuitively, it can be thought as how good is the classifier at reducing False Positives. That is, classifying as positive a negative example. In the case of multi-class problems, the Precision is calculated as the weighted sum of each class's precision. Furthermore, each class is calculated following the One-vs-Rest approach.

$$\begin{aligned} Precision_c &= \frac{TruePositive}{TruePositive + FalsePositive} \\ Precision &= \frac{1}{N} \cdot \sum_{c=1}^{C} N_c \cdot Precision_c \end{aligned} \tag{4.2}$$

3. Recall. Recall is a complementary metric to Precision and it can be thought as how good is the classifier at reducing False Negatives. That is, classifying as negative a positive example. In multi-class, it follows the same philosophy as

$$\begin{aligned} Recall_c &= \frac{TruePositive}{TruePositive + FalseNegative} \\ Recall &= \frac{1}{N} \cdot \sum_{c=1}^{C} N_c \cdot Recall_c \end{aligned} \tag{4.3}$$

4. F1 Score. It combines both Precision and Recall into a single scalar metric by calculating their harmonic mean. In the case of multi-class problem, it computes the metric individually for each class and, then, weighted them based on the of instances of each class as shown in Equation 4.4.

$$\begin{aligned} F1_c &= 2 \cdot \frac{Precision_c \cdot Recall_c}{Precision_c + Recall_c} \\ F1 &= \frac{1}{N} \cdot \sum_{c=1}^{C} N_c \cdot F1_c \end{aligned} \tag{4.4}$$

As F1 score considers both Precision and Recall, we use that one to better capture successful classification.

4.2 Experiment I. Solving Classification Tasks

The objective of this experiment is to validate if the proposed approach is actually able to solve Supervised Classification Learning problems in a consistent manner.

4.2.1 Design of Experiment I

We can say that Bayesian-NEAT is able to solve a classification problem if the classification metrics results are higher than what it would be obtained by just random chance or Random-Guessing.

In order to satisfy the requirements above, this experiment consists on running $k = 5$ executions on Bayesian-NEAT for each of the considered datasets and calculate the classification metrics discussed before on the testing set.

Finally, we also add a comparison between two variants of the Bayesian-NEAT algorithm as we have seen that there is a boost on performance from one to another variant. These two variants consist on the proposed solution with and without the final stage of parameters' fine-tuning.

4.2.2 Results and Discussion of Experiment I

Table 4.3 shows the mean value of the metrics for all the executions for each dataset for both versions of Bayesian-NEAT. Then, Figure 4.1 shows several boxplots comparing both methods and the RG value (horizontal blue line) for each dataset. Based on the results, we can confidently claim that the proposed solution is able to solve Supervised Classification problems consistently as it obtains classification metrics that are always better than simply random guessing.

Table 4.3: *Bayesian-NEAT Mean Classification Metrics.*

Dataset	Accuracy	F1	Accuracy$_{FT}$	F1$_{FT}$	RG Accuracy (%)
Iris	67.36 %	0.6011	95.26 %	0.9528	33.33 %
Wine	97.33 %	0.9735	96.88 %	0.9690	33.33 %
Breast Cancer	96.08 %	0.9608	96.36 %	0.9636	50 %
Titanic	78.39 %	0.7786	78.62 %	0.7808	50 %
SpamBase	84.81 %	0.8460	88.13 %	0.8798	50 %
MNIST	38.52 %	0.3233	63.20 %	0.6084	10 %

On the other hand, Fine-Tuning clearly helps improve performance in three of the datasets studied (Iris, SpamBase and MNIST) while not hurting the performance in the other ones. There is one dataset (Wine) where the classification metrics are actually worst in mean but the difference is not significant. In fact, the actual validation loss is better after Fine-Tuning as Table 4.4 shows.

Table 4.4: *Bayesian-NEAT Mean Validation Fitness.*

Dataset	Without Fine-Tuning	With Fine-Tuning
Iris	0.885162	0.599088
Wine	0.676358	0.642262
Breast Cancer	0.386794	0.369603
Titanic	0.530425	0.499358
Spambase	0.496322	0.488217
MNIST	2.181421	1.908603

Since the fitness function is just a proxy of the actual classification metrics, it can happen that the metrics are slightly worse, especially in small datasets like Wine where results can have greater variability. That is, the smaller the dataset, the higher the impact that one mis-classified example has in the calculated metrics. Finally, looking at the validation loss, it also seems that for datasets Breast

Cancer, Wine, Titanic and SpamBase the approach without fine-tuning is already optimizing the fitness correctly because adding the fine-tuning does not provide a great improvement to the optimization. Therefore, it makes sense that we do not see a large improvement in the classification metrics for those datasets.

