Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Eivind Børstad

Evolving Deep Neural Networks using Genetic Algorithms

Master's thesis in Computer Science
Supervisor: Keith L. Downing
June 2020

NTNU
Norwegian University of Science and Technology

Eivind Børstad

Evolving Deep Neural Networks using Genetic Algorithms

Master's thesis in Computer Science
Supervisor: Keith L. Downing
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Evolving Deep Neural Networks using Genetic Algorithms

Master's Thesis

Eivind Børstad

Subject Code	TDT4900
Subject Name	Computer Science, Master's Thesis
Advisor	Keith Downing
Semester	Spring 2020
Publishing Date	15.06.2020
Restrictions	Open

Summary

In this master's thesis, neuroevolution has been used to automatically evolve the hyperparameters of neural networks, which usually must be set manually by the user. The proposed methodology differs from most previous work within neuroevolution by evolving these hyperparameters directly and simultaneously. Population diversity and incremental complexification are two aspects of neuroevolution which have been given a particular focus in this thesis.

Three different systems have been implemented to conduct tests, two of which produce dense neural networks, and one which produce convolutional neural networks. The systems have been tested on five different datasets to investigate their general performance. These datasets are three image datasets (MNIST, EMNIST and fashion-MNIST), a chess endgame dataset, and a yeast dataset.

Many different tests have been run on the systems, showing that the methodology is capable of evolving well-performing neural nets. They indicate that the topology, learning rate, learning rate decay and dropout probability are especially valuable to evolve instead of setting manually. Also, it turns out that the use of a diversity measure as part of the fitness function, and incremental complexification, is not really necessary to make the systems perform well.

Sammendrag på norsk (Summary in Norwegian)

I denne masteroppgaven har nevroevolusjon blitt brukt for å automatisk utvikle de fleste av hyperparametrene for nevrale nett, som ellers må settes manuelt av bruker. Den foreslåtte metodologien skiller seg fra de fleste tidligere arbeider innenfor nevroevolusjon ved å utvikle disse hyperparametrene direkte og samtidig. Mangfold i populasjonen og inkrementell kompleksifisering er to apsekter innenfor nevroevolusjon som har blitt viet et spesielt fokus under denne oppgaven.

Tre forskjellige systemer har blitt implementert for å utføre tester. To av disse utvikler tette nevrale nettverk, mens det siste utvikler konvolusjonelle nevrale nettverk. Systemene har blitt testet med fem forskjellige datasett for å undersøke om de fungerer godt generelt. Dette er tre bildedatasett (MNIST, EMNIST og fashion-MNIST), et sjakksluttspillsdatasett, og et gjærdatasett.

Mange tester har blitt kjørt på systemene, og viser at metodologien er kapabel til å utvikle nevrale nett som gir gode resultater. Spesielt topologien, læringsraten, læringsratens reduksjonsrate og "dropout"-raten til de nevrale nettene viser seg å være verdifulle å utvikle, fremfor å sette manuelt. I tillegg viser resultatene at det å bruke et mangfoldsmål som del av evalueringsfunksjonen til den genetiske algoritmen, og å bruke inkrementell kompleksifisering, ikke hjelper systemene å utvikle bedre nevrale nett.

Table of Contents

List of Tables

List of Figures

Terminology

Activation Function: Function applied as the last step in the calculations within a neuron

Aging: Technique used in genetic algorithms penalizing individuals which have existed for a long time

AutoML: Automation of all components part of machine learning

Backpropagation: Algorithm used to update the weights in neural nets

Batch Size: The number of training examples used to update the weights in a neural net

Child Individuals: The individuals created in the current generation of a genetic algorithm

Classification Problem: Problem where the goal is to find the class which something belongs to

Convolutional Layer: Special type of layer which might be used in a neural network, and which is well-suited for image data

Convolutional Neural Net (CNN): Special type of neural net which uses convolutional layers, and is especially suited for tasks involving images

Cost Function: Function which evaluates the performance of a neural network, and which is used to determine how the weights should be adjusted

Crossover: Combining the genes of two (or more) individuals to create new individuals in a genetic algorithm

Deep Learning: Learning using neural nets with more than one hidden layer

Deep Neural Network (DNN): Neural network with more than one hidden layer

Direct Encoding: Genetic encoding where every gene describes a property in the phenotype directly

Diversity: The degree of variation in a population of a genetic algorithm

Dropout: Technique used when training neural nets to avoid overfitting

Elitism: Guarantying that the best individual will be kept in a genetic algorithm

EMNIST: MNIST-like dataset with both letters and digits, see MNIST

Ensembles: Combining several machine learning models in a way making them perform better than they could individually

Fitness Function: Function evaluating the performance of an individual in a genetic algorithm

Fitness Value: Value indicating the performance of an individual in a genetic algorithm, determined by the fitness function

Fashion-MNIST: MNIST-like dataset with pictures of clothing, see MNIST

Feature Engineering: The process of generating new features for a machine learning model, which might improve the results

Features: Input data for a machine learning model, which the predictions will be based upon

Generation: One iteration in a genetic algorithm

Genetic Algorithm: Machine learning technique inspired by biological evolution

Hyperparameter: Parameter which must be set manually before an algorithm can run

Incremental Complexification: To gradually increase the complexity of the topology of the individuals in a genetic algorithm

Indirect Encoding: Genetic coding where each property of the phenotype is not directly described by its own part of the genome

Individual: Possible solution part of the population in a genetic algorithm, described by a set of genes

