TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Holistic Auto-ML on medical data (fibrillation data)

Master Thesis

by

Aditya Iyengar

August 7, 2020

Technische Universität Kaiserslautern,
Department of Computer Science,
67653 Kaiserslautern,
Germany

Examiner: Prof. Dr. Andreas Dengel
Dr. Jens Krüger

Declaration

I, Aditya Iyengar, declare that this thesis titled, "Holistic Auto-ML on medical data (fibrillation data) " and the work presented in it are my own. The sources and additives used have been marked in the text and are exhaustively given in the bibliography.

Kaiserslautern - 7.8.2020

_____

Aditya Iyengar

Acknowledgment

I would like to thank Prof. Dr. Andreas Dengel for providing me with the opportunity to write my thesis in deep learning. I would also like to thank him for enabling the smooth completion of the thesis.
I am also deeply indebted to my supervisor Dr. Jens Krüger for his immeasurable support throughout the thesis. I would like to thank him for giving me the freedom to explore this topic and implement the ideas. This thesis would have been practically impossible without his interminable support. I shall ever cherish the time spent working with him on this thesis and the various topics we discussed over time. Ever grateful for his unfailing support during the tenure.
I would also like to thank my peers at Fraunhofer ITWM, for all the technical assistance during these times. I would like to thank Dominik Loroch and Nico Weber, for their valuable help, for assistance and guidance for the various topics.
Lastly, I'd like to thank my parents (Prasanna Iyengar and Y S Savithri), my fiancé (Swathi) and my siblings. This would not have been possible without their constant support and encouragement.

Acronyms

NN	**Neural Network**
ANN	**Artificial Neural Network**
CNN	**Convolutional Neural Network**
DNN	**Deep Neural Network**
ML	**Machine Learning**
ECG	**Electrocardiogram**
AF	**Atrial Fibrillation**
NAS	**Neural Architecture Search**
FPGA	**Field Programmable Gate Array**
GPU	**Graphical Processing Unit**
CPU	**Central Processing Unit**
ReLU	**Rectified Linear Unit**
MR	**Mutation rate**
BWN	**Best Weighted Network**
DL	**Deep Learning**
ConV	**Convolutional layer**
ASIC	**Application specific integrated circuit**
I/O	**Input/Output**
FLOP	**Floating point operations**
API	**Application programming interface**
FP	**Floating point**

Abstract

In recent years, the complexity of applications where Neural Networks (NNs) can be used have evolved. NNs have become important in field of electrocardiography to detect abnormal heart rhythms. There has also been a shift from designing deep neural networks (DNNs) manually for such problems to automating this design process. The field related to this automated approach is called AutoML. AutoML allows you to provide the training data as input and receive an optimized model as output.

This thesis explores the design of an AutoML framework for time series electrocardiogram (ECG) data to detect atrial fibrillation (AF). The framework also concentrates on building hardware specific DNN models trying to reduce the number of parameters and maintaining good accuracy as the optimization criteria. It uses the concept of Neural Architecture Search (NAS) based on genetic programming to search for the optimal DNN.

Various selection policies for NAS are developed in order to explore multiple DNN topologies. Based on the selection criteria and measures, mutation of one or more selected model takes place which leads to the generation of new models. The mutation rate is dynamically adjusted for generating good models.

Experiments are performed to compare the results from different selection policies. The results are aggregated in order to verify the effectiveness of using multi-metric NAS. Using an appropriate mutation rate and weights for one of the selection policy, we obtain a model with only 8,200 parameters (an 8.8x improvement to the original model) producing very good accuracy for this classification task.

Contents

List of Figures

1 Introduction

Machine Learning (ML) is an area of computer science that provides the systems the ability to learn patterns and make deductions from it. Machine learning focuses on the development of computer programs (or algorithms) that can access data and use this data to learn salient features from it. ML algorithms are often categorized as supervised/unsupervised algorithms and reinforcement learning algorithms.

Supervised algorithms learn from labelled training data and build a model out of it. The labelled data is a pair, consisting of an input value and a desired output value. Supervised algorithms analyse the training data and produce the output values. These output values are then used to map onto the desired output values to determine the performance of the model. In contrast, unsupervised learning looks for patterns in the dataset with minimum human supervision. The dataset does not contain any labels and the algorithms instead look for patterns.

ML is also closely related to, and used in computational statics and data mining. Statistics and data mining can provide various analytical insights - descriptive, diagnostic, predictive, and prescriptive.

Descriptive analytics is used to view existing data statistically which informs us about the past. This can provide the context to the dataset. Whereas, diagnostic analytics is used for cause analysis in a problem. This again is a pattern detection problem wherein ML can play an influential role. Predictive analytics is used for predicting an event from historical data. The historical data is used as an input to a machine learning model, which predicts the future event in a probabilistic method. Prescriptive analytics takes predictive analytics to a new step and suggests various courses of action and the potential implication of each action.

For ML, one can also distinguish between training and inference phases. The dataset is divided accordingly into training, testing, and a validation dataset. In the training phase, the model tries to learn from the training dataset. If supervised learning is used, then the model learns to differentiate between various classes of data. The model predicts the class label and compares them to the actual labels. A validation dataset can be used to tune this model for producing better results. In the infer-

ence phase, this trained model is used to infer and predict for the testing dataset. This is usually the stage where this model is deployed to predict real-world data.

ML is used in a wide variety of applications. From virtual personal assistants to more industrial applications such as email filtering and computer vision, this field has improved the efficiency and consistency with which quality results are delivered.

Spurred by such high quality results and advances in processing power, memory, storage and an unprecedented wealth of data, the applications of ML are swiftly infiltrating many areas of the healthcare industry as well. From diagnosis and prognosis to drug development and epidemiology, machine learning has the potential to transform the medical landscape [Deo15] [Bea+19].

The increase in knowledge and understanding of diseases from numerous sources has provided a large source of information. This information can be used to tackle increasingly complex clinical tasks, often with astonishing success. A simple pipeline showcasing the use of such information is shown in Figure 1.1.

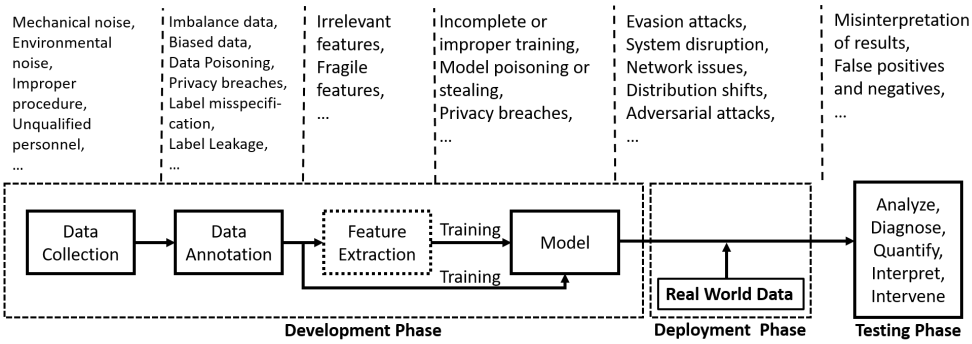


Figure 1.1: *A simple pipeline for clinical care [Qay+20]*

The pipeline shows the development phase, where the data is collected through various sources and annotated. The annotated data is required in the training process for the model. The model as such, extracts features and does the classification task. This model can then be used to classify unknown data which becomes helpful to analyze, diagnose, and intervene to speed up the clinical process.

Machine learning has also become important in the medical field of electrocardiography, which deals in the process of producing an electrocardiogram (ECG) for heartbeats. ECG is a recording of the voltage of the electrical activity of the heart using electrodes, versus time. The

normal rhythm of the heart is called sinus rhythm and this is necessary for normal electrical activity of the heart (Figure 1.2).

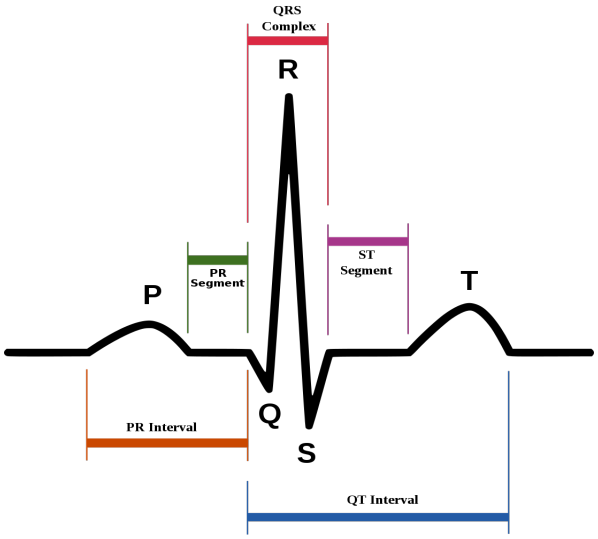


Figure 1.2: *ECG of a heart in normal sinus rhythm [Wik20a]*

Changes in the normal ECG pattern are called cardiac abnormalities. The cases of these cardiac rhythm disturbances are called atrial fibrillation (AF) and ventricular tachycardia. "Tachyarrhythmia characterized by predominantly uncoordinated atrial activation with consequent deterioration of atrial mechanical function" is the definition coined by the American College of Cardiology (ACC), the American Heart Association (AHA) and the European Society of Cardiology (ESC) [Fus+01] for atrial fibrillation.

Among the many cardiac disturbances, atrial fibrillation (AF) is the most common cardiac arrhythmia [Smo17] [Eur+10]. AF is associated with a high mortality rate and its detection is challenging as it may be episodic. Early detection of AF can be clinically significant to mitigate the risks caused by it.

Deep learning (DL), a subclass of machine learning algorithms has shown high accuracy in performing pattern recognition for speech [SSB14], image recognition [Rus+15], in detection of diabetic retinopathy [Gul+16] and skin cancer [Est+17]. Convolutional neural networks (CNNs) and Recursive neural networks (RNNs), a subset of DL, have achieved good results for ECG classification (Hannun et. al [Han+19]) and can be used to detect AF.

1.1 Problem Statement

State of the art deep neural networks (DNNs) for image classification commonly have a large number of parameters. For example, Inception-v3 [Sze+16] and ResNeXt-50 [Xie+17] architectures have about 25 million parameters, with the size of the models greater than 90MB. These are considerably large models with very high training times especially on conventional x86 central processing units (CPUs).

To overcome this high training times, specialized hardware such as graphical processing units (GPUs), application specific integrated circuit (ASIC) and field programmable gate arrays (FPGAs) are designed. GPUs are commonly used hardware with a high parallel structure to enable high throughput. But because of the flexibility and ability to be re-configurable, the use of FPGAs is becoming increasingly common to train deep neural networks. FPGAs are also known to be more energy efficient when the classification pipelines grow [Qas+19].

With the development of such specialized hardware, research has been carried out into the field of energy efficiency. Energy-efficient hardware design requires a conscious consideration of the entire design space at various design levels, i.e at application, algorithm, architecture, and technology levels. DNN models that are fairly large in size can become an obstacle to specialized hardware such as FPGAs and embedded devices. On such devices, DNNs should be optimized with regards to minimizing the number of operations required to perform a single classification action. This directly translates into the amount of energy consumed by the device.

Designing a new topology for such problems, with specific network constraints can be a difficult process. With the classical design strategy of trial and error, this is a cumbersome task as the search space is highly dimensional. A purely heuristic approach to solve this problem can be a time-consuming process. As such, evolutionary methods with genetic algorithms (AutoML approach) can be used to find an optimal network topology accommodating various network constraints.

Many network topologies can be analysed for the suitability of the problem whereas the optimal network structure depends on the goal to be achieved. The balance between multiple optimization criteria such as the number of parameters (which can be used to track energy consumption), the accuracy for the classification of the task, computational complexity, memory bandwidth of the hardware, and the memory requirement become essential. This approach with multiple objective functions to

find a DNN topology by evolutionary means is the starting point for the project.

1.2 Our Contribution

Keeping a holistic view of the problem, the main contribution of my work are as follows:

1. Analyze ECG data to examine if preprocessing is essential for the dataset.
2. Develop baseline models (convolutional neural networks) to classify ECG data.
3. Develop an AutoML toolbox for topology search using evolution based methods with genetic algorithms for ECG data.
4. Add multiple selection strategies.
5. Analyze various selection strategies and mutation policies.

This work is integrated into a project which deals with the development of a generic AutoML toolbox. The optimization and implementation methods used in this project should be generic enough to be used for a wide variety of DL applications.

1.3 Structure of the report

The report is divided into 7 chapters. After the introduction, the theoretical background of the various topics covered in the project is explained in chapter 2. This includes a theoretical explanation of convolutional neural networks (CNNs), deep neural networks (DNNs), DNNs for specialized hardware such as FPGAs and GPUs, the concepts of neural architecture search (NAS) and genetic algorithms for AutoML.

In chapter 3, we speak about the various standard and state of the art approaches which have been attempted before for ECG signal classification in section 3.1, deep neural networks for specialized hardware in section 3.2, AutoML and neural architecture search approaches in section 3.3.

In chapter 4, we speak about data exploration. The description of the data in the dataset is pointed out in section 4.2. The dataset is explored

using various visualisation techniques in section 4.3 and a small analysis on the dataset in mentioned in the section 4.4 of that chapter. The seed models are discussed in section 4.5.

In chapter 5, we speak about the proposed approach for Auto-ML. The NAS pipeline used for classification is discussed in section 5.1. Encoding of layers is discussed in subsection 5.1.1 where as in subsection 5.1.2 and subsection 5.1.3, the search space and fitness evaluation metrics are discussed respectively. The subsection 5.1.4 speaks about the various selection strategies used in the thesis.

In chapter 6, we discuss the results obtained from the various experiments conducted. The results in section 6.1 are for the seed model where as section 6.2 are for the NAS experiments. In chapter 7, we summarize the present work and discuss about the future work.

2 Theoretical Background

2.1 Artificial Neural Networks

A neural network is inspired from the working of "neurons". Neuron, a biological term, is the fundamental unit of the nervous system and is responsible for transmission of sensory input, sending motor commands to muscles and relaying electrical signals between the different parts of the body. These electrical signals are called action potential and signal the "activation" of a neuron if the potential is above threshold value. The action potential and consequent transmitter release allow the neurons to communicate with other neurons.

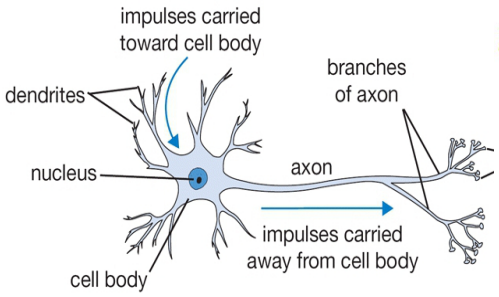


Figure 2.1: *Biological Neuron [Kar16]*

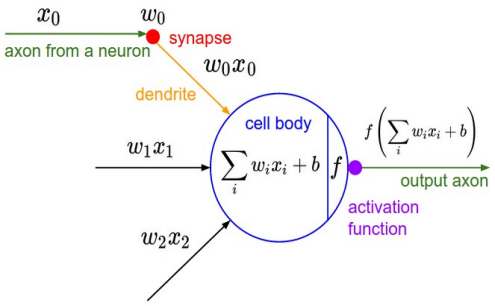


Figure 2.2: *Mathematical model of the neuron [Kar16]*

In the context of Artificial Neural Networks(ANN), a neuron is nothing but a set of inputs, a set of weights, and an activation function. The connections between the neurons are modelled as weights and the activation function is responsible for the magnitude of the output (as shown in Figure 2.1 and Figure 2.2).

An artificial neural network consists of at-least 3 layers (shown in 2.3) - the input layer, the hidden layers and the output layer. The initial data to the network is provided through the input layer, the computations are performed and representations are learned in the hidden layers and the output layer produces the result for the given inputs. In general, the formula for computation of activation for a n^{th} layer in a multi-layered neural network is done through:

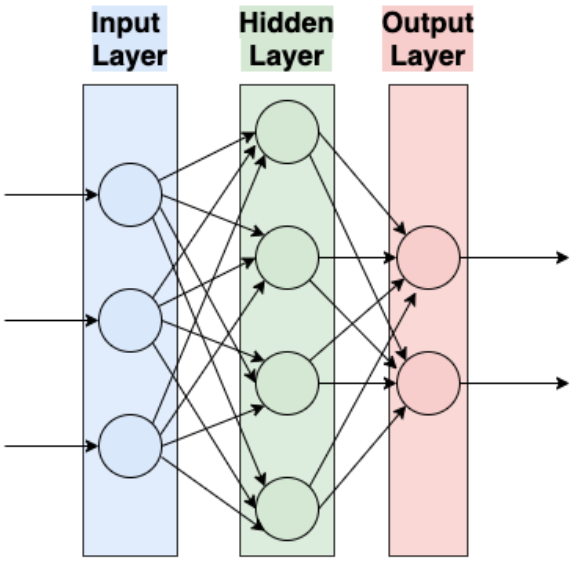


Figure 2.3: *A simple neural network*

$$a^n = \sigma(W^n \,.\, \sigma(W^{n-1} \,.\, \sigma(\,.....\, \sigma(W^1 \,.\, a^0) \,....\,)))$$

where σ is the activation function, W is the combined weight in a layer and a is the activation value at a layer.

Various linear functions as well as non-linear functions such as Sigmoid, tanh (hyperbolic tangent function), ReLU (Rectified Linear Unit), and Leaky ReLU can be used for activation. The most important features for the activation function are:

- Non-linearity of the activation function
- The function should be differentiable

It easy for the model to generalize or adapt with a variety of data and to differentiate between the output by using non-linear activation functions. "Sigmoid" functions (Fig 2.4) are used in models where we have to predict the probability as an output. This function has a "S"-shaped curve or sigmoid curve. The formula for the sigmoid function is:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.1}$$

The output from this activation function can then be rounded off to produce binary output (0 or 1). "tanh" is also a logistic function like the sigmoid but the range of the function is (-1 to 1).