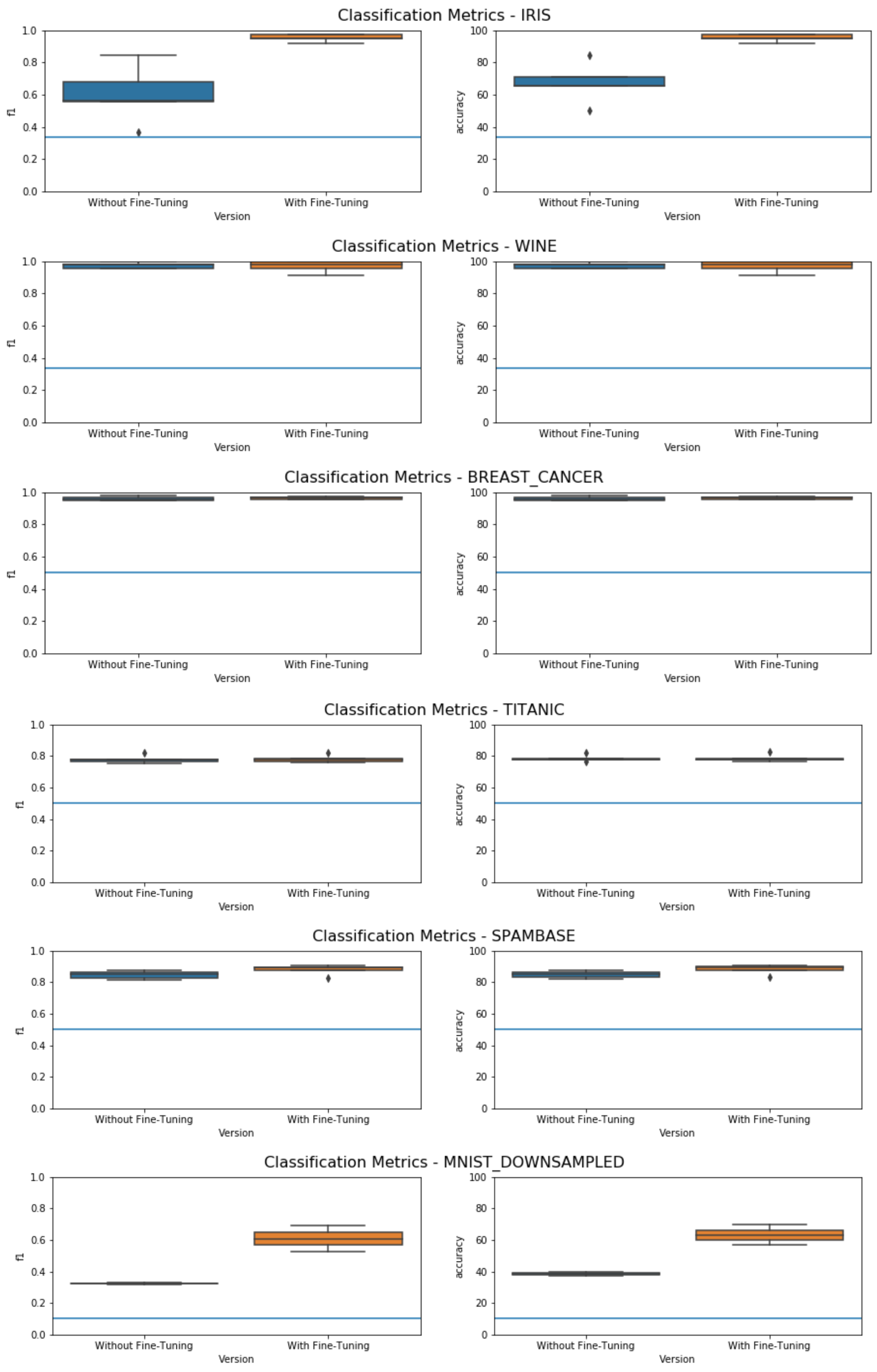


Figure 4.1: *Boxplot with Experiment I Metrics.*

4.3 Experiment II. Comparison against Benchmarks

The objective of this experiment is to compare how Bayesian-NEAT compares against the defined benchmark methods. The comparison is made at two levels: in terms of classification performance, and model's complexity and architecture.

4.3.1 Design of Experiment II

In this experiment, we compare the Bayesian-NEAT algorithm with the benchmark methods described before. As in the previous experiments, each method is executed $k = 5$ times for each dataset in order to get robust performance metrics.

4.3.2 Results and Discussion of Experiment II

First, we evaluate the results based on the results in terms of classification performance showed in Tables 4.5 and 4.6.

Table 4.5: *Test F1 Score by Method.*

Dataset	MLP	Bayesian-MLP	NEAT	Bayesian-NEAT
Iris	0.947235	0.952652	0.968563	0.947556
Wine	0.955357	0.933477	0.968971	0.969041
Breast Cancer	0.976140	0.987396	0.971846	0.966448
Titanic	0.780609	0.780333	0.818103	0.781404
Spambase	0.929950	0.908552	0.922762	0.881245
MNIST	0.957968	0.755776	0.683181	0.622393

Table 4.6: *Test Accuracy (%) Score by Method.*

Dataset	MLP	Bayesian-MLP	NEAT	Bayesian-NEAT
Iris	94.736842	95.263158	96.842105	94.736842
Wine	95.555556	93.333333	96.888889	96.888889
Breast Cancer	97.622378	98.741259	97.202797	96.643357
Titanic	78.511450	78.473282	82.137405	78.702290
Spambase	92.988705	90.929626	92.289314	88.256588
MNIST	95.800000	76.208000	70.512000	64.656000

With a quick look, we can see that Bayesian-NEAT is competitive with benchmarks in five out of six datasets (Iris, Wine, Breast Cancer, Titanic and Spambase) as they get very similar results. Now let's compare Bayesian-NEAT individually with each benchmark:

1. Bayesian-NEAT vs. B-MLP. Comparing with its Bayesian counterpart we observe they get very similar results in all datasets but are clearly worse in those with higher dimensionality (Breast Cancer, Spambase and MNIST). Furthermore, despite Spambase have similar input dimensionality, Bayesian-NEAT performs much worst (12 %) in the MNIST dataset which has a higher output dimensionality (10 classes). As we see below, this is a current limitation of the proposed approach.