Labels: The correct output which machine learning models use during training to learn

Learning Rate: How much the weights in a neural network should be adjusted each time

MNIST: Famous dataset with images of handwritten digits which should be classified

Momentum: Technique preventing neural nets from being trapped in local minima

Mutation: Random changes to the genes in a genetic algorithm, which hopefully will explore new parts of the search space

Neural Architecture Search (NAS): General term for the field of finding optimal architectures for neural networks

Neural Network/Neural Net: Machine learning model inspired by biological brains

Neuroevolution: The use of genetic algorithms for finding hyperparameters for neural nets

NeuroEvolution of Augmenting Topologies (NEAT): Probably the most famous system within neuroevolution, introduced in Chapter 3.2.2

Neuron: Component which executes calculations, and which neural networks consist of

Novelty Search: Search for new types of individuals in a genetic algorithm, which might help increase the diversity of the population

One-Hot Encoding: Binary encoding where exactly one bit is 1, and the rest is 0

Optimizer: Techniques which might improve the training of neural nets by avoiding local minima

Overfitting: When a machine learning model is better at predicting the training data than other data

Pairwise Failure Crediting (PFC): Method which measures the diversity of the population in a genetic algorithm by considering every pair of individuals

Phenotype: The actual "creature" which the genes are representing

Population: The set of individuals in a genetic algorithm

Recurrent Neural Net (RNN): Special type of neural net that contains cyclic connections, which gives the net a kind of memory

Regression Problem: Problem where the goal is to find a correct continuous value

Reinforcement Learning (RL): To learn by experience through trial-and-error

Search Space: The set of possible configurations which a machine learning model is searching for solutions among

Test Set: Neutral part of the dataset which is only used for a final, independent evaluation of a machine learning model, and which might measure overfitting

Topology/Network Architecture/Network Structure: How the neurons are organized in a neural network

Training Set: The part of the dataset which is used to train a machine learning model

Training Step: Every time the weights are updated in a neural network

Validation Set: The part of the dataset which is used to evaluate a machine learning model before the training is completed

Weight: Numeric values used in the calculations in a neural net, which are adjusted during training

Chapter 1

Introduction

This chapter is based on the introduction chapter included in the midterm report written for this master's thesis (Børstad, 2019, pp. 10-12).

Neural networks have existed for a long time, but never received the same amount of attention as for the last few years. Deep learning, the usage of large, complex neural nets, has finally shown its potential due to the exponential increase in computational power available over time. This again has resulted in great breakthroughs.

A problem with such deep neural nets (DNNs) is the many hyperparameters which must be determined before they can be used. This includes the network structure, how fast and for how long the nets should be trained, how the calculations should be executed, and so on. As an example, finding the optimal network structure is an NP-complete problem (Hwang, Choi & Park, 1997, p. 1). As a result, designing well-performing networks requires a combination of human expertise and time to execute trial-and-error. Automatic tuning of these hyperparameters can therefore be regarded as very attractive, and has already been tried for decades using many different methods (He, Zhao & Chu, 2019, pp. 9-13). In general, this is called Neural Architecture Search (NAS).

One of the most successful techniques within NAS is neuroevolution. This method is inspired by biological evolution, and evolves neural networks based on a genetic algorithm. Through so-called "generations", where the hyperparameters of different neural nets are mixed and mutated, continually better nets will hopefully evolve.

Methods within neuroevolution have already achieved great results, but many of the systems explore only a small part of the field, either by picking one or a few hyperparameters to evolve, or by specializing toward one very specific problem domain (e.g. Patel (1996) and Real et al. (2017); more previous work is presented in Chapter 3). This master's thesis investigates whether or not a system based on a genetic algorithm may work well in general to evolve neural networks which achieve good results in a variety of problem domains. If so, the difficult choice of hyperparameter values will no longer be left to the user.

The general research goal for the thesis has been defined as follows:

Investigate the use of genetic algorithms to evolve neural networks.

Based on this goal, the following four research questions have been defined:

1. To what degree can well-performing neural networks be created by directly evolving most of the hyperparameters which normally must be set manually?
2. Which hyperparameters are the most critical to evolve for the neural networks to be well-performing?
3. How important is it to include diversity as part of the fitness function when it comes to evolution of such neural networks?
4. What are the advantages of starting with a minimal network structure, and then perform incremental complexification, instead of starting with more complex networks?

Each of the four research questions will now be explained in more detail. A few technical terms are introduced here. See chapters 2 and 3 for a more detailed explanation of these terms.

The first question is the most general one, and covers the main portion of the work. It concerns the use of a genetic algorithm to discover optimal values for hyperparameters which otherwise must be set by the user through trial-and-error. Examples of such hyperparameters are network structure, activation function and learning rate. By the word *directly*, it is meant that the hyperparameter values themselves are being evolved, instead of a function which later will determine the values (see e.g. Stanley et al. (2009) presented in Chapter 3.2.2), or module-like components which later will be combined into actual neural nets (see e.g. Miikkulainen et al. (2019) presented in Chapter 3.2.2). The approach also differs from most of the previous work within neuroevolution as it tries to evolve all hyperparameters at the same time, instead of just the network structure and/or weights. In this project, the weights of the neural nets will not be set through evolution, but instead be tuned by the backpropagation algorithm (see Chapter 2.1.1).

Some hyperparameters are probably more valuable to evolve than others. The second research question will investigate this. By "turning off" the evolution of one specific hyperparameter at a time, and hard code the value of this one instead, one can investigate what hyperparameters are the most critical to evolve in order to achieve good results.