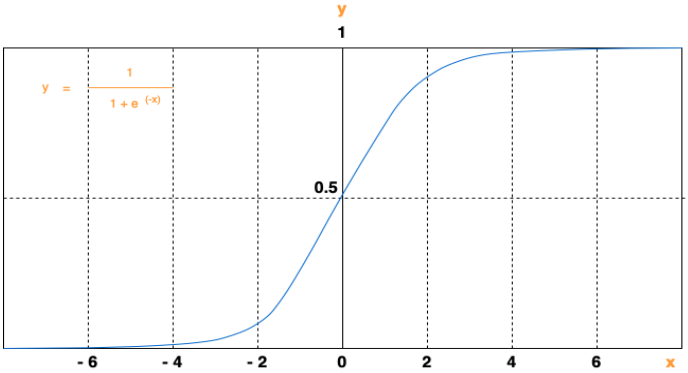


Figure 2.4: *Sigmoid activation function*

ReLU (Fig 2.5) is one of the most used activation function whose range is [0 to infinity). Using this function, the negative values become zero. This is computationally very cheap and and sparsely activated. This activation function also has a better gradient propagation with less vanishing gradient problem. The formula for the ReLU function is:

$$y = max(0, x) \tag{2.2}$$

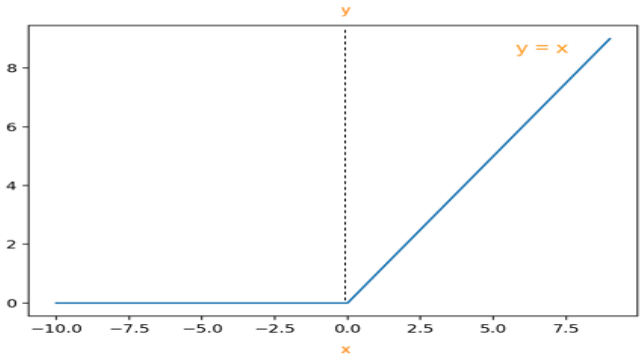


Figure 2.5: *ReLU activation function*

2.2 Convolutional Neural Network (CNN)

One of the first self-learning neural network model was developed by Fukushima, called "Neocognitron"[Fuk80]. The Neocognitron had an extended hierarchy structure and was inspired by the visual nervous system model proposed by Hubel and Wisel in the early '60s. Fukushima talks about a neural network model in his paper that performs self-learning for visual pattern recognition by unsupervised learning. Fukushima mentions the use of modular structures and cells which are connected by cascading and preceded by an input layer. He also mentions two types of layers - convolutional layers and downsampling layers. According to this paper, such types of networks could recognize a pattern with position-invariance.

If there is high position-invariance, as is the case in daily life pattern recognition problems, it becomes important to identify an appropriate set of features. If such features are identified, then the position of features become less important. As mentioned in the paper by LeCun et al. [LeC+99] Convolutional Neural Networks(CNNs) can be designed to perform such tasks. CNNs can be used to recognise patterns without explicitly segmenting the data. CNNs can thus be considered to be good feature extractors. This is typically the case in the field of computer vision [AK15], for time series data [Yan+15] and are also often deployed for the medical data [PKM17].

The convolution layer (ConV) is the nucleus of CNNs. The convolutional layer's parameters consist of a set of learnable filters, sometimes also called kernels. These filters have a small receptive field [Kar16]. During the forward pass, we convolve across the spatial position of the input volume and we compute the dot product between the entries of the filter and input at any given position. As we slide over the input, the filter produces a feature map that gives the responses of that filter at every spatial position. The network will thus become aware of those filters, which activates when features are detected.

In principle, we have multiple feature maps that learn different features. These feature maps are stacked together to produce the output volume. Thus, every entry in the output volume can be interpreted as an output of a neuron that looks at a small region in the input.

The other important layers of the CNN network are the pooling layers. A pooling layer is a non-linear method of downsampling. Pooling layers are important because downsampling does not change the input, thus incorporating translation invariance. It decreases the spatial dimensions,

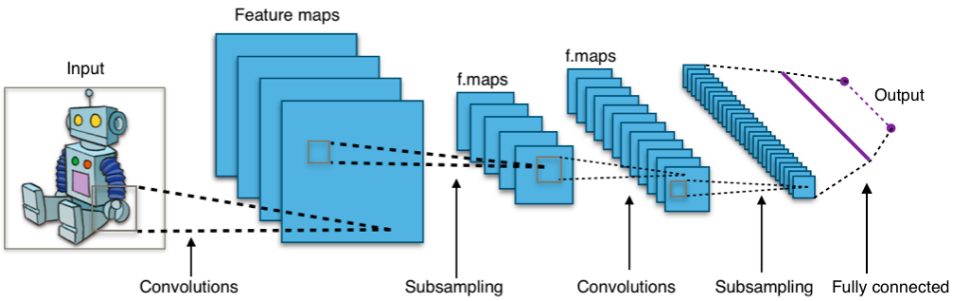


Figure 2.6: *CNN Architecture [Wik20b]*

thus reducing the number of parameters and the amount of computation in the network. Also, pooling computation is independent of the depth of the input. Most common pooling functions are max-pooling, average-pooling, $l_2 - norm$ pooling, region of interest(ROI), etc. A regular CNN architecture incorporating these layers is shown in Figure 2.6

Mathematically, if the data is single dimensional and if the ConV layer accepts an input of size $W \times D$ $(timesteps \times features)$, then the ConV layer produces an output of volume $W_1 \times D_1$, where W_1 and D_1 are calculated as follows:

$$W_1 = ((W - F + 2P)/S) + 1$$

$$D_1 = K$$

where K is the number of filters, F is the filter size or spatial extent, P is the padding and S is the stride used during convolution. The number of filters is typically a power of 2, whereas the padding is used to control the spatial dimension of the output volume. Stride controls how the filter convolves over the input volume.

The number of parameters for a layer is the count of "learnable" elements. These are learnt during training and the number is calculated as follows:

$$\text{Number of parameters} = F \times D \times D_1$$

These parameters are added for all the layers of the model in the end and are called trainable parameters.

2.3 Deep Neural Network (DNNs) for specialized hardware

Convolutional neural networks are a subclass of deep neural networks. Most modern deep learning networks use multiple layers progressively to extract higher-level features from the input. But these multi-layered neural networks are commonly large, complex and involve usage of a considerable amount of resources to train and evaluate the dataset. Such complex and intensive networks can result in very long processing times on standard x86 CPUs. Specialized hardware such as graphical processing units (GPUs), application specific integrated circuit (ASIC) and field programmable gate arrays (FPGAs) become essential for such tasks. Specialized hardware can improve the processing capabilities and performance of deep learning models. Computing tasks on specialized hardware would decrease the latency and increase throughput. As such, specialized hardware can be used to train the deep learning models. We speak about FPGAs and GPUs in the next section.

2.3.1 FPGAs vs GPUs

A GPU is a specialized hardware with a highly parallelized structure which makes them more efficient than general-purpose CPUs for heavy computing problems. GPUs have orders of magnitude more computational cores when compared to CPUs. Researchers from Stanford explored the usage of GPUs for training CNNs [Coa+13] and found that GPUs can be leveraged to perform more arithmetic operations per second to train large CNN networks. This would decrease the time for training and inference.

The other class of specialized hardware is the FPGAs. FPGAs contain an array of re-programmable logic blocks that can be configured to perform complex combinational operations with arbitrary precision. FPGAs are re-configurable integrated circuits and can be reprogrammed to implement different logic functions. This is different from instruction based hardware such as GPUs and CPUs, which are configured through software. An early publication of the usage of FPGAs was by Microsoft researchers. They used FPGAs to accelerate the search of "Bing" (which is using CNNs) by a factor of nearly 2X in their datacenters [Ovt+15].

FPGAs have further advantages when compared to GPUs and CPUs. GPUs are typically connected as a co-processor to CPUs. For any practical use case, it requires assistance from the CPU, whereas FPGAs can

be directly connected to any data source, such as a network interface, sensors, or other interfaces. FPGAs can be configured to directly access hardware I/O without the latency of internal bus structures which gives high bandwidth and low latency for the FPGAs. But, GPUs can execute thousand of arithmetic operations parallelly and brute-force throughput of GPUs should not be underestimated.

Many recent studies in the field of computer vision have shown that FPGAs are known to outperform GPUs in terms of energy consumed per frame as the complexity of the pipelines grow [Qas+19]. But considering the constrained on-chip resources on FPGAs, the DNN model must be optimized for energy efficiency. DNNs can be optimized with number of operations that would translate into energy required for a single classification action. But, it is not very easy to design the DNN models which would take into account these constraints of FPGAs. DNNs can be fairly large in size which can become an issue on FPGAs. This leads to a discussion of the next topic - AutoML.

2.4 AutoML

Traditionally, machine learning model development is resource-intensive and requires significant domain knowledge and effort. Then comes the process of experimenting with various models, preprocessing and data augmentation. The layers of the DNN would have to be modified to see if the model performance improves and then to find suitable hyper-parameters for the model. Thus, the success of DNNs depends on the proper configuration of its architecture and hyper-parameters. As such the entire process is tedious. Such problems led to the development of the concept of AutoML. AutoML has the goal to automatically search for an optimal DNN and automating the tasks across the end-to-end ML pipeline.

The process to automate DNN's configuration is rather a compelling thought. By this approach, we can find innovative DNN configurations that might not be possible by standard design techniques. We can also find configurations that are practical to be implemented on specialized hardware such as the FPGAs, and this becomes possible without the domain expertise.

2.4.1 Neural Architecture Search (NAS)

As discussed, finding a DNN model requires significant architectural engineering. It might not be convenient to design such a network, considering that it requires a fair amount of trial and error and experimentation which is time-consuming. This is where Neural Architecture Search (NAS) comes in. NAS is a subset of AutoML which can be used to find various DNN architectures.

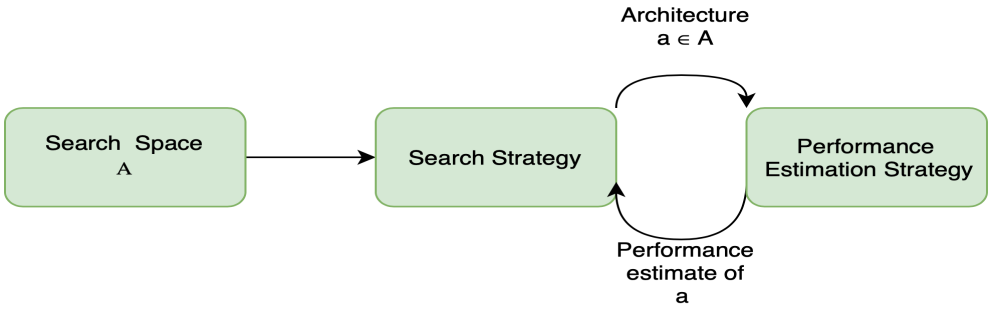


Figure 2.7: *NAS method illustration [EMH18b]*

There are essentially 3 steps which is used in NAS (as shown in 2.7):

- Search Space: This is the set of all feasible solutions for the given problem. It defines the set of architectures that can be represented in principle. In general, the search space is huge. It is essential that the size of the search space is reduced by incorporating prior knowledge of architectures well suited for a task even if this introduces human bias.

- Search Strategy: This step defines the approach on how to explore the search space. There are many search strategies that can be used such as the random search, the Bayesian optimization, reinforcement learning, and the evolutionary methods. While deciding the search strategy, exploration-exploitation trade-off to find well-performing architectures or to find models quickly needs to be considered.

- Performance estimation strategy: This defines the guiding criteria for the NAS to select a DNN as the best performing DNN. The simplest way of achieving this is to train the DNN for the training data and evaluate the performance on the validation data. Accuracy, for example, can be used as the guiding criteria in image recognition tasks.

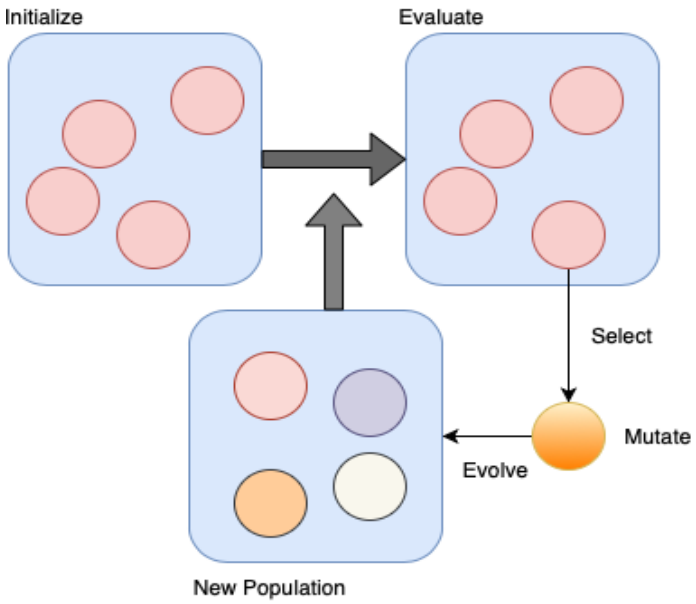


Figure 2.8: *A typical Genetic Evolution cycle*

2.4.2 Genetic Algorithm

Genetic programming or genetic algorithms are inspired by the biological evolutionary system and can be used for evolving DNNs in machine learning. Genetic Algorithms were developed by John Holland, his students and colleagues at the University of Michigan, most notably David E. Goldberg. Based on such foundations of neural evolution of DNN topologies, today, many groups such as Google's DeepMind, Uber Labs, etc. work on the genetic algorithms on a large scale.

A typical genetic cycle is shown in figure 2.8. Initially, while starting with the genetic algorithms, either a seed DNN can be used as a parent or a random set of DNNs can be generated. This is followed by a selection of the fittest DNN/DNNs for reproduction (crossover) and mutation according to a predefined fitness measure. Mutation involves the creation of new DNNs, which are also evaluated.

- **Initialize**: A seed DNN model or an initial set of DNN topologies is created at random. This is the initial set of a population of possible solutions that needs to be evaluated.

- **Evaluate**: This is the stage when the DNN topologies are evaluated. The evaluation is done on the performance estimation strategy. Different performance measures can be used. The most common one used for image recognition tasks is the accuracy of the

DNN topology.

- **Selection**: This is the stage in which DNN topologies are selected from a population for later breeding (mutation). The selection may take place through the evaluation of a fitness function for individual DNN topology. This kind of selection is called fitness-based selection. One (single parent) or more than one (multiple parents) DNNs can be selected for the process of mutation. Such a selection mechanism is very robust as better-performing DNNs have a chance of being selected for mutation. But such a selection process has the problem that it might lead to slower convergence for the selection of final DNN topology.

- **Mutate**: The process of producing the new topologies is called mutation. Mutation, in general, involves substituting some part of a DNN topology with another part. Mutation maintains the genetic diversity from one generation to another. After selecting the DNN for the mutation process, new DNN topologies are produced from the selected topology. The new DNN topologies belongs to the new population and is called a new generation. There are many methods to mutate and produce a new generation. This is a recursive process until a specified number of generations is produced.

  The goal of the genetic algorithms is to improve the performance of the DNN over generations. The selection process should select one or more new parent topologies. Mutation also follows certain policies such as random mutation, smart and selective mutation, etc. After mutation, the performance of a DNN may change entirely from the previous DNN. Depending on the kind of mutation, members of the new generation are generally more fit than the previous generation, although this is not always true.

  A simple example of mutation can be demonstrated as below. If we consider a set of bits as the genes of a parent, then flipping just one of the bit will result in a new child. The flipping of the bit can be at any random position and the resulting child might produce a completely different solution. If we were to find an analogy between the set of bits with that of a DNN, then the bits can be compared to layers of the DNN.

Parent:

1 0 0 1 0 0 ***0*** 1

↓ mutation

1 0 0 1 0 0 ***1*** 1

Child

Genetic algorithms have various advantages. These algorithms have very good parallel computing capabilities. The siblings in the new generation can be trained in parallel as there is no dependency between those. Genetic algorithms are also known to optimize both continuous and discrete functions and can provide a list of good solutions instead of only one. Genetic algorithms are also known to always produce a solution.

From a mathematical perspective also, genetic algorithms perform very well. Typical machine learning methods provide solutions to a problem that are locally optimal. This means there can be an improvement to the solution but there is a tendency to not scrutinize it. Since genetic algorithms are iterative, they tend to explore more than one solution and can provide a global solution (as shown in 2.9).

Genetic algorithms can be pretty noisy as they generate many bad networks. But this can be used as an advantage for exploration. This can be achieved by controlling the extent of the mutation. For a network which produces good results, the extent of mutation should be low, where as if a network produces bad results, then mutation should be high.