2. Bayesian-NEAT vs. NEAT. In this case, the methods use the same algorithm for evolving and searching architecture. The only differences are that NEAT does not impose a complexity cost and the output is deterministic. Looking at classification metrics we see that NEAT slightly outperforms Bayesian-NEAT in five out six datasets. This phenomenon is also observed when comparing MLP against Bayesian-MLP. This is not surprising as, as we said before, Bayesian Neural Networks typically perform worse in terms of classification metrics than the standard networks (6).

3. Bayesian-NEAT vs. MLP. In this case, we observe higher differences in the datasets with higher dimensionality and more data availability (Spambase and MNIST). In fact, in these two datasets, a traditional MLP performs better than any other method. These are not strange results as those are the datasets that have higher amount of data and dimensionality. MLPs behave well on high dimension because they do not have an extra parameter to optimize (σ), and the more data is given, the better they get.

It is worth discussing the case of MNIST. In this dataset, Bayesian NEAT is very far from the Bayesian-MLP's performance. At the same time, NEAT is also far away from MLP's performance. This leads us to think that we have encountered a scalability limitation of the algorithm since Spambase has a similar input dimension and performance is not degraded, we establish the hypothesis that the scalability problem comes from the output dimension. What happens if we reduce the output dimensionality of MNIST? To answer this question we run an additional experiment with five classes. That is taking as classes the digits 0, 1, 2, 3 and 4. The results of this experiment are in Tables 4.7 and 4.8.

Table 4.7: *Test F1 Score for MNIST-5.*

Dataset	MLP	Bayesian-MLP	NEAT	Bayesian-NEAT
MNIST-5	0.983851	0.94216	0.939491	0.900692

Table 4.8: *Test Accuracy for MNIST-5.*

Dataset	MLP	Bayesian-MLP	NEAT	Bayesian-NEAT
MNIST-5	98.384	94.224	93.968	90.128

As we can see the gap between methods is much lower. This indicates that our approach has some limitations with the output dimension size. Actually, this makes sense because as we increase the output dimension we are also increasing the complexity of the search space. One possible solution could be to run the algorithm for many more generations. However, it becomes unfeasible with the resources available.

Regarding models's architecture and complexity, Tables 4.9, 4.10 and 4.11 show the number of parameters, the number of network elements (sum of nodes and connections in a network) and the number of layers, respectively, for each of the methods.

Table 4.9: *Mean Number of Model's Parameters.*

Dataset	MLP	Bayesian-MLP	NEAT	Bayesian-NEAT
Iris	241.0	402.0	50.40	61.2
Wine	301.5	600.0	123.60	198.0
Breast Cancer	551.5	1204.0	93.40	144.8
Titanic	233.0	310.0	83.60	96.4
Spambase	1042.0	780.0	185.75	369.0
MNIST	1762.0	3470.0	335.20	637.2

Table 4.10: *Mean Number of Network Elements.*

Dataset	MLP	Bayesian-MLP	NEAT	Bayesian-NEAT
Iris	241.0	201.0	50.40	30.6
Wine	301.5	300.0	123.60	99.0
Breast Cancer	551.5	602.0	93.40	72.4
Titanic	233.0	155.0	83.60	48.2
Spambase	1042.0	390.0	185.75	184.5
MNIST	1762.0	1735.0	335.20	318.6

Table 4.11: *Mean Number of Network's Layers.*

Dataset	MLP	Bayesian-MLP	NEAT	Bayesian-NEAT
Iris	2.6	2.4	5.6	4.0
Wine	2.5	2.6	8.4	12.4
Breast Cancer	2.7	2.6	7.6	11.8
Titanic	2.5	2.0	12.6	10.4
Spambase	2.7	2.4	8.5	13.8
MNIST	2.6	2.8	3.6	4.4

From those tables, we can observe several patterns across different datasets:

1. Both NEAT and Bayesian-NEAT require far less parameters and need smaller graphs than MLP networks.

2. Despite having less parameters, Bayesian-NEAT and NEAT generate deeper networks than both fully-connected MLP approaches. That is, they have more layers.

3. Bayesian-NEAT typically generates smaller networks than NEAT. That is, the total number of nodes and connections is smaller than in NEAT. However, Bayesian-NEAT usually has the same or more parameters than NEAT. This is explained by the fact that the BNNs, for the same network structure, they double the amount of parameters as they also have a standard deviation parameter. Of course, this increases the complexity of the training process and results obtained could be local optimums.

From these results is obvious than typical MLP approaches have typically contain many nodes and connections that do not contribute to optimize the model's classification metrics. In this regard, a Neuro-Evolution approach is able to generate more efficient networks than traditional Neural Architecture Search methods.

4.4 Experiment III. Comparison against Benchmarks in presence of Mislabeled Data

Based on the previous results where Bayesian-NEAT is able to reach competitive results with most benchmarks, we look for an application where Bayesian-NEAT can standout. As discussed before, Bayesian-NEAT has, typically, a lower amount of parameters and smaller architectures. Therefore, we think that it could generalize better and be more robust in noisy environments. In order to study this hypothesis, Bayesian NEAT is tested in datasets with mislabeled data and compared against the defined benchmarks.

4.4.1 Design of Experiment III

The experiment consist on increasingly modifying the amount of mislabeled examples in the datasets and measure the classification metrics for each execution. Mislabeled examples, in the context of Classification problems, are those whose class is incorrectly assigned. For that purpose, we use the parameter *label_noise* that represents the probability of a training example is wrong. In case of being wrong, a random label is picked uniformly from the rest of the classes. In this experiment, we only add noise to the training and validation set while the testing set is always correct. As with the previous experiments, we run each experiment $k = 5$ times to ensure robust results.