Genetic algorithms often face the problem that the population becomes very uniform with similar individuals, something which in turn may prevent new, promising ideas from being discovered. To prevent this, there exist techniques which ensure diversity in the population. The third research question will investigate if rewarding more unique and diverse individuals during evaluation of the networks has a positive effect on the results. Diversity is introduced in Chapter 3.2.3.

The fourth question investigates the importance of incremental complexification. This means starting with very small neural nets in the first generation of the genetic algorithm, and then letting the networks mutate larger and more complex over time. Larger networks might have a higher chance of overfitting to training data, meaning trying small nets first might be an advantage.

To answer the research questions, three systems have been implemented and used to run tests within various problem domains. This includes a main system evolving dense neural networks, a merging subpopulations system applying an additional technique for ensuring diversity, and a convolutional system evolving convolutional neural networks. To implement a system and run tests is the most suitable way of answering the research questions, as it is practically impossible to theoretically prove how well the proposed methodology works in any other way.

The rest of this thesis will introduce related theory, describe the implemented systems, and present and analyze the results acquired. Chapter 2 will cover general background theory. In Chapter 3, previous related work will be presented. The systems created are described in Chapter 4. The results acquired by running tests with these systems will then be presented and analyzed in Chapter 5. Finally, in Chapter 6, conclusions will be drawn, and ideas for future work will be discussed.

Chapter 2

Background

In this chapter, relevant background theory needed to understand the thesis is presented. Chapter 2.1 covers the underlying machine learning tools used. The problem domains which the systems are tested in are described in Chapter 2.2. Finally, the background theory is summarized in Chapter 2.3.

2.1 Underlying Machine Learning Tools

This part introduces the two components which together make up the field neuroevolution: neural nets (Chapter 2.1.1) and genetic algorithms (Chapter 2.1.2). Information about how to combine the two, as well as more complex techniques that may be used, is located in Chapter 3 along with the related work. This subchapter is based on the corresponding part in the midterm report for this master's thesis (Børstad, 2019, pp. 13-21).

2.1.1 Neural Networks

A neural network is a machine learning method which has been given a lot of attention over the last few years, due to its impressive results within several fields of work (Tyantov, 2017). Inspired by biological brains, it consists of neurons connected in a large network. Every neuron is a unit executing certain mathematical operations. The result of such a calculation is sent to other neurons which are connected to this neuron. Therefore, neural nets are technically directed graphs.

Traditionally, neurons are organized in layers, meaning every neuron in a layer receives a value from every neuron in the layer before, and sends its own result to every neuron in the layer after. The neurons in the first layer contain the input values for the problem at hand, while the neurons in the last layer contain the values which the neural network has calculated as the answer to the problem. These two layers are usually simply referred to as the input layer and output layer. Every layer between these are called hidden layers (Erb, 1993, p. 165). Figure 2.1 illustrates this. It is worth mentioning that this is not the only way to organize a neural network, even though it is the most common one. Other approaches will be briefly discussed later. A neural network is called a Deep Neural Network (DNN) if it contains more than one hidden layer. The number of layers, number of neurons, and how these are connected are called the topology or structure of the network.

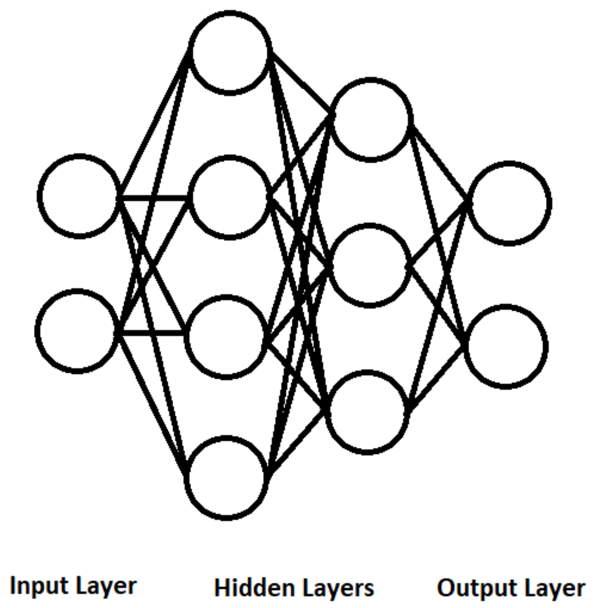


Figure 2.1: A neural network with two hidden layers. Circles represent neurons, and lines represent connections.

There are two main types of tasks which can be solved by neural networks: classification problems and regression problems.

A classification problem is the task of classifying something among a finite set of categories. An example is figuring out what digit a handwritten digit is representing. In this type of problem, the last layer in the network usually contains the same number of neurons as there are categories, and the value of each neuron in this layer represents the probability that the example belongs to the category associated with that particular neuron.

A regression problem, on the other hand, is the task of finding one or more continuous values, like a number, based on the input. An example is finding the sum of two numbers (although of course, for such a simple problem one would just use a calculator, and not a neural net). For regression problems, the last layer usually consists of only a single neuron, whose value represents the predicted answer by the neural net.

In a normal neural network, the calculations executed in every neuron have the general form:

$$y_{l,i} = \sigma((\textstyle\sum_{j=1}^{n} w_{l,i,j} \cdot x_{l-1,j}) + b_{l,i})$$

$y_{l,i}$ represents the output value for neuron i in layer l. $x_{l-1,j}$ is the output value from neuron j in the layer before. For every pair of neurons directly connected, there exists a weight w, determining the importance of the connection. In the same manner, every neuron has a bias, b, which is added directly to the value of the neuron. σ is an activation function, something which will be explained soon. In summary, every product of values from the neurons in the layer before and their associated weights are summed, and then the bias is added. This sum will be used as input to the activation function, which determines the final output value from this neuron.