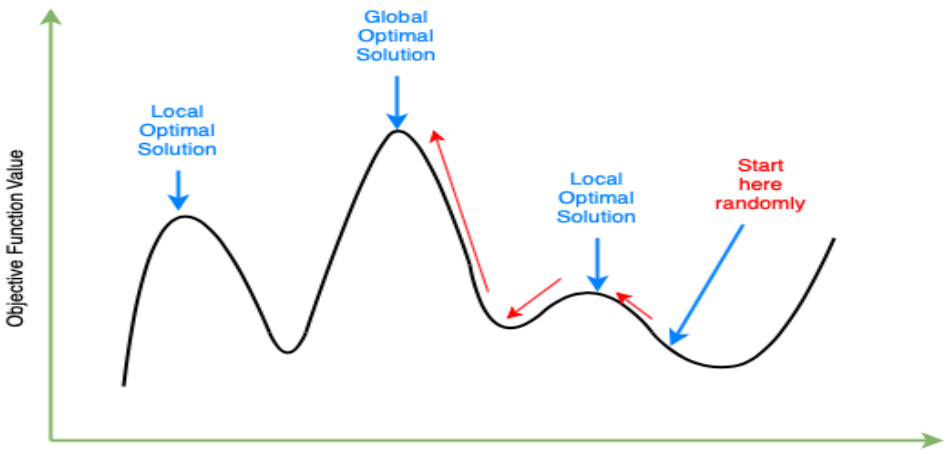


Figure 2.9: *Genetic Algorithm methodology*

2.4.3 Optimization Criteria

The performance of a neural network depends on several factors. Generally, the network architecture takes the spotlight for the performance, but optimization algorithms also play an important part. Mathematically speaking, optimization algorithms are used to minimize an objective function. The objective functions are dependent on the internal learnable parameters of the model such as the weights and the bias. These learnable parameters are used to compute the target values for the problem. The learnable parameters compute the output values and these parameters are updated in the direction of optimal solution i.e. minimizing the loss during the training process.

Today, the most general way to train a deep neural network model is by using gradient descent optimizer or by using one of its variants such as Adam, root mean square (RMS), AdaGrad, etc. The generic formula for gradient descent is:

$$\theta = \theta - \eta.\nabla J(\theta)$$

where η is the learning rate and $\nabla J(\theta)$ is the gradient of loss function $J(\theta)$, θ being the parameters.

In simple words, we use a metric and a method that maximizes or minimizes this metric for the given data. This metric could be accuracy or precision/recall (used for image and text data), or detection rate and false alarm rate (used for ECG data for arrhythmia detection). Our model, with learnable parameters, will try to search and optimize these parameters to maximize or minimize the metric for the data.

NAS is an iterative process. In every generation, we select one or more topologies with the best performance and continue this iterative process of producing a new generation until one or more stopping criteria is met. The stopping criteria can based not only on metrics such as accuracy, precision/recall, etc. but also on the parameters such as number of generations of evolution.

When building DNNs for hardware such as the FPGAs, it is also important to consider the resource consumption as an optimization criteria. Resource consumption can be interpreted as the number of floating-point operations (FLOPs) and the data access which would translate into energy consumed, as previously explained. This way, the number of parameters and number of FLOPs can also be considered as an optimization criteria.

3 Related Work and state of the art

There has been prior work in using AutoML and NAS for image classification tasks, as well as electrocardiogram (ECG) classification using deep neural networks (DNNs). There has also been research topics on using DNNs for specialised hardware. These previous works are discussed in this chapter.

3.1 Electrocardiogram Signal Classification for Atrial Fibrillation Detection using Deep Neural Network

ECG has become a fundamental tool for diagnosing a wide variety of arrhythmic abnormalities. Computer-aided interpretation of the ECG data has become increasingly important to serve as a crucial adjunct to clinical interpretation. Significant improvement in deep neural network algorithms to learn abstract, high-level representations of the medical data has also contributed to improvements in ECG interpretation.

There have been many contributions for developing DNN for ECG data. The 2017 PhysioNet/CinC Challenge [Cli+17] had encouraged participants to classify sinus rhythm (SN) or atrial fibrillation (AF), an alternative rhythm from a single short ECG lead recording (between 30 seconds and 60 seconds in length). This had led the participants to try and develop various neural network models for classification. A simple 16-layered convolutional neural network, where each layer would contain multiple sub layers - batch normalization, ReLU activation, dropouts, 1D convolution, and 1D average pool was developed by researchers in New Zealand [XSZ17]. Their novel neural architecture also contained many skip connections and demonstrated CNNs can perform efficient classification. A similar 16-layered CNN architecture was proposed by Yildirim, O. et al. [Yıl+18] for multi-class classification which was fast and efficient with high accuracy (91.33 %). Another architecture that contained long

short term memory (LSTM) units along with CNN layers to account for feature aggregation across time was proposed in this challenge [ZPT17] which paved the way to use of temporal analysis as well.

In more recent research [Han+19], the authors create a novel ECG dataset that underwent an expert annotation. They develop a comprehensive end to end DNN approach to perform the classification across a broad range of the most common and important ECG rhythm diagnoses. The 34-layered DNN was developed to accomplish the tasks of the entire ECG pipeline, that is feature extraction, feature selection/reduction, and classification. Their research also showed that not much preprocessing, such as Fourier or wavelet transforms, is required on the ECG data to achieve strong classification performance. Their findings demonstrate that with sufficient training data, using a DNN in such an end to end approach has the potential to improve the accuracy and prediction performance of interpretation of ECG data.

3.2 Deep Neural Network for specialized hardware

Recent works in the area of CNNs have produced state of the art neural networks models for image recognition, speech recognition, and time-series data. These models have improved the accuracy for both image recognition tasks and classification, but also have improved the predictions for time series data. There have also been research projects in recent years not only to improve neural network's performance on CPUs but also to develop specialized hardware such as GPUs, FPGAs, ASICs, etc for training neural networks to improve the network's performance in terms of training time and efficiency.

Vanhoucke et al. of Google [VSM11] suggested methods and tools to improve the performance of the neural networks on CPUs. For floating-point implementation, the idea to localize the data available to the CPU by distributing it to nearby memory locations would increase the CPU performance. Methods like loop unrolling, parallel accumulators which gives the compiler more freedom in pipe-lining operations, data layout optimization for Single Instruction Multiple Data(SIMD) instructions for low-level parallelization on CPUs improves CPU's performance. Despite this, current GPUs have better performance.

There has previous work done by researchers such as Pyakillya et al. on ECG classification by using GPU for training neural networks [PKM17].

Their work presented the idea that ECG classification results can be obtained by 1D convolutional networks with fully connected network (FCN) layers on time-series data with decent accuracy on the validation dataset. Alfaras et al. proposed a heartbeat classifier on Echo State Network (ESN) [ASO19] which also showed that computation times of a parallel computing architecture such as GPU outperforms a CPU in the classification of heartbeats using ECG data.

The other type of specialized hardware used is the FPGAs. Microsoft announced a project in 2014 that demonstrated that FPGAs can accelerate Bing Ranking by a factor of nearly 2x in their datacenters [Ovt+15]. The researchers leveraged the infrastructure of the FPGAs to develop a high throughput CNN-FPGA accelerator yielding a high-performance system that consumed considerably less power.

In more recent works by Nurvitadhi et al. [Nur+17], the authors evaluated the emerging DNN algorithms such as ResNet, VGG, etc. on various generations of GPUs (NVIDIA Titan X Pascal) and FPGAs (i.e., Intel Arria 10 GX 1150 and Intel Stratix 10 2800). The experiments show that FPGAs are extremely customizable and the current trends in DNN favour the FPGA platform as they offer superior energy efficiency.

Shawahna et al. talk about the challenges of FPGA-based implementation of deep learning networks [SSE18]. CNNs have a requirement of a significant amount of storage, external bandwidth, and computational resources. For example, AlexNet CNN has over 60 million model parameters that need 250MB of memory for storing the weights based on 32-bit floating-point representation and this large amount of storage is not supported by FPGAs. Also, various CNN layers have different implementations, and hence it becomes important to carefully design the architecture of CNN to maximize the performance of CNN layers. It is not always feasible to design such architectures and hence, the AutoML/NAS concepts that can help design architectures for FPGAs can be used.

3.3 AutoML/Neural Architecture Search

As mentioned in the previous section, memory constraints on FPGAs play a big factor in its configuration. Fedorov et al. discuss about sparse architecture search for CNNs on the micro-controller unit (MCU) in their paper [Fed+19]. Deployment of CNNs on MCUs is challenging as MCUs are highly resource-constrained. The authors use Sparse Architecture search (SpArSe) combining neural architecture search (NAS) and net-

work pruning in conjunction with hyperparameters around morphology and training to discover models that are highly performant with memory-constrained architectures (i.e. models with fewer parameters).

In the paper [Sun+20], the authors talk about the framework for a genetic algorithm which is easy to understand and easily extendable for images. The authors also talk about using various selection strategies for mutation along with the creation of initial networks for evaluation while performing NAS. Liu et al. propose a method for efficient architecture search in their paper, called DARTS (Differentiable ARchiTecture Search) [LSY18]. The authors propose to relax the search space of NAS to be continuous so that the architecture can be optimized for its validation set performance by gradient descent.

In another research, [Lu+18], the authors talk about a multi-objective evolutionary approach for neural architecture search. The NSGA-Net, as the authors explain, can effectively optimize and trade-off multiple objective functions which is also the possibility while designing NAS for FPGAs. The paper also mentions the importance of an efficient exploration and exploitation strategy of search space for the optimization of networks while using NAS.

There has been research about using NAS for medical data. Rapaport et al. in [RSP19] talk about using neural architecture search for Electroencephalography (EEG) Data Analysis and Decoding. EEG is the acquisition and decoding of electric brain signals which is similar to ECG, except for the electrical signals are from the brain. The authors employ genetic algorithms (GAs) as the means for NAS and analyze the evolution of NN architectures over time, which helps better understand the NAS process. The authors concluded that the ability to generalize in EEG classification is key to good CNN architectures.

4 Data and Model Exploration

With the increasing adoption of machine learning across all industries, the data used for training a DNN model has become a key component for its success. It is essential that for supervised machine learning, the training data is classified into various categories. For example, if there are 2 classes of images for the machine learning model to recognise, say a cat and dog, these images must be labeled containing these two classes of animals.

In this chapter, the dataset used for the thesis is discussed using various visualization techniques. Seed models developed for classifying the dataset are also discussed in the later sections.

4.1 Dataset

The data used for the experiments is a two-channel anonymous ECG dataset. The ECG data is in a mixed text/binary format with the data stored as a 16-bit little-endian integer. The data is already collected and it is stored in ".ecg" files. Since it is the ECG data, we have voltage values stored in the file.

4.2 Data description

The voltage value of the ECG signal is the amplitude with respect to a value over a period of 120-180 seconds. To retrieve the voltage value from the 16-bit values from the stream of data in the ".ecg" file, we subtract value of the "offset" from the actual values and then multiply the result by "factor". The "offset" and "factor" are header values which are obtained from each ".ecg" file and are constant values for that particular ECG signal. This is used to normalize the data in our case. The formula, as such, to calculate the value (or the amplitude of the signal) is:

$$(value\ -\ offset) * factor = amplitude\ in\ millivolts\ (\mathbf{mV})$$

For an example, if the 16-bit binary value after decoding (with little endian format) is 2287, if "offset" from the header is 2048 and "factor" is 0.002930, then the amplitude is :

$$(2287 - 2048) * 0.002930 = 0.70027 \ mV$$

An example for the header in the ".ecg" files is shown below in table 4.1:

Example for the text header	
Type:	CL1000_ECG
Sample rate:	1024.000000
Offset:	0.000000
Unit:	mV
Factor:	- 0.002441
Compression:	0
Device:	CL3000
Device SN:	041 04 0072
DateOfLastCalibration:	06/17/2003 12: 46: 11,000
PortName:	USB
Driver name:	d_comm.dll
Driver version:	1.1.0.9
Time of first sample:	0
Channels:	8th
Channel Names:	I II V1 V2 V3 V4 V5 V6

Table 4.1: *Example for ECG header in the "*.ecg" files*

The sampling frequency (or sample rate) is also part of the header of the ".ecg" file. This is used to calculate the duration of the signal. The data stream consists of data from **2 channels** for this duration. If the USB device used to collect the data was temporarily disconnected during the recording or if the transmission was disturbed, then this period in the signal will contain zero values. If there are gaps in the ECG, then these gaps are also filled with zero values. These zero values must also be accounted for when decoding the data. It is also noted that binary data is aligned on 8-byte boundaries and sometimes, the data must be moved to the beginning of this 8-byte aligned data manually when reading the ".ecg" file.

4.3 Data Visualization

ECG signals can have different waveforms and morphologies. Similar to many other time-series data, the main problem with the manual analysis of ECG signals lies in the difficulty of detecting and categorizing these different waveforms and morphologies into the various classes. The current dataset of the ECG consists of 2 classes of data. The two classes are sinus rhythm and atrial fibrillation (AF) rhythm. AF is the irregular heartbeat, or arrhythmia, whereas the sinus rhythm is the regular heartbeat.

The dataset consists of 8,000 files for sinus rhythm and 8,000 files for AF. The dataset, as per the usual conventions, is divided into 70% for training, 15% for validation and 15% for testing. This results in:

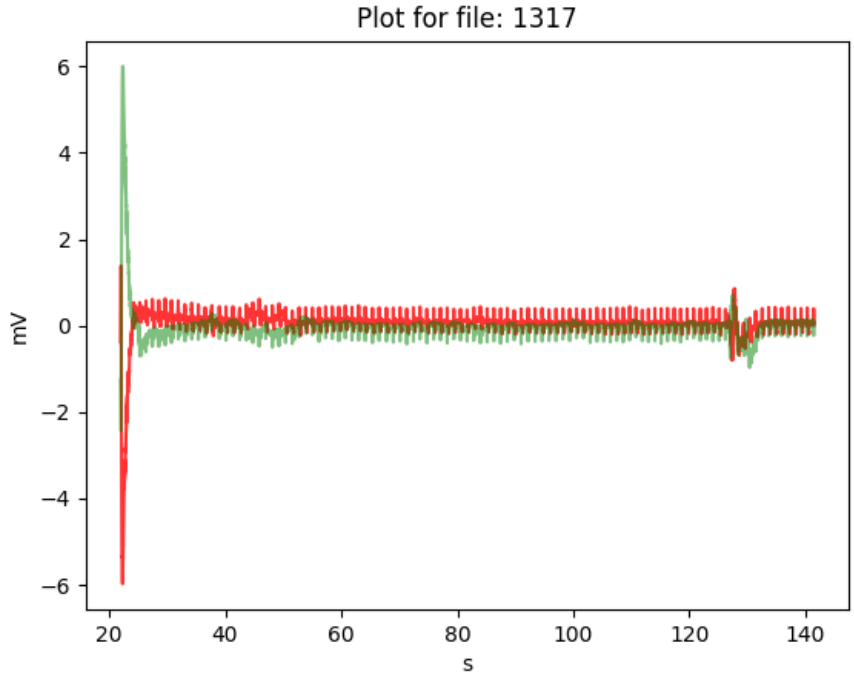
- 5,600 files for training - for sinus
- 5,600 files for training - for AF
- 1200 files for validation - for sinus
- 1200 files for validation - for AF
- 1200 files for testing - for sinus
- 1200 files for testing - for AF

Visualization for two ".ecg " files are provided in figure 4.1, figure 4.2 and figure 4.3.

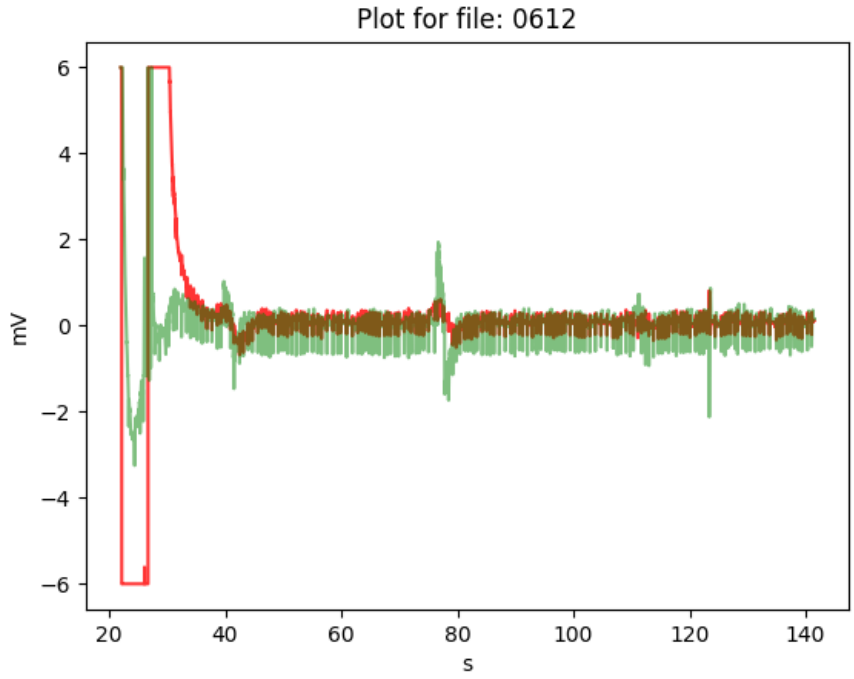
4.4 Data analysis

Figure 4.1(a) shows the ECG signal for the sinus rhythm and Figure 4.1(b) for the AF rhythm. These files were chosen randomly from the available data. For both sinus and AF, the voltage values obtained for two channels are plotted. The $y - axis$ represent the voltage values (in mV) and $x - axis$ represents time (in seconds).