4.4.2 Results of Experiment III

Figure 4.2 shows how the classification metrics evolve as we increase the rate of mislabeled data. The y-axis in the graphs on the left shown the F1 score while the y-axis in the graphs on the right show the Accuracy. Each graph shows the performance for Bayesian-NEAT and the benchmarks discussed before. For each model, each graph displays both the mean and the variability for each experiment as one standard deviation above and below the mean.

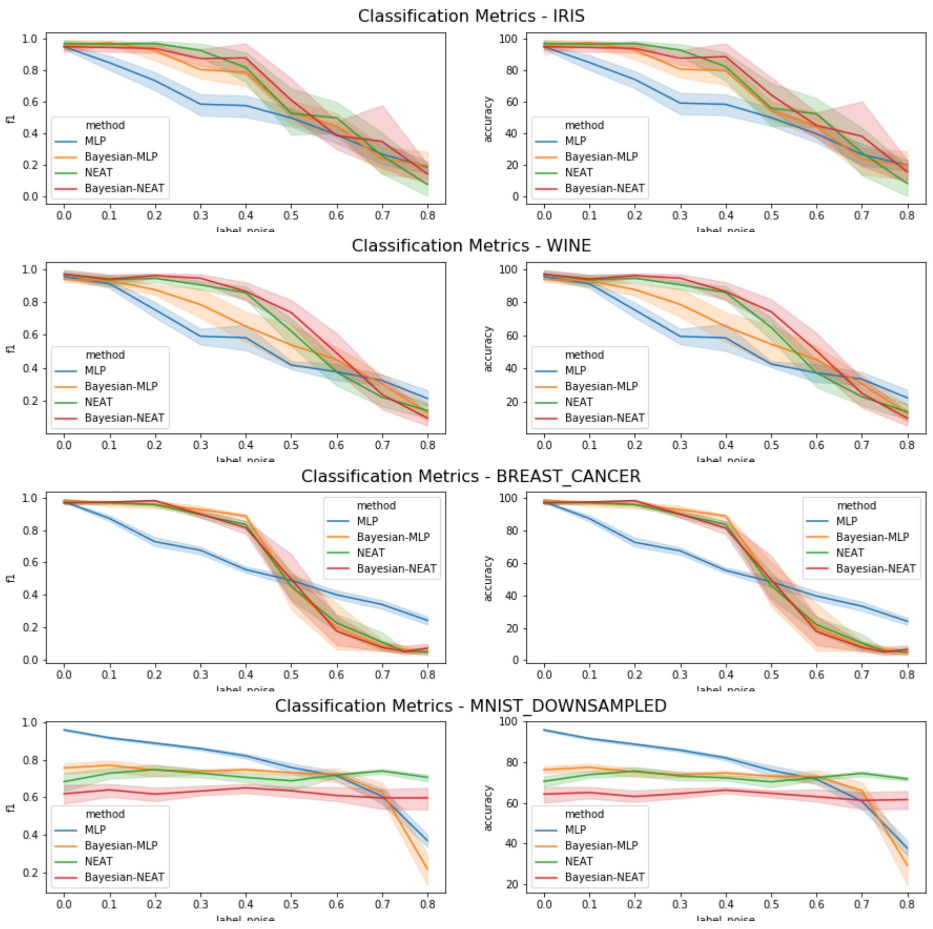


Figure 4.2: *Classification Metrics based on Mislabeled Data Rate.*

4.4.3 Discussion of Experiment III

Figure 4.2 shows how the classification metrics evolve as we increase the rate of mislabeled data. As we can see, the MLP model is the one has a clearly different behavior than the others. The MLP model seems to reduce performance linearly as we increase the noise ratio. This holds for all datasets but for MNIST, where after $label\_noise > 0.7$, the model performance dramatically drops.

On the other hand, the other models are more robust to mislabeled data as *label_noise* starts to increase. However, we can see that there is some point after the performance degrades faster than for MLP. In practice, we think that is not important as real datasets tend towards a small percentage of error. In this regard, it is worth discussing where that inversion of behavior tends to happen. Recalling the experiment design, if a label is mislabeled that means that the wrong label assigned is randomly picked from the other set of labels following a uniform distribution. That is, if we are in a binary classification problem, the mislabeled example will always pick the other class. However, if we are in MNIST (with 10 classes) the label selected will be picked from the other nine labels. Therefore, the impact of an error in MNIST makes sense that it is smaller than in a dataset with fewer classes and it makes sense that models keep performance until close to $label\_noise$ 0.9.

As we observe, all Bayesian methods and NEAT are more robust to this noise than the traditional MLP model. Looking at these results as well as the results regarding models' architecture, we think that robustness against label noise comes from:

1. Bayesian methods (Bayesian-NEAT and Bayesian-MLP) typically act as model regularizers, helping to better generalize to unseen data. Gat et al (11) proved that the Dropout method (26) used to train Neural Networks has a bayesian interpretation where weights follow a Bernoulli distribution.
2. Simpler models are more robust to noise. Models with fewer parameters have the drawback that are less expressive and, therefore, cannot model complex functions. However, they are also more robust to noise.

Based on this, we would expect to see Bayesian-NEAT performing better because it has both the Bayesian nature and it generates smaller models. However, based on the evidence we have, we cannot conclude that.