The main task of the activation function is to remove the linear relationship between inputs and outputs. It turns out neural nets without an activation function (that is, uses the identity function as activation function) only can represent linear relationships between the input and output data, something which severely reduce their usefulness (Erb, 1993, p. 167). Two of the most common activation functions will be introduced next. For a more thorough list of activation functions, see Sharma, Sharma and Athaiya (2020).

Sigmoid, also called the logistic function, was previously the leading activation function, and is defined as $Sigmoid(x) = \frac{1}{1+e^{-x}}$. It follows a characteristic S-curve. The main problem with Sigmoid is the vanishing gradient problem, which is explained by Nielsen (2015, Chapter 5).

ReLU (Rectified Linear Unit) has to a large degree been Sigmoid's successor, due to its simplicity and the fact that it is not exposed to the vanishing gradient problem. It is defined as $ReLU(x) = max(x, 0)$. This means it consists of two straight lines, with a bend in the origin.

When the network topology, activation functions and other hyperparameters have been set, only the values of weights and biases can influence the output given the input. These must be tuned in order to make the net perform well. This process is called training the neural network, and usually happens through the use of the backpropagation algorithm.

During training, a set of training examples are used to adjust the weights and biases. In the following explanations, both weights and biases are included in the term weights, if not otherwise specified. The weights are usually initialized with random values before training. The training examples consist of both features, which are input data, and labels, which are the correct answer corresponding to the features. This separates training data from what the neural network will have available when it later will be used for actual predictions; at that point, only features will be available, and no labels. By making a prediction based on the features and current weights, it is possible to calculate how far away from the labels the predictions currently are.

Exactly how the distance between the current predictions and the labels is calculated, depends on the chosen cost function. For a classification task, this could be as simple as counting the number of correct predictions, but as it turns out, more sophisticated cost functions can make the training easier. A very common cost function is Mean Square Error (MSE). The difference between the correct and predicted value is squared for every neuron in the final layer, and the average of these are calculated. A larger value means a larger error. For a classification task, the labels will consist of a 1 for the neuron of the correct category, and a 0 for all other categories. This is called a one-hot encoding. It means that even though a sample is classified correctly, it might still get a relatively high value from the cost function if the network is in doubt. From this the weights can be adjusted to make the neural network be in less doubt for future predictions. In other words, it is possible to learn even from a correct prediction, something which might be an advantage. For a list of more cost functions, see Yin (2017).

Based on the error determined by the cost function, the backpropagation algorithm is used to find a gradient telling how much the different weights in the network must be adjusted to correct that error. Because the backpropagation algorithm is quite complicated, and not important for understanding the rest of this thesis, the reader is referred to Nielsen (2015, Chapter 2) for an in-depth description of it.

The calculated gradient adjusts the weights, reducing the error for a future prediction similar to this one. Several hyperparameters and choices determine exactly how this adjustment works, each of which will be explained in the following paragraphs.

The learning rate determines how much the weights should be changed at once, and is simply a positive number which is multiplied with the gradient before the weights are adjusted. A higher learning rate may result in a faster training, but also increases the probability of skipping a minimum, and causing the weights to oscillate between different values, or falling out of a minimum completely (Zeiler, 2012, p. 2). If the reader is unfamiliar with the concepts of local and global minima, see Brownlee (2019b, "1. Local Minima").

It might be an advantage to let the learning rate decrease as the training goes on. This is because early on, the weights are far from correct and need to be adjusted a lot. Later, the weights are already in well-performing states, and only need small adjustments in order to make the neural net perform even better. A policy of decreasing the learning rate over time is called a learning rate schedule. Several such schedules exist, including time-based learning rate schedules and exponential learning rate schedules. See Lau (2017) for more information.

In its simplest form, the weights may be adjusted by just adding the calculated gradient, multiplied by the learning rate. However, under most circumstances, more sophisticated techniques are required to achieve good results. One such technique is the use of momentum. With momentum, the weights will be changed more if they already have been changed a lot in the same direction in recent past (Zeiler, 2012, p. 2). This makes the weights adjust quickly when they need to be changed a lot in a specific direction, and slower when they should be fine-tuned in the end. This can in turn help avoiding the oscillating effect mentioned earlier. An analogy to this is a ball rolling down a hill. The ball gains speed and changes position faster while it rolls downward, and slows down when it faces an ascent.

Whether such momentum or other advanced techniques should be used or not, is determined by the chosen optimization technique, also called optimizer. An example of an optimizer using momentum is Adagrad. Additionally, this method uses a higher learning rate for weights associated with features used with low frequency, making the method work well for datasets with limited data. For a detailed overview of such optimizers, see Ruder

(2017, pp. 4-9).

A question yet to be addressed, is how frequently the weights should be updated. Mainly, three alternatives exist. These will be described now.

Stochastic gradient descent (SGD) selects a random training example, and adjusts the weights based on this example only. This method updates the weights very quickly, but as it is based on only one example, the adjustments are not necessarily improving the network with respect to the problem to be solved in general. On the other hand, this randomness could have the positive effect of escaping local minima. In practice, SGD almost always converges to a local or global minimum if the learning rate is slowly decreased during training. However, it often takes a very long time before this happens (Ruder, 2017, p. 2).