The data is unfiltered and not prepossessed. With this, the voltage range for both sinus and AF is between -6mV and 6mV for both the channels. Majority of the voltage values are concentrated around zero mV (between -1mV and 1mV) with occasional values exceeding -1mV or 1mV. This is also evident from the frequency distribution of the data (Figure 4.2). This was also the observation for most of the histogram plots that were generated and studied for different files. Although most

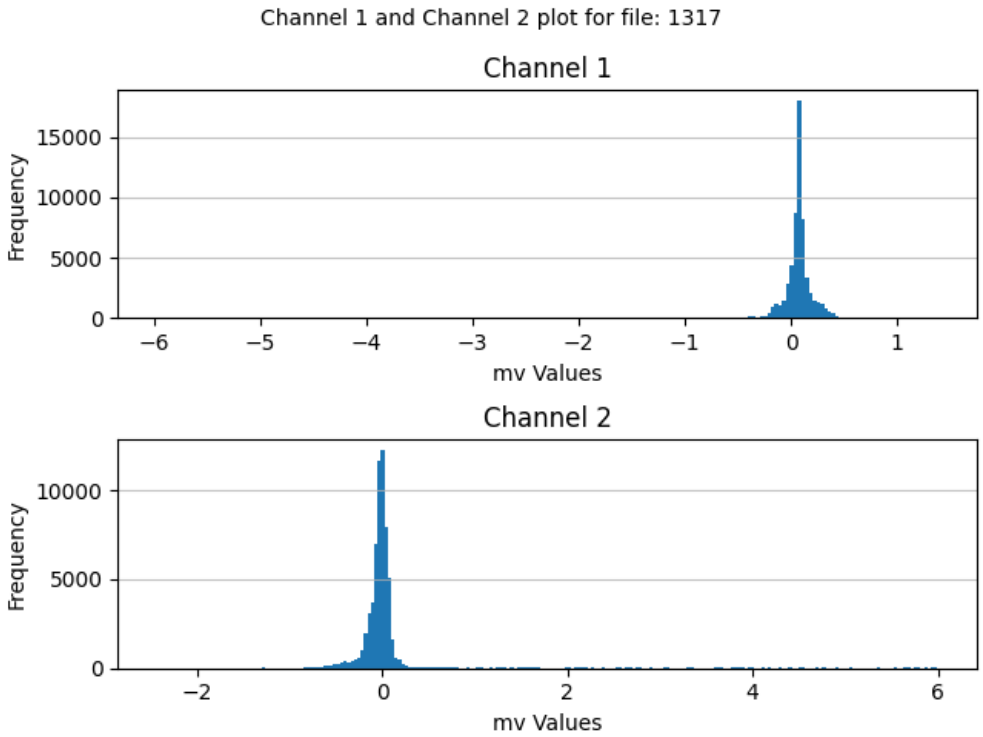


(a) ECG signal of Sinus

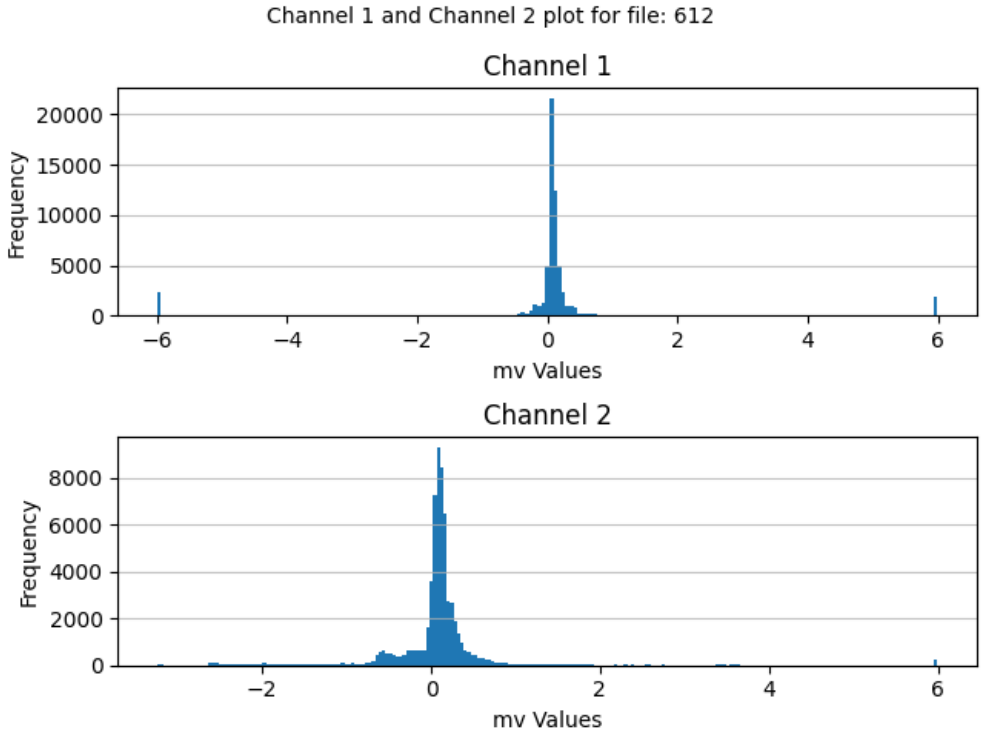


(b) ECG signal of Atrial fibrillation

Figure 4.1: *ECG signal of Sinus vs Atrial fibrillation*

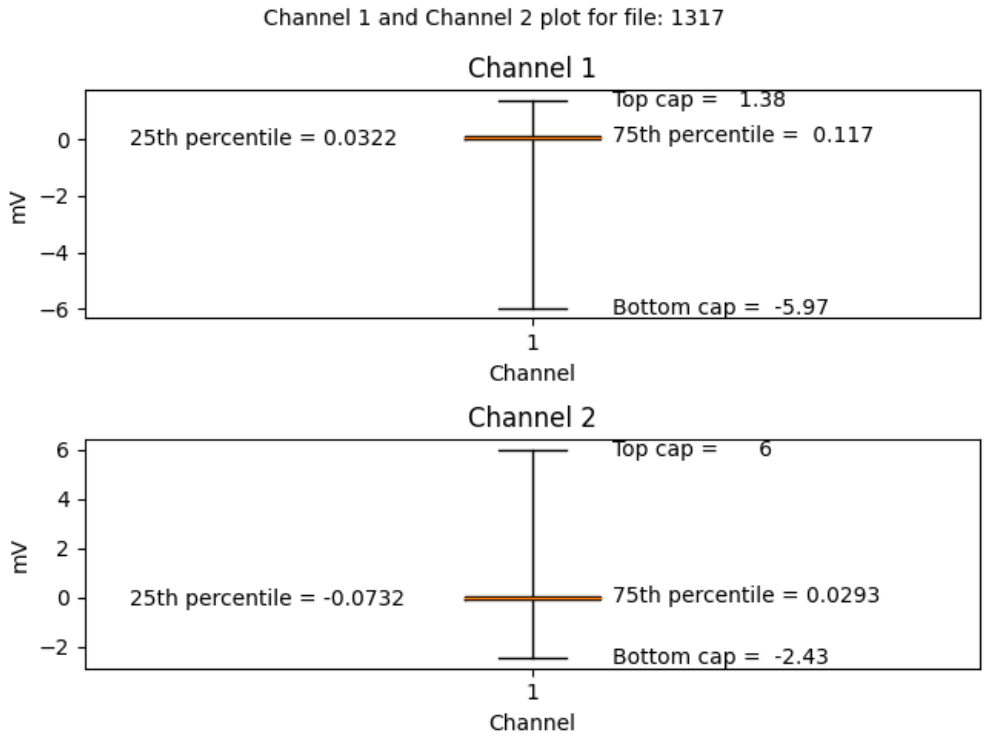


(a) ECG histogram signal distribution of Sinus

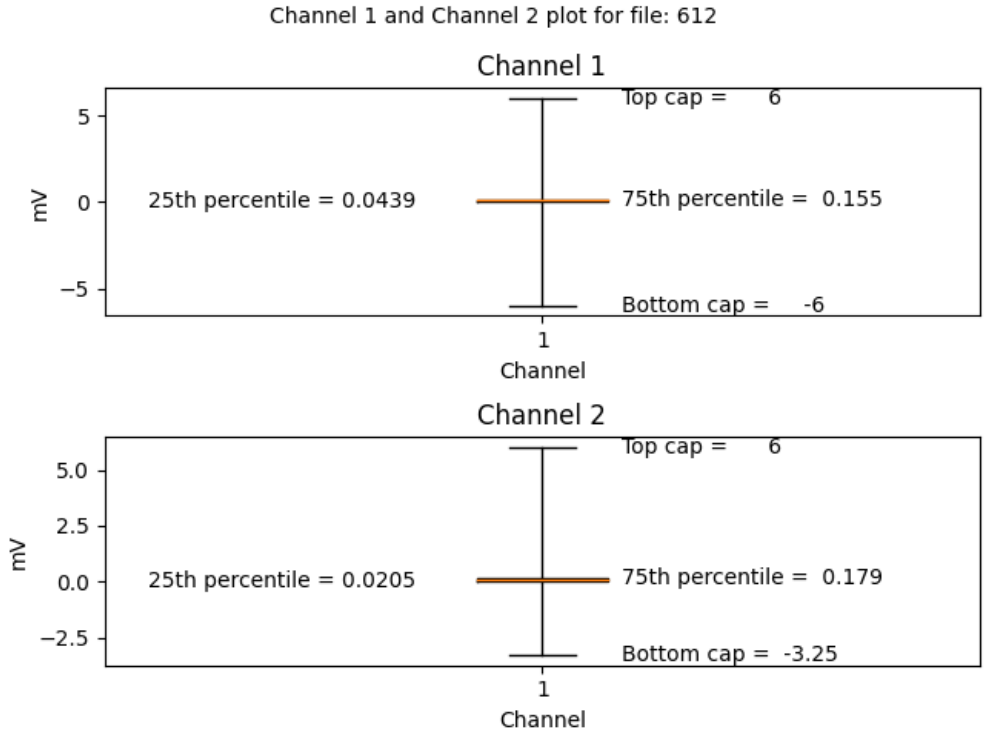


(b) ECG histogram signal distribution of Atrial fibrillation

Figure 4.2: *ECG signal distribution - Histogram of Sinus vs Atrial fibrillation*



(a) ECG box plot signal distribution of Sinus



(b) ECG box plot signal distribution of Atrial fibrillation

Figure 4.3: *ECG signal distribution - box plot of Sinus vs Atrial fibrillation*

of the values are concentrated around zero, those zero values are because of the cardiac cycle and not because of the disconnection or disturbance in transmission. The occasional values exceeding -1mV or 1mV are the spikes seen in the ECG graph (Figure 4.1).

Further, to understand and compare the distribution of the data points, we create box and whisker plots for the same ECG files of sinus and AF (Figure 4.3). A box and whisker plot is a visual representation that provides information about the minimum value, first quartile, median (second quartile), third quartile, and maximum of a set of values. The box and whisker plots can provide information about outliers, the data symmetry and how tightly the data is grouped. It also provides information on the variability or dispersion of the data [MTL78] & [KA14].

As seen from Figure 4.3, again, the majority of the voltage values are in between -1mV and 1mV. We also note that there are not many outliers in this distribution and this was the case for other files as well. This signifies that the ECG data in the dataset is clean enough to try to proceed in building the machine learning model for this dataset without significant prepossessing.

When a normal ECG signal is provided, the detection of AF within the ECG signal is problematic as it may be episodic. Although these two plots in figure 4.1 may appear visually different, it is not very easy to distinguish between an AF signal and a sinus signal. There were many cases for sinus and AF which had similar distribution of signal. This was the general observation throughout the dataset. Hence, it becomes essential to extract minute features from the signal to classify it. As convolutional neural networks are very good feature extractors, we try to use them to classify these signals. We speak about the CNN architecture and experiments in the next section.

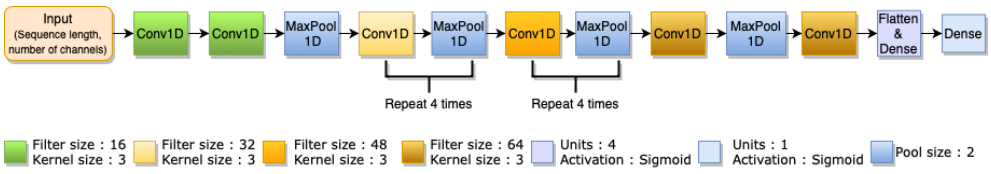
4.5 Seed Model

With this analysis of data, we wanted to proceed and check if CNN models are sufficient to classify the dataset. The goal is to design one or more CNN models which can be used as seed models to achieve the training and validation accuracy of greater than 90% for the training and validation dataset. Also, the authors from the paper [Rea+19] mention that starting with a high-quality model in the initial stages while using neural architecture search increases the probability to get a higher quality model in the end.

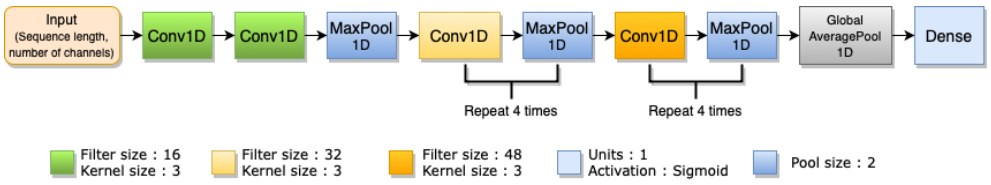
For the training process, the standard TensorFlow input pipeline is

used. From the 11,200 files (5,600 for sinus rhythm and 5,600 for AF), we create a TFRecord file for training. A TFRecord is created to serialize the data and to read it back efficiently and linearly. With 2 channel data for a single ".ecg" file, in this work, we limit the number of samples per channel to be 60,000 (which is precisely the data for 2 minutes). We prevent the rest of the samples from being written to the TFRecord. With this, we essentially create a binary record of the 11,200 files. Similarly, TFRecords for testing and validation each with 2,400 files (1,200 for sinus rhythm and 1,200 for AF) are created.

TFRecord dataset APIs of TensorFlow are used to read the data from the TFRecords. The dataset API also helps to shuffle the TFRecord data as this also becomes critical to randomly distribute the data. After developing this pipeline, the architectures in Figure 4.4 are used to train the two seed models.



(a) Seed model with additional convolutional and dense layers



(b) Seed model with Global average pooling

Figure 4.4: *Seed Models*

The first seed model in Figure 4.4 (a) is developed to use CNN layers to classify the dataset with the minimum number of parameters as possible. After developing this model, we tried to explore the possibility of further reducing the complexity of the model. The thought process was to avoid the use of fully connected layers (dense layers) for the model. This was accomplished by using a 1-dimensional global average pool layer which not only flattens the input from the previous layer, but also reduces the number parameters of the entire model (Figure 4.4 (b)). This results in

reduced complexity with the same hyperparameter settings.

The architectures in Figure 4.4 (a) and (b) consists of a 1-dimensional TensorFlow input layer. This layer takes the sequence length (fixed at 60000) and the number of channels (2 channels provided in the dataset) as the input. This is followed by two 1-dimensional convolutional layers, each having a filter size and kernel size of 16 and 3 respectively. These convolutional layers are followed by a 1-dimensional max pool layer (with pool size 2 to half the first dimension of the data). The max pool layer is followed by a set of convolutional layer (filter size of 32 and kernel size of 3) and a max pool layer (with pool size 2), repeated 4 times in that order. This is followed by another set of convolutional layer (filter size of 48 and kernel size of 3) and a max pool layer (with pool size 2), repeated 4 times in that order. The architecture structure is similar for both seed models up to this point.

For the architecture in Figure 4.4 (a), the max pool layer is followed by a 1-dimensional convolutional layer (filter size of 64 and kernel size 3), a max pool layer (with pool size 2) and another convolutional layer (filter size of 64 and kernel size 3). These layers are followed by a flattening layer which flattens the data for the dense layer (fully connected layer), which uses 4 neurons. In the end, a dense layer with a single neuron for classification is used.

For the architecture in Figure 4.4 (b), the max pool layer is followed by a 1-dimensional global average pool layer which flattens the data for the dense layer which is used for classification.

All the convolutional layers use "**ReLU**" activation, whereas the dense layer uses a sigmoid activation. The sigmoid activation is used to classify the result either to '0' (sinus rhythm) or '1' (AF). For both models the "**Adam**" optimizer is used with a learning rate of $4e^{-4}$ and decay rate of $1e^{-3}$. As we are performing binary classification, binary cross-entropy is used as a loss function. The TFRecord for validation is passed for performing validation for the training process. The accuracy metric along a batch size of 32 is used for both training and validation. With these conditions, we train the models for 5 epochs. The models are then tested for the test data (TFRecord for test data). The models result in very good testing accuracy, thus confirming our intuition that CNN models are sufficient to classify this dataset. The results for the seed models are mentioned in section 6.1.

Hyperparameters

The hyperparameters used for the seed models are described in Table 4.2. These hyperparameters are derived by experimenting with the seed models. They set the foundation for the NAS, as the same set of hyperparameters are also used for the NAS process.

Hyperparameter	**Value**
Optimizer	Adam
Learning rate	$4e^{-4}$
Decay	$1e^{-3}$
Number of epochs	5
Batch size	32
Max runtime seconds	300 s
Training Samples	11200
Validation Samples	2400
Testing Samples	2400
Steps per epoch (training samples / batchsize)	350
Validation steps (validation steps / batchsize)	75

Table 4.2: *Hyperparameters used for the seed models*

5 Approach and Implementation

This chapter covers the various design approaches, flowchart of the approaches and methodology used in this work.

5.1 Proposed Approach

For Neural Architecture Search (NAS), various architectures needs to be explored and a method to encode this architecture is introduced in subsection 5.1.1. The concept of gene programming is used. While using NAS, the framework for the selection of the parents for mutation becomes essential. Along with the selection strategy, the method by which the mutation takes place also plays a vital role in generating new topologies that are efficient. In the following sections, we discuss the framework for the selection policies and the mutation strategy of generating new topologies. The fitness evaluation process is also discussed in the following sections.