4.5 Experiment IV. Modeling Uncertainty

As discussed earlier, one of the most interesting characteristics of Bayesian Neural Networks is their ability to provide a measure of uncertainty for individual examples.

4.5.1 Design of Experiment IV

In this experiment, we analyze how well uncertainty is modeled in Bayesian NEAT and how it compares with Bayesian-MLP. Other benchmark methods do not admit this comparison as they are not stochastic models. In order to measure uncertainty, we first carry out a qualitative study on individual examples and later offer a more comprehensive study for all datasets considered. In the latter part, the dispersion in the predictive distribution is considered as a proxy for uncertainty estimation.

4.5.2 Results of Experiment IV

Figures 4.3 and 4.4 show, respectively, an example of uncertain and certain output of a stochastic network for an example in the Wine dataset, generated by Bayesian-NEAT. While the first 3 plots on the left show the probability distribution for each class ($P(y = 0|x, w)$, $P(y = 1|x, w)$ and $P(y = 2|x, w)$), the last plot shows a 2D representation of the test set (built using Principal Component Analysis) with all examples and their class. Furthermore, it shows in green, the 2D position of the specific example that we are showing. It is worth mentioning that the estimator was able to correctly predict both samples. However, with the first one, it was more uncertain.

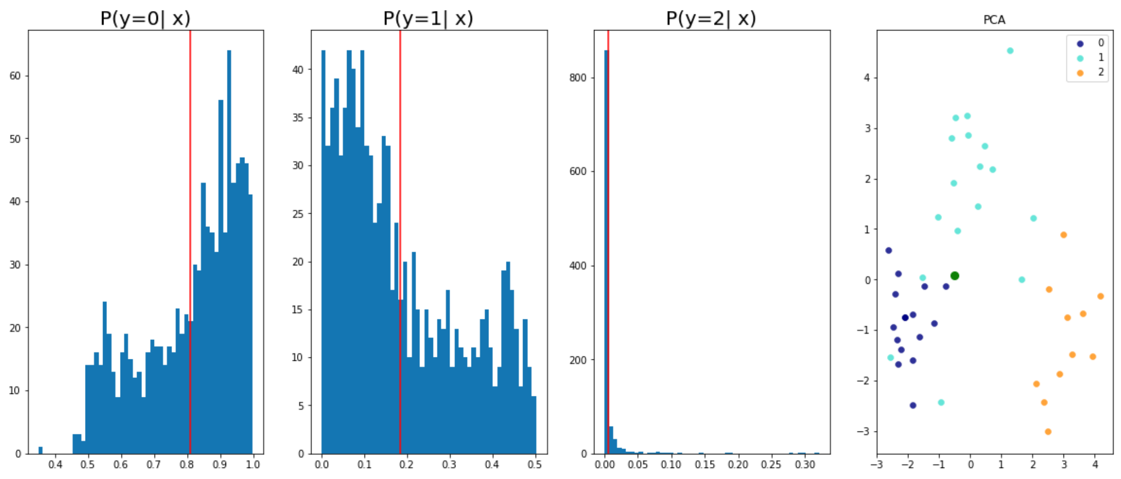


Figure 4.3: *Uncertain Class Prediction Distributions for a Wine Example.*

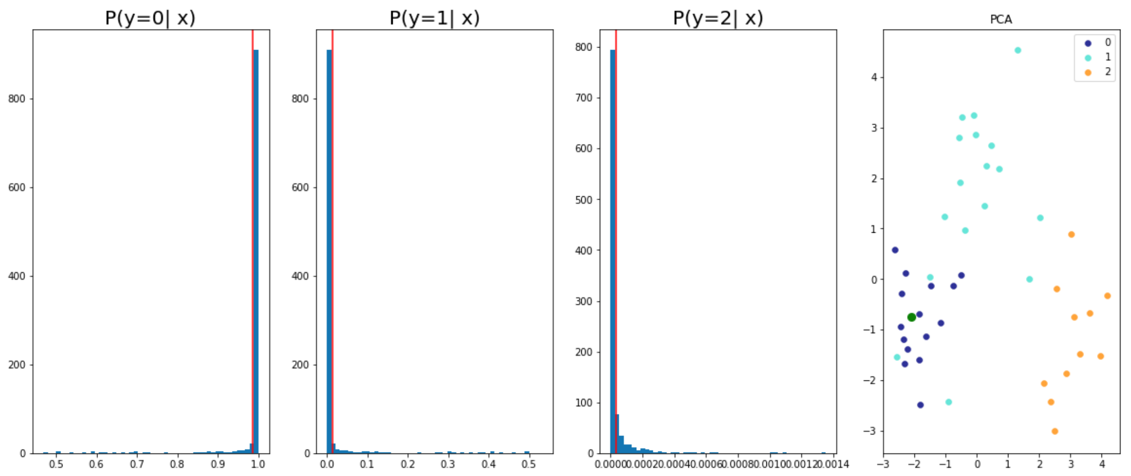


Figure 4.4: *Certain Class Prediction Distributions for a Wine Example.*

Figure 4.5 shows how F1 Score changes as we remove more uncertain examples. That is, as we move from left to right in the x-axis, the most uncertain examples are removed. The most uncertain example is the one with the highest standard deviation in the predictive probability distribution. While at the left of the x-axis, all examples in the test set are used, the more we move towards the right, only the most certain predictions are kept. Therefore, if the model is able to measure uncertainty, F1 Score needs to monotonically increase.