Batch gradient descent is the opposite of SGD. Instead of selecting a single training example as base for the weight adjustments, the whole training set is used. This means every single example must be evaluated for every training step. The advantage of this is that the weights will always be adjusted in the way that reduces the cost function error the most for the whole training set, meaning this method will always converge to a local minimum. The disadvantages are that every update takes much more time and requires more memory, and that the lack of randomness stops the possibility of escaping local minima (Ruder, 2017, p. 2).

Mini-batch gradient descent is a hybrid of the two methods mentioned above. Instead of using a single training example, or all of them, n examples are used to update the weights. The value of n is often referred to as the "batch size". This method extracts the positive properties of both methods above: it is relatively quick, requires less memory, contains some randomness to escape local minima, and uses enough training examples to give a good representation of the whole set. In practice, this is usually the method used (Ruder, 2017, p. 3).

Each adjustment of the weights is called a training step. If every training example is used once before any example is used for a second time, that is called an "epoch", independently of which of the three methods above that is used. In the case of batch gradient descent, the number of training steps is equal to the number of epochs. The use of epochs is not strictly necessary; one could simply draw random training examples for every step, meaning some examples might be drawn more times than others. However, Bottou (2019) and Gürbüzbalaban, Ozdaglar and Parrilo (2019) indicate that the use of epochs statistically give better results. The length of the training could be a specified number of steps or epochs, but could also be determined by a specific amount of time, or by more sophisticated methods, like stopping the training when no improvement has been seen for a certain number of steps.

The dataset used to train a neural net is usually split into different parts. The training part is the one used to adjust the weights as described previously. However, one wants the neural net to perform well for the selected task in general, and not only for the specific cases trained with. To measure this ability to generalize, the test part of the dataset can be used to evaluate the performance of the network after the training is completed. The test set should contain cases similar to those in the training set, but not the exact same cases. Validation sets are similar to the test set, but are used to evaluate the network during training. Based on the results from the validation set(s), the course for the rest of the training might be changed. Therefore, such sets do not maintain their neutrality like the test set. For more information about the different parts of the dataset, see Shah (2017). The key point is to never use the test set during training, as it will then lose its neutrality, and the results will not be representative anymore.

If the model is better at classifying the examples in the training set than the examples in the testing set, the model is overfitted. Several methods exist to prevent overfitting, three of which will be explained here.

Early stopping is to stop the training when the neural net performs the best, evaluated by a validation set. By running an evaluation after every nth step, one can figure out when the net performed the best during the training phase. After the training is completed, the weights can be rolled back to the values they had when the net performed the best, as long as these values have been saved. Additionally, the training can terminate earlier than planned if no improvement on the validation set has been found in a certain number of steps, in order to save time. Early stopping is an advantage because networks trained for too long often starts overfitting. In other word, the performance on the training set keeps improving as training goes on, while it after a while starts to impair for independent validation and testing sets (Orr & Müller, 1998, p. 51). Figure 2.2 shows this general tendency.

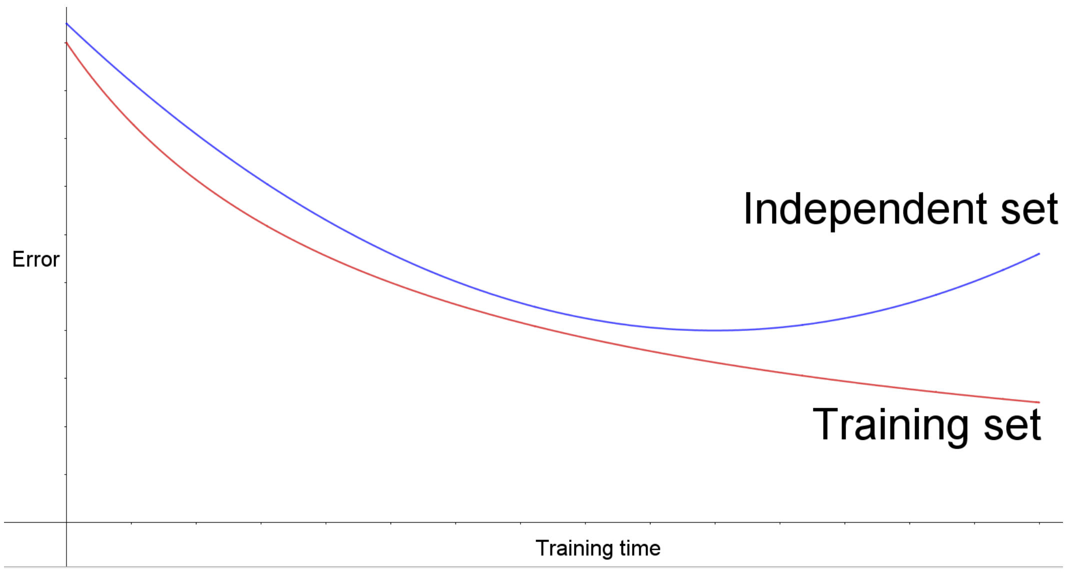


Figure 2.2: Impact on performance by the number of training steps for a training set and an independent set.

Batch normalization can both speed up training, and prevent overfitting. The output of a neuron is normalized based on the mean and standard deviation of all the neurons in the layer for the current mini-batch. For more theory about batch normalization, see Ioffe and Szegedy (2015).