5.1.1 Encoding Layers

Evolution is essential for using the NAS. For implementing the concept of evolution, this work uses genetic programming (gene expression programming). The architectures are represented by chromosomes that work as genotype and the set of architectures is referenced as the population. Parsing these chromosomes creates parse trees (or architecture tree as shown in Figure 5.2), which are called phenotype. A chromosome consists of a linear symbolic string of fixed length composed of genes of equal size.

Figure 5.1 shows a chromosome as a sequence of genes. Each gene is represented by an array of 3 numbers. The first number stores the information about the layer this gene is going to represent. The second and the third number stores the information about the layers from which this gene can receive the input from. If this layer can have multiple

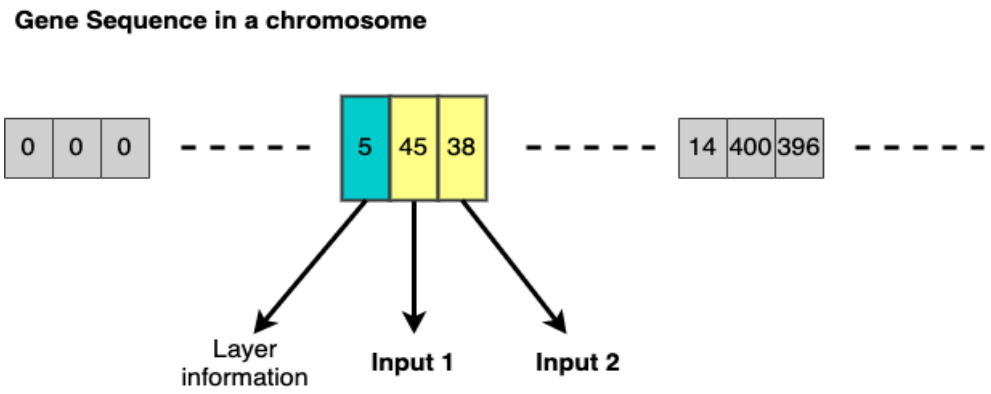


Figure 5.1: *Example of a gene sequence*

inputs, then both the numbers are used, else only the first number is used for the input.

The number of genes in a chromosome can also be a variable. Also, all the genes are not active meaning that we don't use the entire gene sequence of the chromosome to form the network. Only the layers of the active genes are used to construct the network. We make use of two variables - one for the minimum number of active genes and one for the maximum number of active genes, to control the minimum and maximum number of layers in the network.

5.1.2 Search space

The approach to create and explore the search space plays a vital role to find an optimal solution in NAS. The search space contains all feasible solutions which can lead to complications such as where to look for solutions to the problem, where to begin, or how to determine if a solution is the best solution.

Figure 5.2 shows relatively simple architectures in which the layers of the neural networks are chained, in the sense that the architecture can be written as a sequence of n layers. Each node in the architecture corresponds to a layer of the neural network. Each edge represents the connection between two layers with the arrow providing the information about the input and output.

In the architecture on the left, the $i'th$ layer receives input from the $i-1'th$ layer, resulting in a sequential model. The $i'th$ layer produces an output which serves as an input to the $i+1'th$ layer. The architecture

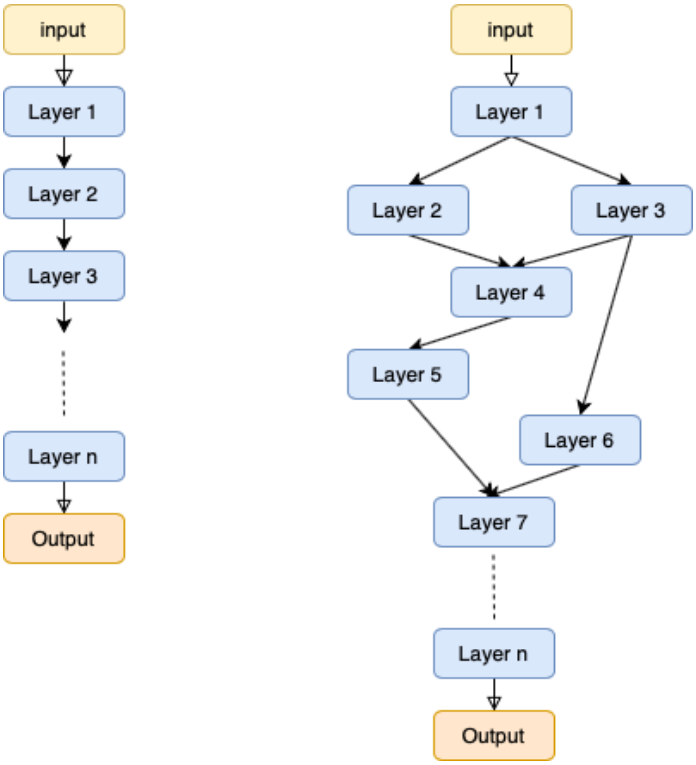


Figure 5.2: *Illustration of various architecture*

on the right is more complex as the $i'th$ layer can receive input from multiple previous layers. Also, the output from this layer can serve as an input to multiple layers, resulting in a non-sequential model.

In this work, convolutional layers, max-pooling, average pooling layers, sum, concatenation, and fully connected layers are used. The parameters to the convolutional layer are the filter size and kernel size with a stride size and padding size of '1' each. Particularly, this work uses filter size of 16, 32, 48 and kernel size of 3 *and* 5. The parameters to the pooling layers are the kernel of size 2 *and* 3 and the stride of size 1 *and* 2. To introduce a non-sequential element to the architectures, "sum" and "concatenation" layers with two inputs are used. Fully connected and average pooling layers are used as output layers.

The restrictions to the search space are added in the following way:

- "n" number of layers : The number of layers is presently unbounded.
- The maximum and minimum number of parameters of an architecture are restricted to 100,000 and 4,000 respectively. This means that the number of parameters of a network cannot exceed 100,000

and cannot be less than 4,000.

- The architecture is evaluated for fitness before training. Only valid networks are trained after determining if each of the architecture path is valid. This also includes special checks for the concatenation and sum layer.

After introducing restrictions to search space, we use the principle of mutation to explore it. Mutation is part of the neuro-evolutionary method to explore search space. Evolutionary algorithms evolve a population of models. In every evolution step, at least one network (possibly trained) serves as a parent to generate offspring networks by applying mutation to the parent. Mutation, as mentioned in subsection 2.4.2 involves altering a layer, either by making one of the gene inactive and making another gene active in the parent chromosome or modifying the layer information of one of the active gene.

The mutation rate, a crucial hyperparameter, is used in the process of mutation of the parent. This parameter determines the variance of the configuration of the child from the parent. This essentially means that if a parent has a bad configuration, which is determined by the accuracy metric or a combination of accuracy, detection rate, false alarm rate and number of parameters, the child should be drastically different from the parent. This results in aggressive mutation for the child with a high mutation rate. But on the other hand, if the parent is a good network, then the child should only have small variations in the configuration with a low mutation rate. The mutation rate has a lower bound and an upper bound which adds variance in the process of mutation.

We use 3 kind of mutation in the work :

1. **Random Mutation:** This kind of mutation is used to randomly mutate the genes of a network. Random, in this context, means changing the gene information by either activating/deactivating the gene or by modifying the layer information or by modifying the input layers to the gene. This mutation is only used when no previous network exists as this kind of mutation helps to generate new networks randomly. In this work, this mutation is only used to generate the initial population.

2. **Genetic Mutation:** This kind of mutation is used to generate a child network from a parent network. This mutation is used when at least one previous network exists, that can be used as a parent. This kind of mutation helps to generate new networks by altering the

genes of the parent network. If there are **i** children to be generated in one generation, then this mutation creates a child - child[i] from a parent[i] i.e. the genes are mutated selectively from the parent network. In this work, except for the initial set of networks, all the other networks are generated using genetic mutation.

3. **Neutral Mutation:** This kind of mutation is used to mutate the inactive genes of all parents for a generation. This mutation is used to enhance the gene diversity. This mutation is used to create a child network by only altering the inactive genes of a parent network.

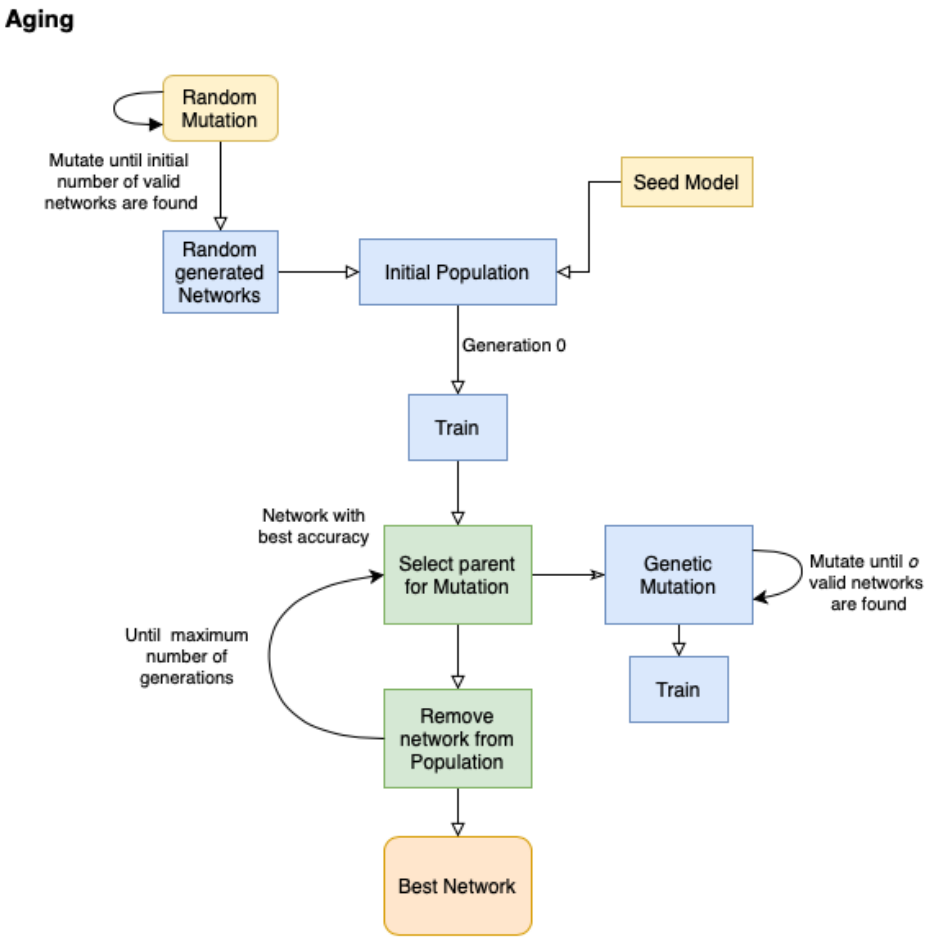


Figure 5.3: *Workflow of implementation of NAS*

An overall picture of the process of NAS implementation is explained in Figure 5.3. This implementation is inspired by the work done by

[Sun+20]. As shown in the figure, initially, a certain number of random networks are generated with the restrictions to the search space mentioned above. This initial number is passed as a parameter from the command line. These networks belong to the set of initial population which are generated through random mutation. Also, the seed model (explained in section 4.5) can be added to the initial number of generated networks to improve the results. This first generation of the NAS process is then trained.

5.1.3 Fitness Evaluation

All of the networks generated during the NAS process including the initial set of networks use the same input pipeline defined in section 4.5 for the training process. In figure 5.3, this process is represented by the block "Train". Every network is associated with a network identifier and this is a counter that is incremented every time a new network is subjected to training. For the validation process, a custom validation function is added which uses the TFRecord for the validation dataset. Additionally custom values - detection rate and false alarm rate are added along with validation accuracy and validation loss. The values detection rate and false alarm rate are calculated based on true positive, true negative, false positive and false negative values.

- True positives - This is the count of the correctly predicted 'atrial fibrillation' labels by the model for the validation dataset. If the predicted label and the actual label is atrial fibrillation (which is value 1 in our work), then this counter is incremented.
- True negatives - This is the count of the rightly predicted 'sinus rhythm' labels by the model for the validation dataset. If the predicted label and the actual label is sinus rhythm (which is value 0 in our work), then this counter is incremented.
- False positives - This is the count of the falsely predicted 'atrial fibrillation' labels by the model for the validation dataset. If the predicted label is atrial fibrillation but the actual label is sinus rhythm (which is value 0 in our work), then this counter is incremented.
- False negatives - This is the count of the falsely predicted 'sinus rhythm' labels by the model for the validation dataset. If the predicted label is sinus rhythm but the actual label is atrial fibrillation (which is value 1 in our work), then this counter is incremented.

Based on the above four counters, detection rate and false alarm rate are calculated as follows:

$$detection\ rate\ =\ true\ positives\ /\ (true\ positives\ +\ false\ negatives)$$

$$false\ alarm\ rate\ =\ false\ positives\ /\ (false\ positives + true\ negatives)$$

In every generation, the first step is to select the fittest parents for mutation. This selection of parents is based on the selection strategy (explained in subsection 5.1.4) and is represented by the block "Select parents for Mutation" in the figure 5.3. After selecting the parents, the method of genetic mutation is used for generating children (or individuals). Again, depending on the selection strategy, either certain children or all of them are selected for training and trained in the manner described above.

The next step is to select, among the newly trained children and the parents of this generation, those networks that are going to survive. These are the set of networks that are going to be used in the next generation for selecting a new set of parents. Along with this, the best network in this generation is also selected and stored in a variable. This network can be one of the newly trained children or one of the parents. We next look at the selection of parents for NAS.

5.1.4 Selection strategy

Evolutionary algorithms have a variety of methods to sample and select the parents for mutation and to generate offspring networks. This selection plays a significant role in exploring search space and producing good networks. In this project, there were four methods used to select the parent network.

The selection policies are also used to train the offspring and select the networks to survive for the next generation among the entire population. The number of parents to be selected per generation is passed through the command line (we will use p to represent this number). The other parameter used is the number of offspring to be generated at the end of each generation (we will use o to represent this number). This parameter is also passed through the command line. For the selection strategy aging (described in the next section), this adds a condition that the initial population should be greater than the number of parents.

Also, as one of the goal is to produce the smallest possible network with detection rate greater than 90% and false alarm rate less than 20%, we introduce a term called **weighted objectives**. This, along with accuracy

are the measuring terms used in this thesis. Weighted objectives takes into account the number of parameters, the detection rate and the false alarm rate to calculate a weighted term which is then used for selecting the parent and the best network for the generation. The advantage of using this is that our definition of a good network is not based only on the accuracy, but also on the detection rate, false alarm rate and number of parameters. Essentially, if a network with less number of parameters produces good results, then such a network is more considered better for our scenario than a network with very high accuracy, but also with high number of parameters.

The selection policies are explained in the following sections and are listed below:

1. aging - This method trains all the children but the selection of a parent is random.

2. all_children - All offspring created from the parents are trained in every generation.

3. selected_children - Only selected offspring created from the parents are trained in every generation.

4. lemonade - This method uses the concept of Pareto front. Pareto front optimizes multiple objectives such as accuracy and number of parameters.

Aging

For the selection strategy of *aging*, the logic for selection process is inspired from [Rea+19]. One of the goals of the selection process is to select a parent or parents for mutation. For *aging*, at each cycle, from the set of available parents for mutation, p networks are chosen at random, each drawn uniformly without replacement. As these networks are already trained, all the essential metrics such as accuracy, detection rate, etc. are available. From these networks, the network with either the best accuracy or with best weighted objectives is chosen as a parent for the mutation process.

The selection of the parent is followed by genetic mutation to mutate the parent to generate o new networks. These are the offspring networks of this generation and are added to the current population. All these offspring are trained using the same pipeline mentioned above. The next phase of the NAS process is to eliminate a network or a set of networks

from the current population. By this elimination process, we reduce the number of networks available for parent selection in the next generation.

For this strategy, the oldest network from the current population is eliminated (hence, the name aging). The remaining networks are available for parent selection in the next generation. This entire process is repeated for all the generations.

The mutation of the parent provides exploration whereas parent selection provides exploitation. The parameter p is used to control the aggressiveness of the exploitation process. Setting $p = 1$ reduces to random selection of parent and leading to random search.

This selection strategy is used in evolutionary algorithms because of its simplicity. Although mutation causes a random construction of architecture thus rendering the entire process random, this strategy provides a distinctive advantage of mutating only the good models. Also, the NAS process is not affected significantly even if the best network is eliminated as the best available network is already mutated. Furthermore, by eliminating the oldest model, this strategy essentially favors the newer models in the population allowing us to explore the search space significantly rather than zeroing on the good models too early. Thus, this selection tends to improve the population over time.

All_children and selected_children

These selection policies are inspired by [Sun+20]. For these strategies also, the first step is to select parent networks for mutation. The selection process for the first generation begins by selecting network identifiers as parents for mutation from the initial population. It is achieved by randomly creating o number of pairs from p available parents. Further, from all these pairs, the network with, either the better accuracy or with better weighted objectives is selected resulting in o parents which can be used for mutation. An important note is that, because of the randomness of the creation of the pairs, there is the possibility that the same parent can be used for mutation multiple times.

This is followed by genetic mutation to mutate o available parents to create offspring for this generation. For the selection strategy *all_children*, every offspring created from mutation is trained resulting in o new networks every generation. For *selected_children*, a set of networks is selected randomly. This set can include parents from the previous generation and a subset of the offspring of this generation. Only the subset of the offspring are trained.

For both the selection policies, the next phase of the NAS process

is to eliminate networks from the current population. This is achieved by pairwise comparison of networks according to the objectives. The pairs are created just before the training of the networks. Thus, after training, the same pairs are available with their essential metrics. The network with higher accuracy or better weighted objectives is chosen from the pair and is available for selection for the next generation. The remaining networks from the pairs are eliminated. This entire process is then repeated for all the generations.