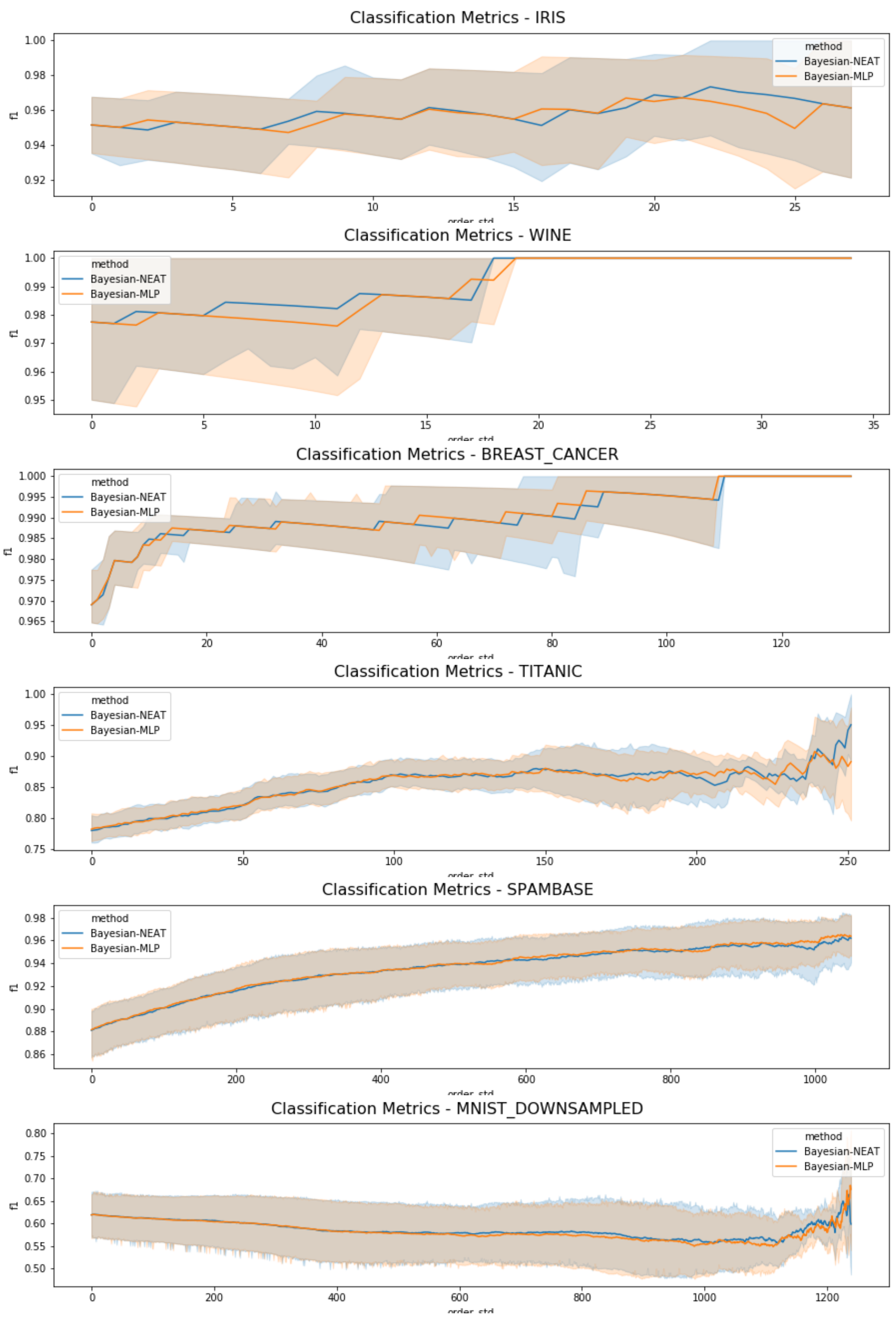


Figure 4.5: *Prediction Distributions for a Wine Example.*

4.5.3 Discussion of Experiment IV

Figures 4.3 and 4.4 show, respectively, an example of uncertain and certain output of a stochastic network for an example in the Wine dataset, generated by Bayesian-NEAT. While the first 3 plots on the left show the probability distribution for each class ($P(y = 0|x, w)$, $P(y = 1|x, w)$ and $P(y = 2|x, w)$), the last plot shows a 2D representation of the test set (built using Principal Component Analysis) with all examples and their class. Furthermore, the mean of each class' distribution is shown as the red vertical line. Finally, in green, these Figures show the 2D position of the specific example. It is worth mentioning that the estimator was able to correctly predict both samples. However, with the first one it was more uncertain.

As we can see in the PCA plot of Figure 4.3 the example is actually in the frontier between class 0 and 1. That makes the estimation more uncertain between those 2 classes. As we can see, it assigns very low probability to class number 2 a very little uncertainty. That is, is very certain that is not class 2. Looking at Figure 4.4 we can still see that both classes 0 and 1 are possible but the estimator is much more sure about being class 0.

Because the previous figures are just individual data examples, we study how we can use this uncertainty measure to evaluate the model. In order to do that we have created the graphs in Figure 4.5. As we can see, the F1 Score increases as we move from left to right for all datasets besides Iris and MNIST. This indicates that the model is actually modeling the uncertainty with some degree of correctness. That is, ideally the curve would reach 1.0 quickly. However, despite the trend is actually positive, there are actually some noise in those curves that indicate that is not able to always model the uncertainty.