Dropout is another method for preventing overfitting. This is done by randomly ignoring the output from certain neurons at specific training steps. The actual value of that neuron is simply replaced by a 0. It might not be very intuitive, but it turns out this makes the networks more robust, as they must be able to perform even without some of their neurons, something which in turn might prevent overfitting (Srivastava, 2013, pp. 1-2). For every training step, new neurons are dropped. Whether or not a neuron is dropped is drawn independently based on a dropout probability/rate, whose value determines the probability of dropping each neuron.

Finally, it should be mentioned that there exist other types of neural nets than what has been described here. Much of this applies in general, but certain properties differ. The main difference is how the layers of the net are structured. In a dense neural net, like the one described here, all neurons in every layer are connected to all neurons in the layer before and after, and every connection between two neurons has its own weight. Contrary to this, convolutional neural nets (CNN) have other types of layers, including convolutional layers and pooling layers. Here, several connections between neurons share a weight. Recurrent neural nets (RNN) contain connections to neurons in previous layers, allowing signals to go in a loop, something which gives the network a kind of memory. CNNs are evolved by one of the side systems presented in this thesis. However, as it is only a small part of this work, an in-depth explanation is left to Ahire (2018, Chapter 8). For an overview of RNNs, Venkatachalam (2019) is recommended, however, it is not particularly relevant for this thesis.

2.1.2 Genetic Algorithms

Genetic algorithms are also very much inspired by biology, and specifically how biological evolution works. The fact that nature is capable of designing such well-performing creatures based on the simple process of evolution, is quite remarkable, and indicates that similar methods might work well for other problems as well.

In its simplest form, genetic algorithms work approximately like this: A population of possible solutions exists, and is often randomly generated at the start of the run of the algorithm. The population consists of many individuals, each specified by its genes. The genes describe what the solution of this individual looks like. Exactly how the genes are represented may vary, but they are often just strings of binary numbers. For every iteration of the algorithm, called a generation, a set of operations are performed on the population. First, individuals are selected to be parents for the upcoming generation's individuals. Often, the best individuals are selected as parents, but other factors might also be considered. New child individuals are usually created from the parents through two operations: crossover and mutation. Crossover is the process of mixing the genes of two parents, creating a new set of genes. Mutation is the process of changing the genes of an individual randomly with a certain probability. Usually, a new individual is created by first performing crossover, and then mutation (Whitley, 1994, p. 68). As seen here, all terms are quite intuitive, and are derived from biological evolution.

As the last step of the algorithm in each generation, the worst individuals are removed from the population. After this, one has a new population ready for a new generation. It is important to emphasize that both new child individuals and older individuals, including those selected as parents, might be kept and removed, meaning some individuals might stay in the population for many generations. In other versions of the algorithm, no individuals are removed, meaning the population grows larger and larger for every generation.

When it comes to determining what are the ”best” and ”worst” individuals in the population, this is done through the use of a fitness function. Within neuroevolution, the fitness function will build, train and evaluate the neural networks represented by the genomes. The accuracy of the trained neural nets will affect the fitness value. In its simplest form, ”affect” might here mean that the accuracy will solely determine the fitness value. However, one could also include other factors, like diversity, something which is introduced along with the related work in Chapter 3.2.3. It is worth emphasizing that it is the neural net which the genes represent, and not the genes themselves, which the fitness value is based upon. More technically, one could say the fitness is based upon the phenotype instead of the genotype. This is quite common for genetic algorithms.

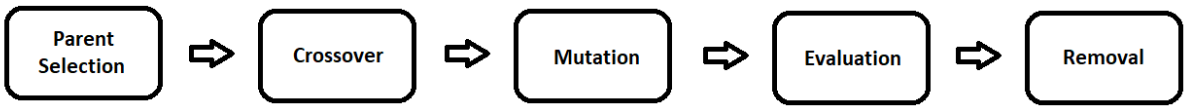


Figure 2.3: The main steps of a generation in a genetic algorithm.

Figure 2.3 summarizes these main steps of a generation in a general genetic algorithm. The rest of this subchapter will give some more detailed information about the following components of genetic algorithms: initialization of the population, gene representation, selection of parents, crossover, and mutation.

In order to make the genetic algorithm capable of evolving new individuals, it first requires an initial population as a starting point for the first round of parent selection. Countless techniques exist for generating this initial population. The simplest and most common idea is to generate the starting individuals randomly. This way, one will statistically, given a large population, acquire individuals spread across the search space. This will prevent the whole population from focusing on a specific part of the search space, and might stop the algorithm from being stuck in a local minimum.

As described before, it is quite common to represent the genes of an individual as a binary string. The advantages of this are the compact representation and the fact that it allows simple techniques for crossover and mutation (described below). However, a binary string is not the only possible way of representing the genes. When it comes to neuroevolution, there are two main representations: direct and indirect (Stanley, D'Ambrosio & Gauci, 2009, p. 207). Direct encoding means the genotype and phenotype is approximately equal, meaning one change in the genes will result in one specific change in the phenotype. Every property of the phenotype is stored in its own gene. Indirect encoding means the genes are instead a recipe for how to generate the phenotype, meaning one change in the genes might result in big changes in the phenotype. Biological evolution uses indirect encoding. An advantage with the latter approach is the fact that it is more scalable, but it could also give an unpredictable bias, as discussed in Stanley and Miikkulainen (2002, p. 102).