Lemonade (Pareto-front)

This selection strategy is inspired by [EMH18a]. Lemonade, as described by authors, is an evolutionary algorithm for multi-objective search with the search objectives in our case being the performance and the number of parameters of the model. This algorithm handles objectives separately categorizing them into cheap objectives (such as an architecture's number of parameters) and expensive objectives (since it requires training the model first). The expensive objectives are the error rate, false alarm rate, and non-detection rate (1 - detection rate).

The first step of the process is to determine candidates for mutation. The sampling of parents is done through a density function for the number of parameters (cheap objective). The density function searches for networks on the Pareto front where the network density is high. The next step is to generate o new networks from the selected parents using genetic mutation. In the next stage, p networks are chosen from o networks based on the number of parameters, and these are trained and evaluated to obtain the results.

The final stage is to select networks which would survive for the next generation. A Pareto front is generated for the cheap objectives and expensive objectives. In every generation, the old Pareto front is combined with the children to generate a new Pareto front, and only those networks which lie on the Pareto front survive.

Comparison of selection policies

Every selection strategy has its advantages as well as disadvantages. The aging selection strategy is very easy to implement and has the advantage of generally mutating only good models. The population improves over time due to the removal of the oldest network but that might act as a disadvantage if run for a long period.

The selection policies - all_children and selected_children try to reduce

the randomness by creating pairs and choosing the better network in the pair as parents. This improves the method used for the selection of parents in aging, which selects parents only randomly. Again, for the process of elimination, pairs are created for comparison, and a network with better accuracy or weighted objective is selected instead of removing the oldest network, which was the case in aging. This again improves the networks available for selection for the next generation.

As selected_children does not train all the children, there could be the case that a child that is created but not evaluated has a better accuracy than those trained. The strategy all_children tries to explore more but requires a lot of training time as it tries to train all the created children.

In contrast, lemonade uses a Pareto-front strategy. Multiple objectives like error and number of parameters are optimized simultaneously. With this strategy, a set of networks is returned and it is up to the user to select a suitable network. The technique to optimize multiple objectives is an advantage for our use case. The final decision to select the network is taken by the user, for which the user needs to have a basic understanding of the use case.

The results of all the above-mentioned selection policies are discussed in the next chapter.

6 Experiments and Results

In this chapter, all the experiments performed using the seed model architecture (section 4.5) and NAS with different parameters are described along with their empirical evaluations. The experiments are supported by their qualitative and quantitative results. The validation is run on the validation dataset (TFRecord for validation data) which contains 1,200 AF and 1,200 sinus rhythm files. The results of the selection strategies are discussed and compared.

Hyperparameters

Hyperparameters are kept the same throughout the experiments for NAS. For training, Adam optimizer is used with an initial learning rate $4e^{-4}$. A batch size of 32 is used for training and validation for the seed model as well as for all the models trained during the NAS. The table 4.2 shows the hyperparameters used in the experiments with few changes. The hyperparameter - decay rate is not used for NAS, while all the models for NAS are trained only for 5 epochs.

6.1 Evaluation of Seed models

The seed models with the architecture described in Figure 4.4 are trained with the hyperparameters defined in table 4.2. The models are trained for five epochs. The table 6.1 provides the comparison of the results for the two seed models. The results are the average of three runs for both the models.

As seen from the table, Seed Model 1 produces an average training accuracy of greater than 97% where as Seed Model 2 produces an average training accuracy of nearly 94.5%. This result is as per expectations as the decrease in number of parameters would, in general, reduce the accuracy. But in this scenario, the number of parameters is nearly halved between the two models. The validation and the testing accuracy also confirm the training accuracy. This in turn implies that the model is not over-fitted.

Results	Seed Model 1	Seed Model 2
#epochs	5	5
#Parameters	72,777	37,361
Training Accuracy	0.972	0.943
Validation Accuracy	0.962	0.946
Testing Accuracy	0.965	0.949
Detection Rate	0.990	0.970
False Alarm Rate	0.060	0.072

Table 6.1: *Comparison of seed model results*

High detection rate and low false alarm rate are also important measures of a good model as they provide us information about the ability of the model to classify new samples. Both the models have resulted in high detection rate (nearly 99 % and 97 % respectively) and low false alarm rate (6 % and 7 % respectively). The seed models, thus produce very good results.

6.2 Experiments and results with NAS

Experiments for NAS are run with various configurations in order to evaluate the effect on the results. This includes various selection policies, different number of offspring networks generated per generation, different number of the initial networks, different number of parents for generation and for how many generations should we run the experiment for.

All the experiments were run using the TITAN GPUs provided by the Fraunhofer cluster. Most of the experiments were run on more than one node with multiple GPUs in every node to enable parallel training of the models. Mutation was performed parallelly on several CPUs nodes.

The mutation parameters for the experiments are set as follows:

- Threshold accuracy of a parent is set to 80%. If the accuracy of the parent if less than 80%, then the mutation rate is high, as the parent is considered to be a bad parent. The child should greatly vary from the parent.

- If the accuracy of the parent if greater than 80% and number of parameters is greater than the average of maximum and minimum parameters, then the mutation rate is moderate as the parent is considered to be a decent parent. This essentially signifies that the offspring architecture should not differ very much from the parent.

- If the accuracy of the parent if greater than 80% and number of parameters is less than the average of maximum and minimum parameters, then the mutation rate is very low as this is considered to be a good parent. This also signifies that the offspring architecture should vary very little from the parent.

6.2.1 Experiments with Selection strategy - Aging

For aging, two experiments are run - one with the inclusion of the seed models and another without them. The same setting which is used in Amoebanet, producing one offspring per generation and keeping the population constant is used for these experiments (Table 6.2). We ran few experiments with less than 10 initial networks, but this resulted in reducing the number of parents for selection process. Although, this increased the probability of choosing a good network, it reduced the number of networks to choose from. Hence, we choose the population to be constant at 10 networks. Also, Mutation rate of 70% - 90% for a bad parent, 10% - 30% for a decent parent and 5% - 10% for a good parent is used for these experiments. Accuracy is used as a measure for these experiments.

#epochs	#Initial population	#parents	#offspring	#generations
5	10	5	1	200

Table 6.2: *Parameters for selection strategy - aging*

Analysis

For this selection strategy, for the initial population (generation zero), the network with the best accuracy and for the subsequent generations, the accuracy of every offspring generated is used for analysis. Figure 6.1 and Figure 6.2 show these results. These graphs also show the number

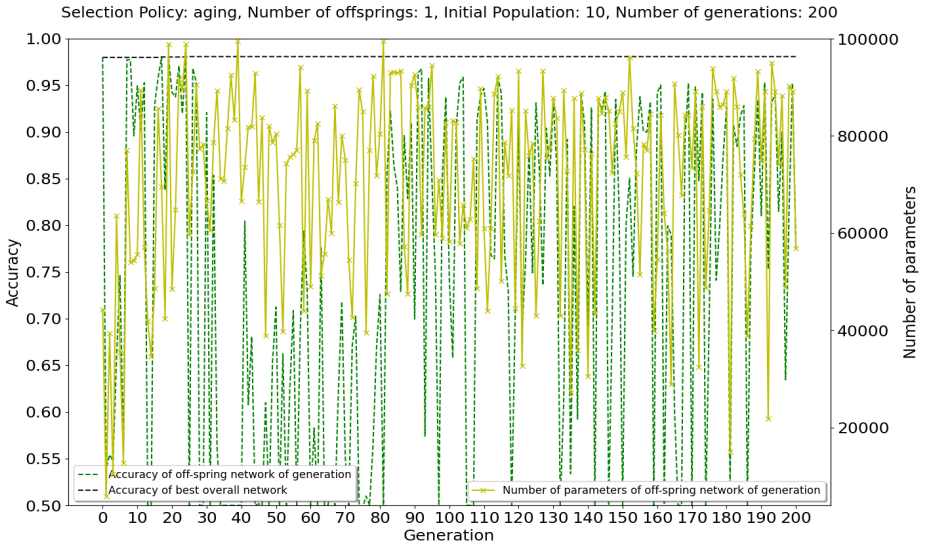


Figure 6.1: *Aging, with seed models*

Figure 6.2: *Aging, without seed models*

of parameters of the networks and the best overall accuracy for these experiments. Figure 6.1 clearly explains that our seed model produces very good results and it is very difficult to improve these results. There is a slight increase in accuracy in the 15th generation, but this is at the cost of increase in number of parameters as well. This, in comparison with

Figure 6.2 where no seed model is used, shows the actual evolving nature of the entire NAS process. This figure explains that the NAS process is always trying to improve the accuracy of the networks, but at the cost of increasing number of parameters. We find a network with very high accuracy between the 130th and 140th generation, but with high number of parameters. This result is expected, as well as other results of these experiments of aging for NAS process, as there is random selection of parents during the selection phase. The results are very noisy with many networks producing 50% accuracy (networks which don't learn). This is also to be expected from aging during the NAS process, as in these experiments, the process tries to find a network with high accuracy irrespective of the number of parameters. We try to reduce the randomness by using selection policies - all children and selected children.

6.2.2 Experiments with Selection strategy - All children, Selected children and Lemonade

For the selection strategy of all children and selection children, there were many experiments run with the fixed parameters shown in table 6.3. The experiments are run with the accuracy metric as well as the weighted objective metric. We mostly concentrate on using weighted objective metric, as accuracy metric focuses on accuracy and in our scenario, the results should also focus on detection rate, false alarm rate and number of parameters. We choose the initial population to be 10 networks, the same as that in aging. The number of offspring is preset to 20, as this will result in 20 mutated networks every generation and nearly 2000 networks for 100 generations for NAS to explore. The number of parents for selection process for every generation is also preset to 20, the same as the number of offspring.

The experiments are run with variations in mutation rates (MR) and different weights assigned to the metrics while using weighted objectives.

#epochs	#Initial population	#parents	#offspring	#generations
5	10	20	20	100

Table 6.3: *Parameters for selection strategy - All children, Selected children*

These experiments help us to observe the effect of changes in the weights as well as mutation rate in the performance of the NAS process. We try to concentrate on reducing the number of parameters keeping in mind that the objective is to have models with least number of parameters with detection rate greater than 90% and false alarm rate less than 20% simultaneously. This would also result in good accuracy, as high detection rate and low false alarm rate are analogous to good accuracy. The mutation rate and weights for every experiment are mentioned along with the results.

For all children, we run an experiment with the lower and upper bound for mutation rate, for a bad parent, set to 70% and 90% respectively, as this would facilitate the offspring to greatly vary from the parent. For a decent parent, the upper and lower bound is 10% and 30% where as for a good parent it is 5% and 10%, as this would make the offspring not to vary a lot from the parent (Table 6.4). After running an experiment with the weights for the metrics - detection rate, false alarm rate and number of parameters set to equal weights of one, which did not produce great results, we set these metrics values to three, three and one respectively.

Increasing the weights for detection rate and false alarm rate would penalize those networks with detection rate below and false alarm rate above the threshold values respectively. This would also emphasize more on the detection rate and false alarm rate for the best weighted network (BWN), while trying to reduce the number of parameters. The results for these experiments with and without the seed model are mentioned in Figure 6.3 and Figure 6.4, respectively. The x-axis for both the graphs define the generation number where as, the y-axis define the detection rate/false alarm rate on one axis and number of parameters on the other.

MR bad parent	MR decent parent	MR good parent	Weight detection rate	Weight false alarm rate	Weight #Parameters
70% - 90%	10% - 30%	5% - 10%	3	3	1

Table 6.4: *Mutation rate (MR) and weights for metrics*

Although the seed model (generation 0) has very high detection rate and low false alarm rate, it still has more than 40,000 parameters. The evolution process, using these weights, tries to further reduce the number of parameters but by keeping detection rate and false alarm rate above and below the threshold values. For the BWN of every generation, the

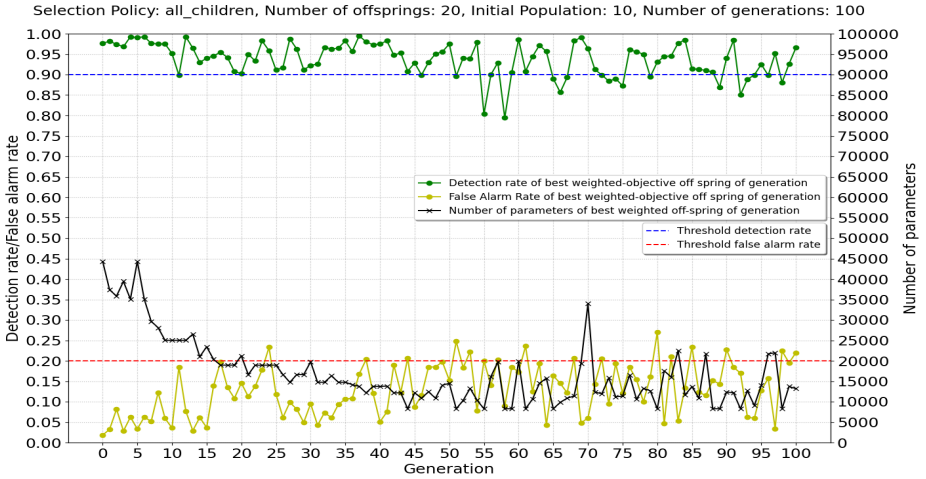


Figure 6.3: *NAS with selection strategy - all children with seed model*

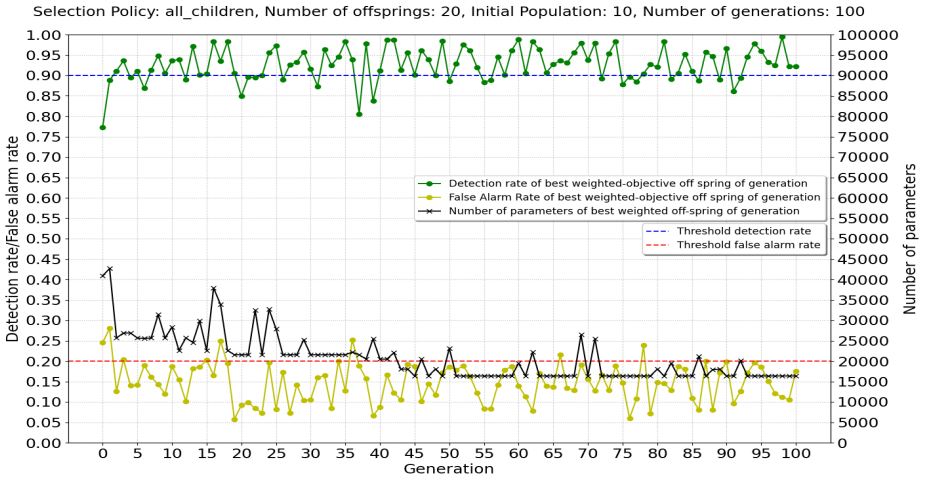


Figure 6.4: *NAS with selection strategy - all children without seed model*

number of parameters reach below 20,000 mark in the 17th generation and remain below this value for most of following generations. Further, there are nearly 15% BWNs that have less than 10,000 parameters for this experiment with the network in the generation 59 having around 8,000 parameters with other metrics above and below the threshold. There are exceptions for few generations, but the general downward trend of the number of parameters is evident from Figure 6.3.

When no seed model is used, the evolution process starts with random

models with BWN producing nearly 75% detection rate and 25% false alarm rate for the initial population (generation 0). This improves in the later generations as the detection rate and false alarm rate reach above and below the threshold values in the third generation, although it takes another eight generations (in the 11th generation) to reduce the number of parameters further. The number of parameters reach below 20,000 mark in the 43th generation (26 more generations than in the experiment with seed model) and reaches the minimum of nearly 16,000 in the 45th generation maintaining the detection rate and false alarm rate above and below the threshold values. For the majority of the next generations, the number of parameters for the BWN remains the same at around 16,000 (as shown in Figure 6.4).

We can observe that the experiment with the seed model produces BWN models with lower number of parameters (15% BWN models have less than 10,000 parameters) when compared to that without the seed model. The detection rate and false alarm rate for both the experiments follow almost the same pattern of trying to maintain values above and below the threshold values. The evolution process for the two experiments produces nearly 75% and 50% BWNs with high detection rate and low false alarm rate with less than 20,000 parameters, less than half of the number of parameters of the seed model.

To observe the effect of variations in mutation rate, we run an experiment (without the seed model) with changes in mutation rate but with the same weights used in the previous experiments. The lower bound for mutation rate for a decent parent is increased by 5%, whereas both the lower and upper bound for the good parent is increased by 5% (described in table 6.5). This would increase the variations for a child from a decent and a good parent. The result is displayed in Figure 6.5.