On the other hand, we barely see the positive trend in the Iris or MNIST dataset. On reason to explain what happens in Iris is that we might not have enough data to actually see this effect since the F1 score is already high (around 0.95). In the case of MNIST, it has enough data. However, as shown in previous experiments, learning in MNIST might not be good enough to provide good uncertainty estimates.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this work, we have introduced and validated the algorithm Bayesian-NEAT to generate both architecture and weights in Bayesian Neural Networks in the context of small and medium size Supervised Learning tasks. Moreover, we have proved that it can actually provide a measure of uncertainty for each independent data example. This is the main contribution of this work. However, there are others:

1. We have compared Bayesian-NEAT against proper benchmarks showing that it achieves competitive performance in most datasets. In this regard, we have seen that for high multi-class problems, the algorithm does not match traditional Neural Networks. We think that having more compute power and running for longer generations could help.
2. We have provided a use-case, mislabeled data, where the proposed algorithm is more robust than standard Neural Networks. We think that the fact that Bayesian-NEAT creates simpler models and has stochastic nature, help to better generalize in the presence of noise. That is, both characteristics act as model regularization mechanisms.
3. We have designed and implemented a conversion or mapping between the

Genotype and a feed-forward Bayesian Network in a Deep Learning framework for the first time. This contribution has the following implications:

   (a) The final Bayesian Neural Network can run in both CPU and GPU. While being able to run on CPU allows us to parallelize fitness evaluation for the population during the evolutionary search, being able to run on GPU allow for faster predictions and fine-tuning.

   (b) We were able to leverage Automatic Differentiation tools provided by the Deep Learning framework to apply the final fine-tuning stage.

4. We have designed and implemented a software system that facilitates experimentation and research on the topic.

Finally, based on the contributions explained above, we can claim that the objectives of the research are fulfilled in high degree. Issues encountered in higher dimensional problems are discussed as future research directions.

5.2 Future Work

Despite the algorithm was validated for low dimension problems, we saw that it did not perform well for classification problems with high number of classes. In order to be able to extend the use of the algorithm to high dimensional problems, we proposed several research directions:

1. Indirect Encoding. The current approach follows a direct encoding scheme where each gene of the genotype represents a part of the network. However, direct encoding does not scale well as the search space increases. For higher search spaces, the literature recommends indirect encoding schemes.

2. Better search methods. In this work, we have not researched alternative search methods. However, we think that further research on this could help improve results.

3. Scale computation resources. Having more computation resources or larger running times could help obtain better results.

Regarding the regularization effect that we see with Bayesian Methods and NEAT it would be interesting to compare them to common regularization mechanism such as drop-out and L1 and L2 regularization.

Bibliography

[1] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. *Proceedings of the International Conference of Machine Learning*, 2013.

[2] D. A. Berry and D. A. Berry. *Statistics: a Bayesian perspective*. Number 04; QA279. 5, B4. Duxbury Press Belmont, CA, 1996.

[3] C. M. Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[4] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe. Variational inference: A review for statisticians. *Journal of the American Statistical Association*, 112(518):859–877, 2017.

[5] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra. Weight uncertainty in neural networks. *International Conference on Machine Learning*, 2015.

[6] M. W. Dusenberry, G. Jerfel, Y. Wen, Y.-a. Ma, J. Snoek, K. Heller, B. Lakshminarayanan, and D. Tran. Efficient and scalable bayesian neural nets with rank-1 factors. *arXiv preprint arXiv:2005.07186*, 2020.

[7] A. E. Eiben, J. E. Smith, et al. *Introduction to Evolutionary Computing*, volume 53. Springer, 2003.

[8] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 2019.

[9] Frank E. Harrell Jr., Thomas Cason. Titanic dataset, 2017. Data retrieved from OpenML Repository, https://www.openml.org/d/40945.

[10] A. Gaier and D. Ha. Weight agnostic neural networks. In *Advances in Neural Information Processing Systems*, pages 5365–5379, 2019.

[11] Y. Gal and Z. Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. *International Conference on Machine Learning*, 2016.

[12] A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019.

[13] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[14] S. Gray, A. Radford, and D. P. Kingma. Gpu kernels for block-sparse weights. *arXiv:1711.09224*, 3, 2017.

[15] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger. On calibration of modern neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1321–1330. JMLR. org, 2017.

[16] X. He, K. Zhao, and X. Chu. Automl: A survey of the state-of-the-art. *arXiv:1908.00709*, 2019.

[17] Y. Kwon, J.-H. Won, B. J. Kim, and M. C. Paik. Uncertainty quantification using bayesian neural networks in classification: Application to ischemic stroke lesion segmentation. 2018.

[18] B. Lakshminarayanan, A. Pritzel, and C. Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *Advances in Neural Information Processing Systems*, pages 6402–6413, 2017.

[19] Mark Hopkins, Erik Reeber, George Forman, Jaap Suermondt. Spambase dataset, 1998. Data retrieved from UCI Machine Learning Repository, https://archive.ics.uci.edu/ml/datasets/spambase.

[20] R. C. Martin. *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall Press, 2017.

[21] Matjaz Zwitter, Milan Soklic. Breast cancer dataset, 1988. Data retrieved from UCI Machine Learning Repository, https://archive.ics.uci.edu/ml/datasets/breast+cancer.

[22] K. Osawa, S. Swaroop, A. Jain, R. Eschenhagen, R. E. Turner, R. Yokota, and M. E. Khan. Practical deep learning with bayesian principles. *Advances in Neural Information Processing Systems*, 2019.

[23] R.A. Fisher. Iris dataset, 1988. Data retrieved from UCI Machine Learning Repository, https://archive.ics.uci.edu/ml/datasets/iris.