Instead of a binary representation, the properties in the genes might also be described in a more atomic way. An example from neuroevolution is that the activation function of the neural network might be stored as a single unit, instead of being binary encoded. The same goes for real numbers, like the learning rate. In practice, everything will of course be converted to binary numbers on the hardware used, but from the algorithm's point of view, it will be viewed as an atomic value.

The procedure of selecting parents for the next generation should follow a few guidelines. The first is that a higher fitness value should correlate positively with a higher probability of being selected, as well-performing individuals often give well-performing children. The second is that there should be some variance in the parents selected, in order to ensure that the new individuals will be diverse and will explore a larger part of the search space. In other words, the best individual should not be the parent of *every* child. Based on these two guidelines, a lot of different ideas for selecting parents appear. Here, two common techniques are presented. The first is sampling parents with replacement with a probability proportional to the fitness value of the individual (Razali & Geraghty, 2011, pp. 3-4). More formally, the probability of selecting individual y with fitness f_y as parent, which is part of population X, is:

$$P(y) = \frac{f_y}{\sum_{x \in X} f_x}$$

The disadvantage with this approach is that if most of the individuals have an approximately equal fitness value, the distribution will be almost uniform. A method which solves this problem is to instead let the probability be proportional to the ranking of the individual compared to the rest of the population (Razali & Geraghty, 2011, pp. 4-5), like this:

$$P(y) = \frac{rank(y, X)}{\sum_{x \in X} rank(x, X)}$$

Here, *rank* is a function returning the ranking of an individual, so that the individual with the lowest fitness value will get a ranking value of 1, the second lowest will get 2, and so on. The best individual will get a ranking value equal to the population size.

More details concerning ways of selecting parents for genetic algorithms are given by Razali and Geraghty (2011).

Crossover, the process of combining the genes of the parents to generate new individuals, is usually done based on two individuals, just like in biological evolution. However, it is not limited to this. For example, a child could be created from crossover with three parents, like in Chandra and Yao (2006). In neuroevolution, it is also quite common to use only one parent (Hancock, 1992, p. 109). In that case, the crossover step is in practice skipped, and only mutation will make the child different from the parent. A biological version of this is found in bacteria. The rest of this discussion will focus on crossover with two parents.

If the genes are binary encoded, the methods one-point crossover and two-point crossover are popular. The first selects a random point in the string, and lets the child get the genes from one of the parents on one side of that point, and the genes from the other parent on the other side of it (Whitley, 1994, p. 68). This is illustrated in Figure 2.4. Two-point crossover is exactly the same, except that it selects two random points instead of just one. The child gets the genes from one of the parents on each side of the two points, and from the other parent between the points (Whitley, 1994, p. 71). This is illustrated in Figure 2.5.

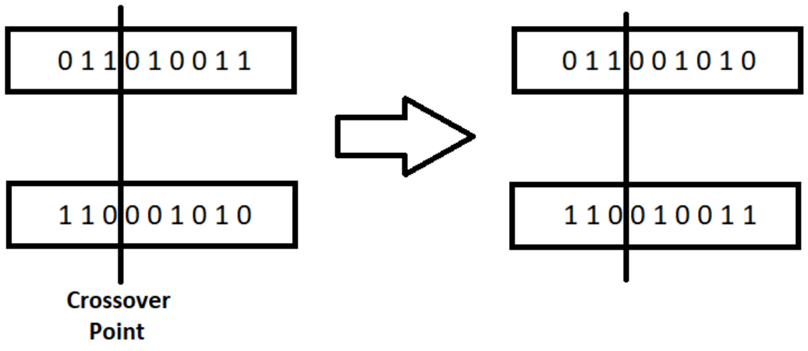


Figure 2.4: One-point crossover.

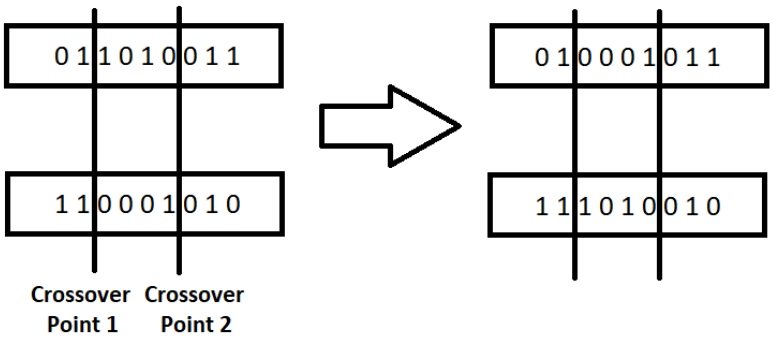


Figure 2.5: Two-point crossover.

These kinds of crossover make most sense if the order of the components in the genome has a specific meaning. An example of this is when a genetic algorithm is used to solve the famous Traveling Salesman Problem (TSP), where the first part of the genome represents the first place to be visited, the second part the next place, and so on. The length dimension in the genome represents the time dimension in the actual problem; i.e. the order in which the places should be visited. If one want a deeper explanation of TSP, see Dasgupta, Papadimitriou and Vazirani (2016, pp. 177-179), but it will not be relevant for the rest of this thesis. When it comes to the encoding of a neural network in a genome, there is no particular order which makes more sense than others, e.g. where the learning rate is placed in relation to the activation function, or network topology. Using crossover methods taking such spatial information into account will therefore give an unfortunate bias. These methods are therefore not very attractive for this kind of systems.