MR bad parent	MR decent parent	MR good parent	Weight detection rate	Weight false alarm rate	Weight #Parameters
70% - 90%	15% - 30%	10% - 15%	3	3	1

Table 6.5: *Mutation rate and weights for metrics*

In this experiment, the detection rate and the false alarm rate goes above and below the threshold values in the 11th generation (when compared to 3rd generation in the previous experiment). Also, the number of parameters for this network is nearly 35,000 which is higher when

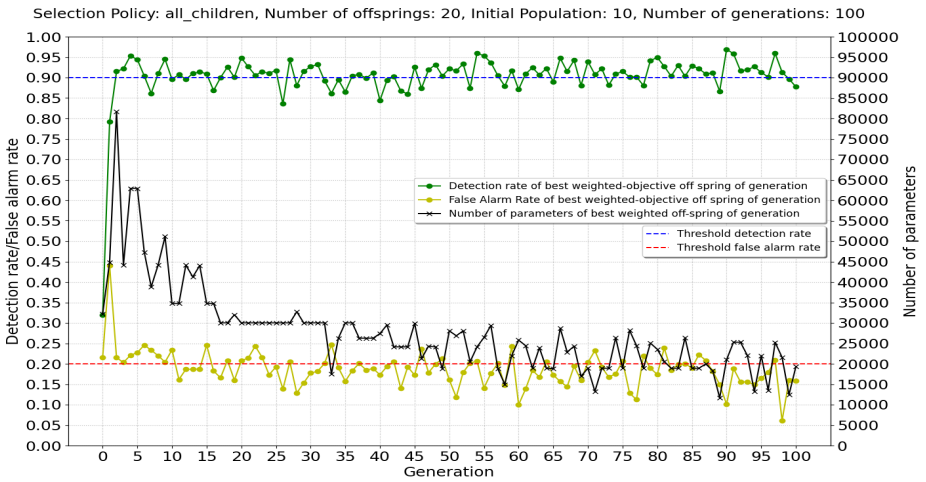


Figure 6.5: *NAS with selection strategy - all children without seed model with higher mutation rate*

compared to the previous experiment. The number of parameters for the BWN does not remain constant, which was the case in the previous experiment. Also, the number of parameters does not decrease at the same rate. It can also be observed that nearly 60% of the detection rate and false alarm rates are above and below the threshold values for the BWN where as, in the previous experiments, nearly 75% of the detection rate and false alarm rate values were, respectively, above and below the threshold values. This experiment, thus, tell us that choosing the lower mutation rate is essential for producing good networks.

Modification in weights also impacts the performance of the evolution process. To observe these effects, two more experiments, one with seed models and another without it is run with the weights described in table 6.6. The weight for detection rate and false alarm rate is increased to 5 (from 3), as this would penalize those networks with lower than threshold detection rate and higher than threshold false alarm rate, more, when compared to previous experiment. With these changes, the expectation is to find the BWNs to have better detection rate and lower false alarm rate with less number of parameters and also to find more such networks when compared to the previous experiments (Figure 6.3 and Figure 6.4). The results for these experiments are displayed in Figure 6.6 and Figure 6.7.

The results also validate this intuition as increasing the weights for detection rate and false alarm rate increase the detection rate and decrease the false alarm rate for the BWNs. Nearly 90% of the networks

MR bad parent	MR decent parent	MR good parent	Weight detection rate	Weight false alarm rate	Weight #Parameters
70% - 90%	10% - 30%	5% - 10%	5	5	1

Table 6.6: *Mutation rate and weights for metrics*

for the seed model and 80% without seed model have detection rate and false rate above and below the threshold values. For the seed model experiment, the number of parameters reduce to below 20,000 mark in the 16th generation (compared with around the 20th generation in the previous experiment), where as for the experiment without seed model, this result is achieved after the 60th generation (compared to 43rd in the previous experiment). For both these networks, the detection rate is higher and false alarm rate is lower when compared to the previous experiment. Also, 35% of the networks for the experiment with the seed model have less than 10,000 parameters (compared to 15% in the previous experiment). There are 8% networks with nearly 10,000 parameters for the experiment without the seed model, which was not the case in the previous experiment (in which the network with least number of parameters has nearly 16,000 parameters). Further increasing the weights did not provide good results as the number of parameters for the BWN also increased.

To further see the effects of weights for the evolution process, one more experiment where we reduce the weight for detection rate and false alarm rate to 3 (from 5) and increase the weight for the number of parameters to 5 (from 1) (Table 6.7) is run. In this experiment, more bias is provided to number of parameters when compared to detection rate and false alarm rate, which means that the networks with more parameters will be penalised higher than the other metrics. This would also result in the BWN to have more importance to number of parameters and less to detection rate and false alarm rate. The results are provided in Figure 6.8. The results are not very good as during the selection phase, the algorithm gives more emphases to parents with less number of parameters even if they don't have high detection rate and low false alarm rate, thus reducing the probability of producing a good offspring network. This experiment also informs that more significance should be given to detection rate and false alarm rate than the number of parameters.

We run an experiment for selection strategy - lemonade. The threshold

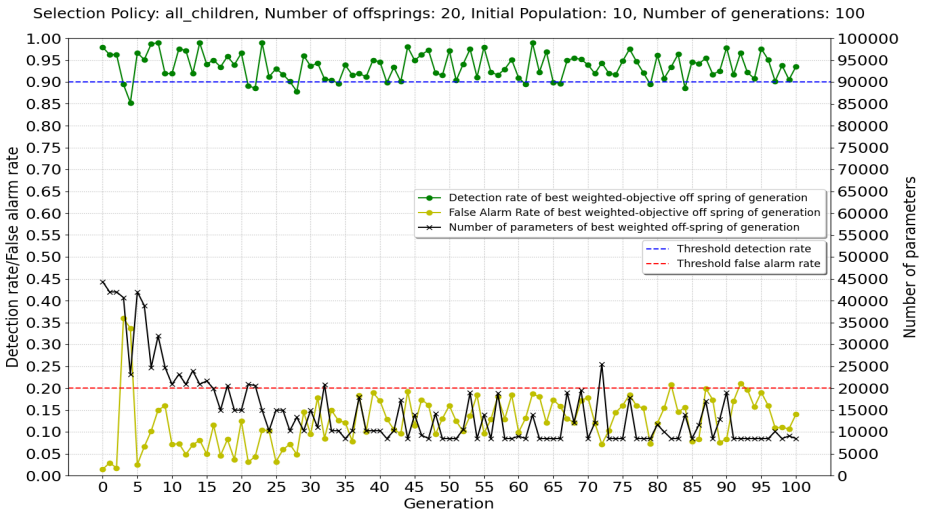


Figure 6.6: *NAS with selection strategy - all children with seed model with higher weights for detection rate and false alarm rate*

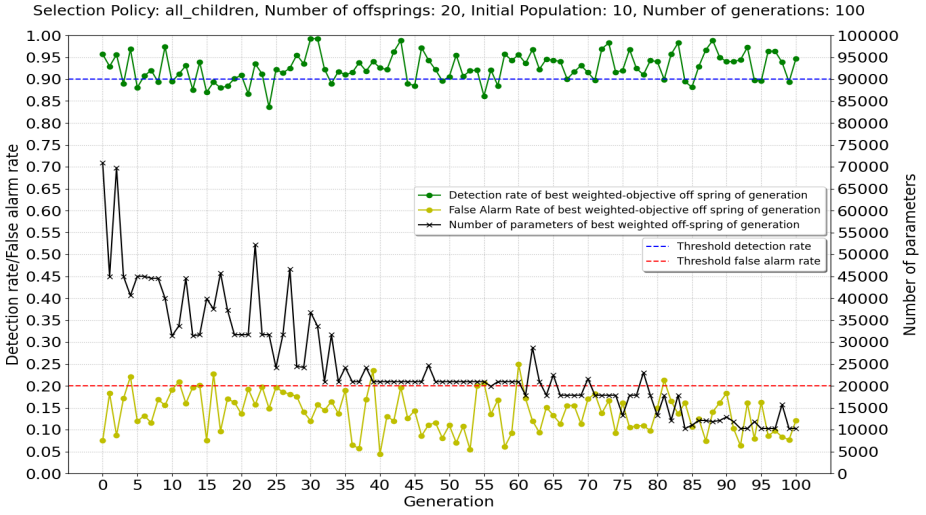


Figure 6.7: *NAS with selection strategy - all children without seed model with higher weights for detection rate and false alarm rate*

accuracy for mutation rate for a good parent is set at 80%, the same as previous experiments. The lower and upper bound for mutation rate for a bad parent to 70% and 90% respectively, the same as previous experiment. This would facilitate the offspring to greatly vary from the

MR bad parent	MR decent parent	MR good parent	Weight detection rate	Weight false alarm rate	Weight #Parameters
70% - 90%	10% - 30%	5% - 10%	3	3	5

Table 6.7: *Mutation rate (MR) and weights for metrics*

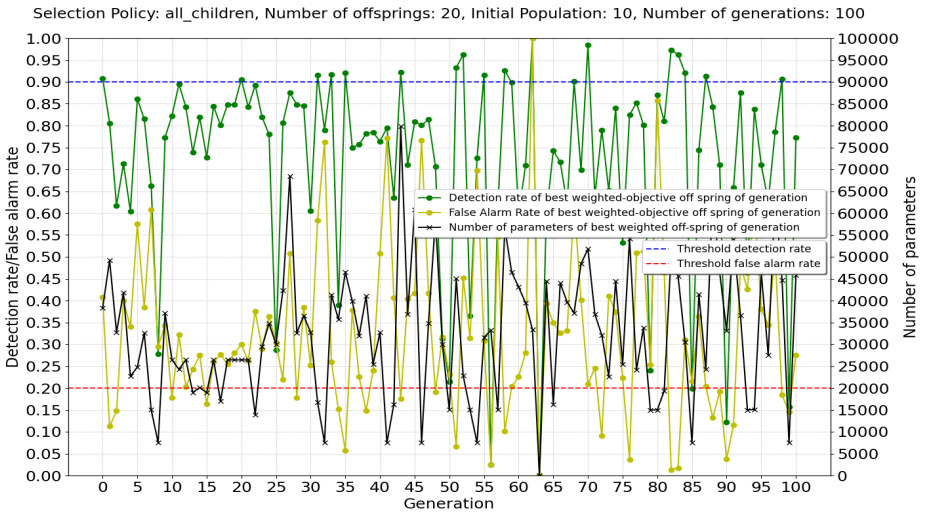


Figure 6.8: *NAS with selection strategy - all children without seed model*

parent. For a decent parent, the upper and lower bound is 10% and 30% where as for a good parent it is 5% and 10%, again the same as the previous experiments. These parameters are shown in table 6.8. We run this experiment with the accuracy measure as the Pareto front is performed on number of parameters and validation error (1 - accuracy) [EMH18a]. The result is shown in Figure 6.9. The Pareto front is plotted for generations 1, 10, 50 and 100, as this would indicate, roughly, the progress of the evolution process.

As expected from the algorithm, networks along the Pareto-front are added continuously over the generations. We can also observe that the number of parameters and validation error gradually decrease for networks on the Pareto front line, thus shifting the line lower with the increase in the number of generations. This is also according to the intuition that lemonade tries to balance the number of parameters with that of the validation error. In generation 100, we have many networks with

parameters between 25,000 and 30,000 with validation error between 5% and 10%. After obtaining this graph, it is up to the user to select a network according to his choice of choosing a network with less number of parameters or one with less validation error.

MR bad parent	MR decent parent	MR good parent
70% - 90%	10% - 30%	5% - 10%

Table 6.8: *Mutation rate (MR) for lemonade, with accuracy as measure*

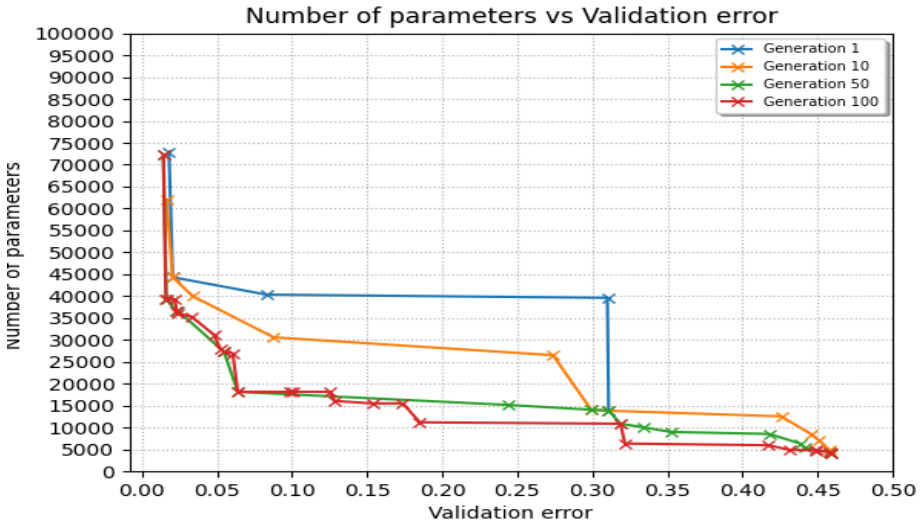


Figure 6.9: *NAS with selection strategy - lemonade with seed model, with accuracy as measure*

Figures 6.10, 6.11, 6.12, 6.13 show the networks for various selection policies with least number of parameters for detection rate greater than 90% and false alarm rate less than 20%. These results are obtained from the various experiments run with various mutation rate and weights.

From these graphs, we can see that aging does not greatly improve the results from that of the seed models as the network with least number of parameters has around 30,000 parameters but also with less detection rate and higher false alarm rate Figures 6.10.

This is not the case with the selection strategy - selected children and all children. With selected children, the number of parameters decrease to around 10,000 maintaining the detection rate of greater than 90% and false alarm rate less than 20% (Figure 6.11). With all children,

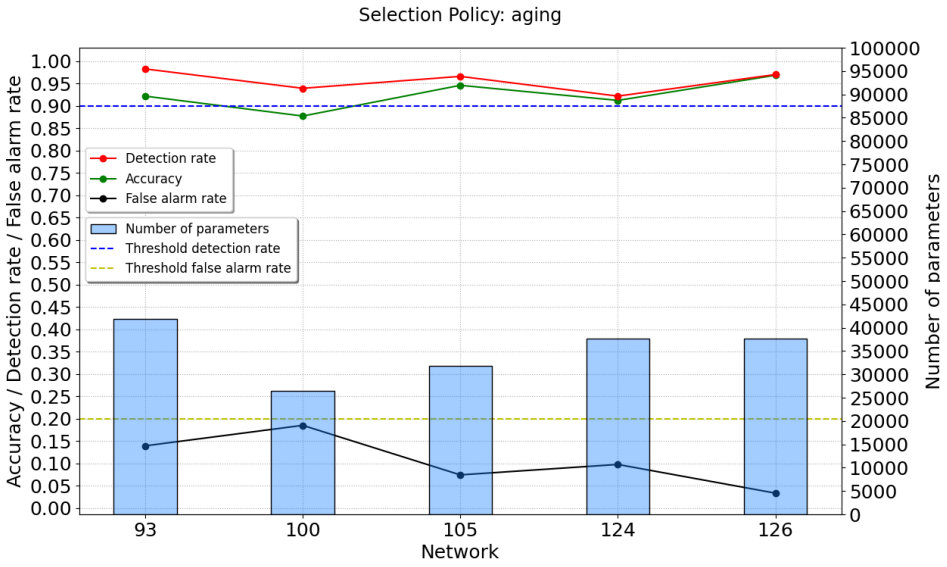


Figure 6.10: *Aging - network with least parameters for detection rate greater than 90% and false alarm rate less than 20%*

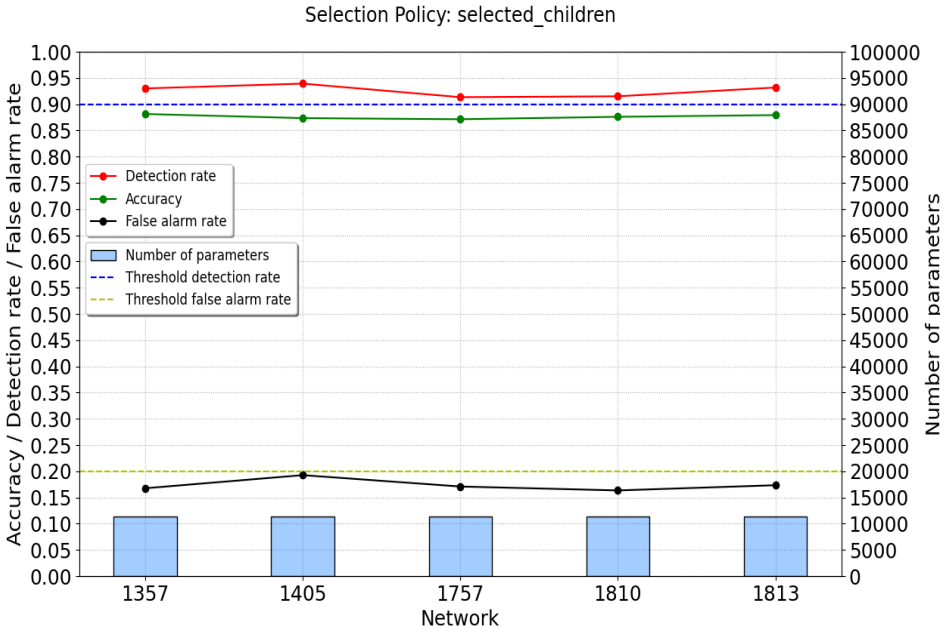


Figure 6.11: *Selected children - network with least parameters for detection rate greater than 90% and false alarm rate less than 20%*

the number of parameters further decrease with many networks having parameters less than 10,000 maintaining the constrains on detection rate and false alarm rate (Figure 6.12, Figure 6.13). These networks have less than one fourth of the number of parameters of the seed model.

The selection strategy of lemonade produces the best accuracy for a network at 98.6% (table 6.9). This network has a very high detection rate and low false alarm rate, but also has higher number of parameters compared to selected children and all children. For the network with minimum number of parameters with detection rate greater than 90% and false alarm rate less than 20%, this strategy produces a network with 18,163 parameters. This network has better accuracy, detection rate and false alarm rate than all the other selection policies but it also has higher number of parameters than selected children and all children.