[24] K. Shridhar, F. Laumann, A. Llopart Maurin, M. Olsen, and M. Liwicki. Bayesian convolutional neural networks with variational inference. *arXiv:1806.05978*, 2018.

[25] J. S. Speagle. A conceptual introduction to markov chain monte carlo methods. *arXiv:1909.12313*, 2019.

[26] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[27] K. O. Stanley. *Efficient evolution of neural networks through complexification*. PhD thesis, 2004.

[28] K. O. Stanley. Compositional pattern producing networks: A novel abstraction of development. *Genetic programming and evolvable machines*, 8(2):131–162, 2007.

[29] K. O. Stanley, D. B. D'Ambrosio, and J. Gauci. A hypercube-based encoding for evolving large-scale neural networks. *Artificial life*, 15(2):185–212, 2009.

[30] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[31] Stefan Aeberhard. Wine dataset, 1992. Data retrieved from UCI Machine Learning Repository, https://archive.ics.uci.edu/ml/datasets/wine.

[32] Yann LeCun, Corinna Cortes, Chris Burges. Mnist dataset, 1998. Data retrieved from http://yann.lecun.com/exdb/mnist/.

Appendix A

Code Repository

The code for the entire project is hosted in GitHub at https://github.com/AlbertoCastelo/Neuro-Evolution-BNN. In order to use it, it has to be cloned to a computer or server. Then, the user can spin up a JupyterLab environment typing the following command in the command line at the root of the project.

```
$ git clone https://github.com/AlbertoCastelo/Neuro-Evolution-BNN.git
$ make jupyter
```

This will automatically install all dependencies you need as long as the computer has git, Docker and Docker-Compose installed. Then, it will offer a JupyterLab server.

Appendix B

Default Parameters

Table B.1: *Bayesian-NEAT Parameters.*

Parameter Name	Value	Description
parallel_evaluation	True	Fitness evaluation is parallelized
n_processes	6	How many cores should be used during fitness evaluation
is_fine_tuning	True	Apply final fine-tuning stage
epochs_fine_tuning	2000	Max number of epochs to run in fine-tuning
is_discrete	False	Type of parameter mutation
train_percentage	0.75	Ratio of dataset used for training
beta	0.0001	Multi-objective weighting factor
fix_std	False	Fix σ mutation and/or training
fix_architecture	False	Fix architecture mutation
is_initial_fully_connected	False	Initialization creates a fully connected network
initial_nodes_sample	50	Number of connections with input layer on initialization
dataset_random_state	42	Seed used to generate data

Parameter Name	Value	Description
noise	0.0	Gaussian noise added to independent variables
label_noise	0.0	Probability of misclassified example
n_generations	150	Number of generations
generation_fix_architecture	150	Stop architecture mutation after it
pop_size	50	Population size
node_activation	tanh	Activation function
n_samples	100	Number of samples taken
n_species	5	Number of species
max_stagnation	30	Max generations without improvement
elitism	2	Elitism per specie
min_species_size	2	Minimum of individuals per specie
mutate_power	0.5	σ used in mutation
mutate_rate	0.8	Probability of mutating a parameter
architecture_mutation_power	1	Number of architecture mutations
node_add$_{prob}$	0.9	Probability of adding a node
node_delete_prob	0.1	Probability of removing a node
connection_add_prob	0.9	Probability of adding a connection
connection_delete_prob	0.1	Probability of removing a connection
mean_init_mean	0.0	Parameter μ initialization μ
mean_init_std	1.0	Parameter μ initialization σ
mean_max_value	10.0	Parameter μ max value
mean_min_value	-10.0	Parameter μ min value
mean_prior	0.0	Parameter μ prior
std_init_mean	0.001	Parameter σ initialization μ
std_init_std	0.0	Parameter σ initialization σ
std_max_value	2.0	Parameter σ max value
std_min_value	0.0	Parameter σ min value
std_prior	1.0	Parameter σ prior

Appendix C

Other Things Tried

C.1 Pruning Networks

Start from a pre-trained network and use Bayesian-NEAT to prune the network. This works by assigning greater probabilities to parameters *delete_node_prob* and *delete_connection_prob* than to *add_node_prob* and *add_connection_prob*. The algorithm was able to prune the network with some some loss in performance. However it is tricky to find the balance between Log-Likelihood and Complexity Cost.

C.2 Fine-Tuning using Evolutionary Methods

For the final Fine-Tuning stage, first we tried to solve it using a fully evolutionary approach by:

1. Run Bayesian-NEAT for n generations. In this stage both architecture and weights are evolved.
2. Activate $fix\_architecture$ flag so that architecture is not mutated.
3. Run Bayesian-NEAT for n_{ft} more generations. In this stage only weights are evolved.

Despite we did not see a huge performance increase, this could be a possible future direction if we future iterations do not want to rely on having a differentiable fitness function.

C.3 Mutation using Backpropagation

The typical mutation scheme for parameters consist on modifying the parameters based on a sample from normal distribution. In order to update parameters in a good direction, we tested using backpropagation to train the parameters for a few epochs (typically 5) before continuing with the evolutionary phase. The issue with this approach was that it became infeasible due to the time it take to run because for each genome of the population it had to:

1. Convert genome to network.
2. Train network for 5 epochs.
3. Convert network back to genome.

All these steps added a lot of overhead.

C.4 Add Noise to Attributes

Before trying experiment III, we tried injecting noise to the attributes. In general, Bayesian methods and NEAT behave better as they regularized better than the standard MLP. We did not pursue further because the use-case in Experiment III showed more potential.