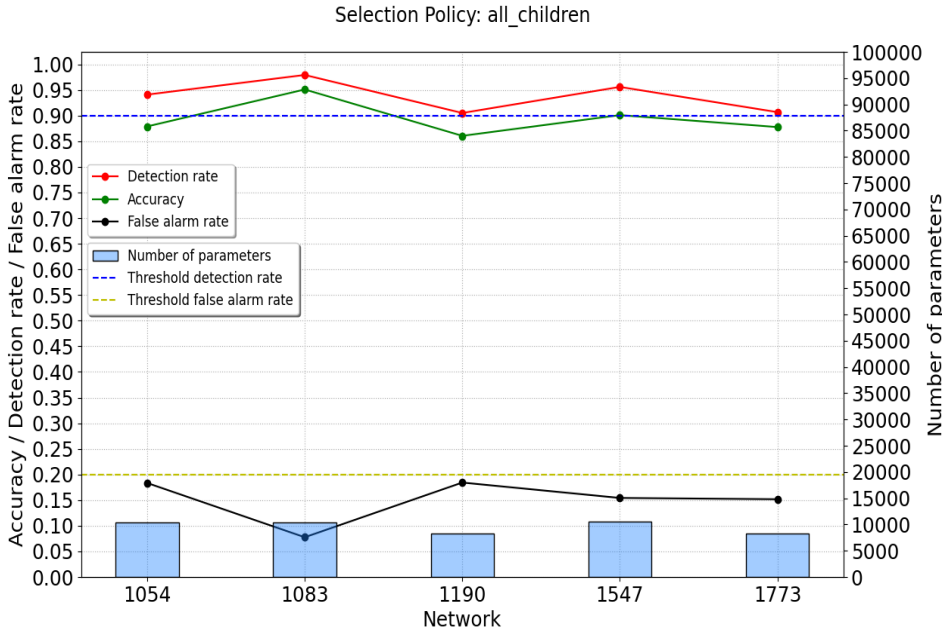


Figure 6.12: *All children - network with least parameters for detection rate greater than 90% and false alarm rate less than 20%*

Further results for the selection policies are described in table 6.9 and table 6.10. Table 6.9 shows the network with the highest accuracy for all the selection policies with other respective metrics of that network, where as table 6.10 displays the network with the least number of parameters for detection rate greater than 90% and false alarm rate less than 20%. These are the cumulative results for all the experiments performed. These results indicate that for both accuracy and for number of parameters, we

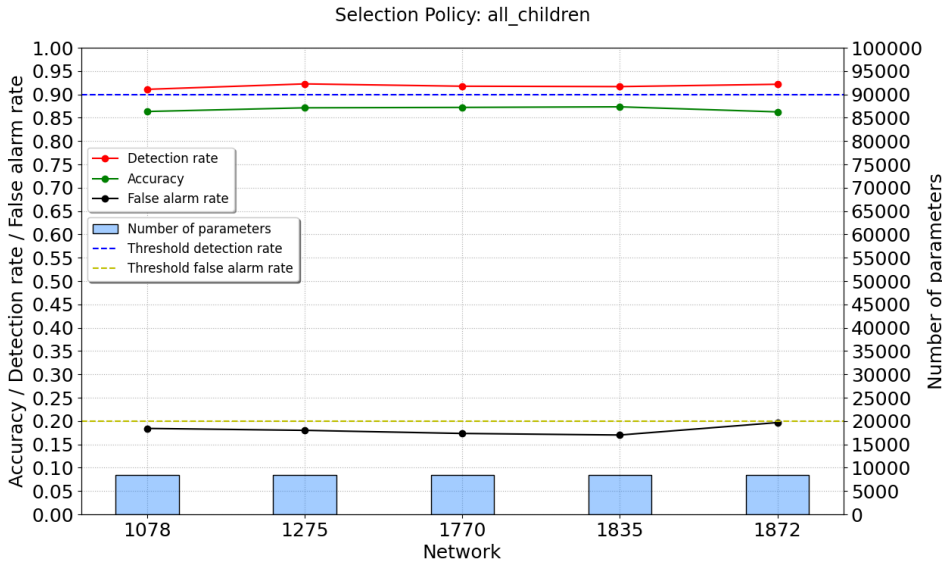


Figure 6.13: *All children - networks with parameters less than 10,000 for detection rate greater than 90% and false alarm rate less than 20%*

get better results using the NAS process when compared to the seed models.

Selection strategy	Generation Number	Accuracy	#Parameters	Detection Rate	False Alarm Rate
Aging	24	0.981	98,849	0.984	0.023
Selected children	3	0.985	64,673	0.985	0.014
All children	2	0.989	69,113	0.985	0.007
Lemonade	19	0.986	72,249	0.99	0.019

Table 6.9: *Networks for selection strategy with maximum accuracy*

Selection strategy	Generation Number	#Parameters	Accuracy	Detection Rate	False Alarm Rate
Aging	81	26,409	0.877	0.939	0.185
Selected children	98	11,219	0.893	0.903	0.117
All children	59	8,273	0.860	0.905	0.184
Lemonade	50	18,163	0.936	0.932	0.059

Table 6.10: *Networks with minimum number of parameters for detection rate greater than 90% and false alarm rate less than 20%*

7 Conclusion and Future Work

In this chapter, the approaches and the results are summarized. Future work, improvements and open research related to this work are discussed.

7.1 Summary and Discussion

The goal of the Master thesis is to develop an AutoML pipeline for atrial fibrillation (AF) and Sinus signal classification of an anonymous two-channel electrocardiogram (ECG) data. Initially we concentrate on the development of seed deep neural network (DNN) models after thoroughly exploring the data. These seed models help to determine if the convolutional models, with a restriction of the maximum number of parameters, are sufficient for the classification task.

For generating new networks and exploring the search space in AutoML, the concept of Neural Architecture Search (NAS) using genetic algorithms is used. Random mutation is used to generate random networks for the initial population. For the subsequent generations, genetic mutation is used. Genetic mutation extracts the genes of a parent and depending on the mutation rate, mutates the parent's genes to produce new networks. We found out that the mutation rate, controlled by the accuracy and number of parameters of the parent, is decisive in producing good subsequent networks.

Various selection strategies - aging, selected_children, all_children and lemonade are discussed and experiments are run for these strategies with different parameter settings. Aging removes the oldest network during the parent selection process for the next generation whereas selected_children and all_children use pairwise comparison of networks for parent selection. Lemonade uses the concept of Pareto-front for the parent selection process.

All these strategies produce good networks. All children with low mutation rate and weighted objectives as a measure has proven to be the best strategy producing 35% networks with less than 10,000 parameters with detection rate greater than 90% and false alarm rate less than 20%, which is less than one fourth over the seed models with respect to number

of parameters. Thus, NAS, with multiple metrics and especially with all children as selection strategy have improved over the seed models and verified its use for this classification task.

7.2 Future work

There is enormous potential for AutoML for such problems. Additional techniques for further reducing model size include quantization. Quantization is a concept that refers to the process of reducing the number of bits that are used to represent a number. This essentially means that quantization can be used to reduce bandwidth and storage as well as increase arithmetic operation performance. In the context of neural networks and deep learning, the predominant format used for weights and biases has been 32-bit floating-point or FP32. INT8 for weights, activations and biases consumes 4x less overall bandwidth compared to FP32. Additionally, integer computations are faster compared to floating-point computations and the hardware will consume less area and is also energy efficient.

Quantization can be part of the AutoML pipeline to determine the format that can be used without compromising the accuracy. This also plays out for hardware-specific implementations of neural networks. For example, FPGAs play out its strength for quantization as they are not limited to standard floating-point precision like FP32 format or FP16 formats, but can be used to represent any number of bits - such as 6-bit values. To improve accuracy, certain fine-tuning can be used. One such method is called scale factor, which can be used to adapt the dynamic range of the tensor from FP32 format to an integer format. This scale factor can be calculated per-layer but can be further used per-tensor as well.

Additional hardware constraints to further fine-tune the models, such as number of floating point operations (FLOPS), memory consumption and inference time of the models can also be added to the search space.

Another research direction would be to use cell based approach for creating and exploring the search space. For this approach, the architecture for the networks can consists of multiple cells. Each cell would in turn consists of either one or more convolutional layer (with different filter size, kernel size and stride size) along with a pooling layer. Other layers such as sum and concatenation layers can also be part of the cell. Thus, each cell will have its own search space, which would add more flexibility to the process. The cell based genetic approach can then be compared

the differentiable architecture search [LSY18]. Reinforcement Learning is another such technique which can be used to compare with the current methods. As such, there is much more to be explored.

Bibliography

[AK15] Ben Athiwaratkun and Keegan Kang. "Feature representation in convolutional neural networks". In: *arXiv preprint arXiv:1507.02313* (2015).

[ASO19] Miquel Alfaras, Miguel Cornelles Soriano, and Silvia Ortın. "A fast machine learning model for ECG-based heartbeat classification and arrhythmia detection". In: *Frontiers in Physics* 7 (2019), p. 103.

[Bea+19] Brett Beaulieu-Jones, Samuel G Finlayson, Corey Chivers, Irene Chen, Matthew McDermott, Jaz Kandola, Adrian V Dalca, Andrew Beam, Madalina Fiterau, and Tristan Naumann. "Trends and Focus of Machine Learning Applications for Health Research". In: *JAMA network open* 2.10 (2019), e1914051–e1914051.

[Cli+17] Gari D Clifford, Chengyu Liu, Benjamin Moody, H Lehman Li-wei, Ikaro Silva, Qiao Li, AE Johnson, and Roger G Mark. "AF Classification from a short single lead ECG recording: the PhysioNet/Computing in Cardiology Challenge 2017". In: *2017 Computing in Cardiology (CinC)*. IEEE. 2017, pp. 1–4.

[Coa+13] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. "Deep learning with COTS HPC systems". In: *International conference on machine learning*. 2013, pp. 1337–1345.

[Deo15] Rahul C Deo. "Machine learning in medicine". In: *Circulation* 132.20 (2015), pp. 1920–1930.

[EMH18a] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Efficient multi-objective neural architecture search via lamarckian evolution". In: *arXiv preprint arXiv:1804.09081* (2018).

[EMH18b] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural architecture search: A survey". In: *arXiv preprint arXiv:1808.05377* (2018).

[Est+17] Andre Esteva, Brett Kuprel, Roberto A Novoa, Justin Ko, Susan M Swetter, Helen M Blau, and Sebastian Thrun. "Dermatologist-level classification of skin cancer with deep neural networks". In: *Nature* 542.7639 (2017), pp. 115–118.

[Eur+10] Developed with the special contribution of the European Heart Rhythm Association (EHRA), Endorsed by the European Association for Cardio-Thoracic Surgery (EACTS), Authors/Task Force Members, A John Camm, Paulus Kirchhof, Gregory YH Lip, Ulrich Schotten, Irene Savelieva, Sabine Ernst, Isabelle C Van Gelder, et al. "Guidelines for the management of atrial fibrillation: the Task Force for the Management of Atrial Fibrillation of the European Society of Cardiology (ESC)". In: *European heart journal* 31.19 (2010), pp. 2369–2429.

[Fed+19] Igor Fedorov, Ryan P Adams, Matthew Mattina, and Paul Whatmough. "SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 4977–4989. URL: `http://papers.nips.cc/paper/8743-sparse-sparse-architecture-search-for-cnns-on-resource-constrained-microcontrollers.pdf`.

[Fuk80] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological cybernetics* 36.4 (1980), pp. 193–202.

[Fus+01] Valentin Fuster, Lars E Rydén, Richard W Asinger, David S Cannom, Harry J Crijns, Robert L Frye, Jonathan L Halperin, G Neal Kay, Werner W Klein, Samuel Lévy, et al. "ACC/AHA/ESC guidelines for the management of patients with atrial fibrillation: executive summary: a report of the American College of Cardiology/American Heart Association Task Force on Practice Guidelines and the European Society of Cardiology Committee for Practice Guidelines and Policy Conferences (Committee to Develop Guidelines for the Management of Patients With Atrial Fibrillation) Developed in collaboration with the North American

Society of Pacing and Electrophysiology". In: *Journal of the American College of Cardiology* 38.4 (2001), pp. 1231–1265.

[Gul+16] Varun Gulshan, Lily Peng, Marc Coram, Martin C Stumpe, Derek Wu, Arunachalam Narayanaswamy, Subhashini Venugopalan, Kasumi Widner, Tom Madams, Jorge Cuadros, et al. "Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs". In: *Jama* 316.22 (2016), pp. 2402–2410.

[Han+19] Awni Y Hannun, Pranav Rajpurkar, Masoumeh Haghpanahi, Geoffrey H Tison, Codie Bourn, Mintu P Turakhia, and Andrew Y Ng. "Cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms using a deep neural network". In: *Nature medicine* 25.1 (2019), p. 65.

[KA14] Martin Krzywinski and Naomi Altman. *Points of significance: visualizing samples with box plots*. 2014.

[Kar16] Karparthy, Andrej. *Neural Networks 1*. `http://cs231n.github.io/neural-networks-1/`. Online. 2016.

[LeC+99] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. "Object recognition with gradient-based learning". In: *Shape, contour and grouping in computer vision*. Springer, 1999, pp. 319–345.

[LSY18] Hanxiao Liu, Karen Simonyan, and Yiming Yang. "Darts: Differentiable architecture search". In: *arXiv preprint arXiv:1806.09055* (2018).

[Lu+18] Zhichao Lu, Ian Whalen, Vishnu Boddeti, Yashesh Dhebar, Kalyanmoy Deb, Erik Goodman, and Wolfgang Banzhaf. "Nsga-net: A multi-objective genetic algorithm for neural architecture search". In: (2018).

[MTL78] Robert McGill, John W Tukey, and Wayne A Larsen. "Variations of box plots". In: *The American Statistician* 32.1 (1978), pp. 12–16.

[Nur+17] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, et al. "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 5–14.

[Ovt+15] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. "Accelerating deep convolutional neural networks using specialized hardware". In: *Microsoft Research Whitepaper* 2.11 (2015), pp. 1–4.

[PKM17] B Pyakillya, N Kazachenko, and N Mikhailovsky. "Deep learning for ECG classification". In: *Journal of physics: conference series*. Vol. 913. 1. IOP Publishing. 2017, p. 012004.

[Qas+19] Murad Qasaimeh, Kristof Denolf, Jack Lo, Kees Vissers, Joseph Zambreno, and Phillip H Jones. "Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels". In: *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*. IEEE. 2019, pp. 1–8.

[Qay+20] Adnan Qayyum, Junaid Qadir, Muhammad Bilal, and Ala Al-Fuqaha. "Secure and Robust Machine Learning for Healthcare: A Survey". In: *arXiv preprint arXiv:2001.08103* (2020).

[Rea+19] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. "Regularized evolution for image classifier architecture search". In: *Proceedings of the aaai conference on artificial intelligence*. Vol. 33. 2019, pp. 4780–4789.

[RSP19] Elad Rapaport, Oren Shriki, and Rami Puzis. "EEG-NAS: Neural Architecture Search for Electroencephalography Data Analysis and Decoding". In: *International Workshop on Human Brain and Artificial Intelligence*. Springer. 2019, pp. 3–20.

[Rus+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. "Imagenet large scale visual recognition challenge". In: *International journal of computer vision* 115.3 (2015), pp. 211–252.

[Smo17] Dawid Smoleń. "Atrial fibrillation detection using boosting and stacking ensemble". In: *2017 Computing in Cardiology (CinC)*. IEEE. 2017, pp. 1–4.

[SSB14] Hasim Sak, Andrew W Senior, and Françoise Beaufays. "Long short-term memory recurrent neural network architectures for large scale acoustic modeling". In: (2014).

[SSE18] Ahmad Shawahna, Sadiq M Sait, and Aiman El-Maleh. "FPGA-based accelerators of deep learning networks for learning and classification: A review". In: *IEEE Access* 7 (2018), pp. 7823–7859.

[Sun+20] Yanan Sun, Bing Xue, Mengjie Zhang, Gary G Yen, and Jiancheng Lv. "Automatically Designing CNN Architectures Using the Genetic Algorithm for Image Classification". In: *IEEE Transactions on Cybernetics* (2020).

[Sze+16] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. "Rethinking the inception architecture for computer vision". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 2818–2826.

[VSM11] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. "Improving the speed of neural networks on CPUs". In: (2011).

[Wik20a] Wikipedia. *Electrocardiography — Wikipedia, The Free Encyclopedia*. `http://en.wikipedia.org/w/index.php?title=Electrocardiography&oldid=951962333`. [Online; accessed 22-April-2020]. 2020.

[Wik20b] Wikipedia contributors. *Convolutional neural network — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Convolutional_neural_network&oldid=944604348`. [Online; accessed 9-March-2020]. 2020.

[Xie+17] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. "Aggregated residual transformations for deep neural networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1492–1500.

[XSZ17] Zhaohan Xiong, Martin K Stiles, and Jichao Zhao. "Robust ECG signal classification for detection of atrial fibrillation using a novel neural network". In: *2017 Computing in Cardiology (CinC)*. IEEE. 2017, pp. 1–4.

[Yan+15] Jianbo Yang, Minh Nhut Nguyen, Phyo Phyo San, Xiao Li Li, and Shonali Krishnaswamy. "Deep convolutional neural networks on multichannel time series for human activity recognition". In: *Twenty-Fourth International Joint Conference on Artificial Intelligence*. 2015.

[Yıl+18] Özal Yıldırım, Paweł Pławiak, Ru-San Tan, and U Rajendra Acharya. "Arrhythmia detection using deep convolutional neural network with long duration ECG signals". In: *Computers in biology and medicine* 102 (2018), pp. 411–420.

[ZPT17] Martin Zihlmann, Dmytro Perekrestenko, and Michael Tschannen. "Convolutional recurrent neural networks for electrocardiogram classification". In: *2017 Computing in Cardiology (CinC)*. IEEE. 2017, pp. 1–4.