A more promising approach is the use of uniform crossover, which randomly samples which parent the child will get each part of the genome from. In the case of a binary encoding, each bit could be drawn randomly, and this is in fact a reason for not choosing a binary encoding in neuroevolution. If say the activation function is encoded binary, different parts of this encoding could be drawn from different parents, and one would therefore end up with an entirely new activation function. This means the choice of encoding gives a bias for which activation functions are close to each other in the gene representation, and therefore have a higher chance of replacing each other (e.g. if Sigmoid is encoded 000 and ReLU is encoded 001, they are only one bit apart, while another activation function encoded 111 will be three bits from Sigmoid). Instead, the whole activation function, or other hyperparameter, should be selected from one of the parents, in order to remove this bias. This might be regarded as an atomic uniform crossover method. For a more complete overview of different crossover methods, see Umbarkar and Sheth (2015).

Mutation makes the population explore a larger part of the search space, as it includes some randomness. This happens by letting the genes be changed randomly by a certain probability. The motivation behind this is that it might find new, promising genes through randomness. In the case of a binary encoding, a mutation could be as simple as flipping a random bit. If a more complex encoding is used, one could also use more complex mutation techniques. Generally, one could either mutate to a new random value (e.g. selecting a new, random activation function), or mutate based on the current value (e.g. doubling the learning rate).

2.2 Problem Domains

This subchapter is based on the equivalent part in the midterm report written for this master's thesis (Børstad, 2019, pp. 21-23). The main difference is that the EMNIST and fashion-MNIST datasets were not described there.

The problem domains which the systems have been tested in are presented here. As the research questions in Chapter 1 imply, this thesis focuses on investigating general methods that may be used in a wide range of domains, instead of solving one specific problem in one specific domain. Based on this, the domains chosen will only be introduced briefly. Also, this means the choice of problem domains are not particularly important, but the chosen domains should have some specific properties:

- Quite normal datasets should be chosen, where the data could be fed into the machine learning model almost directly. The thesis focuses on the evolution of neural nets, meaning it is distracting to choose datasets which require a lot of preprocessing etc.
- More than one problem domain should be chosen in order to find results which could be used to conclude about the methods in general, and not just for a specific dataset.
- To limit the work a bit, only classification problems have been considered.

Based on this, the following five problems have been chosen:

- Classification of Handwritten Digits
- Classification of Handwritten Characters and Digits
- Classification of Clothing Images
- Evaluation of Rook Endgames in Chess
- Localization of Proteins in Yeast

In the upcoming subchapters, each of the problems will be described briefly.

2.2.1 Classification of Handwritten Digits

Classification of handwritten digits might be the most famous of all machine learning problems, because of the MNIST dataset (Lecun, Cortes & Burges, n.d.). The set consists of a total of 70,000 data examples, each consisting of an image and a label. Each image contains 28x28 = 784 pixels, where every pixel is a grayscale with value between 0 (black) and 255 (white). The labels show which category each image represents, where the categories are the digits from 0 through 9. The dataset is available from Lecun et al. (n.d.). From here on, this problem will be referred to as simply MNIST.

For image classifiers, the use of convolutional neural nets (see Chapter 2.1.1) often improve the results. This means the selection of this problem is particularly well-suited for evaluating the convolutional system described in Chapter 4.3. However, decent results could also be achieved by using only normal, dense layers.

2.2.2 Classification of Handwritten Characters and Digits

Inspired by MNIST, the EMNIST dataset (Cohen, Afshar, Tapson & van Schaik, 2017) was created. EMNIST contains both handwritten letters and digits. The images have the same format as in MNIST. There are a few different configurations of the dataset available, differing on whether or not the categories are balanced, whether uppercase and lowercase letters are part of the same category or not, and so on. The balanced configuration is selected for this project. It contains 131,600 images from 47 different categories. Uppercase and lowercase letters are separated for some letters, but not for others. This is because it is close to impossible to separate e.g. an uppercase from a lowercase ”c”, without a context of other letters around it. See Cohen et al. (2017) for more details about this dataset. The dataset is available from NIST: National Institute of Standards and Technology (2017). From here on, this problem will be referred to as EMNIST.

2.2.3 Classification of Clothing Images

Fashion-MNIST (Xiao, Rasul & Vollgraf, 2017) is also inspired by MNIST as the images have the same format, but contains images of pieces of clothing from Zalando. It has 70,000 data examples belonging to ten different categories, each representing a different type of clothing: t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, and ankle boot. More details can be found in Xiao et al. (2017), and the dataset is available for download from Zalando Research (n.d.). From here on, this problem will be referred to as fashion-MNIST.

2.2.4 Evaluation of Rook Endgames in Chess

For the rules of chess, see Chesscom (2020). This dataset concerns chess positions where only the kings and one white rook is left on the board. It is always black to move first in the examples in the set. These kinds of endgames are always won for white (that means white is able to checkmate black with best play), unless the black king immediately captures the white rook, or the position is a stalemate (see Chesscom (2019, ”How to Draw a Chess Game”)). In these two cases, the game ends in a draw. For all other cases, white can always checkmate in at most 16 moves with best play, where the exact number of moves depends on the positions of the three pieces. Figure 2.6 illustrates such an endgame, where white can checkmate black in one move.

Each example in the dataset consists of six features, representing on what rank and file each of the three pieces are placed. The horizontal ranks are numbered from 1 through 8, and the vertical files are numbered from *a* through *h*. The value to be predicted is whether the position is a draw, or how many moves there are until checkmate with best play, between 0 and 16, where 0 means the position is already a checkmate. That means there are a total of 18 categories to choose between.

The dataset contains 28,056 data examples, which are relatively evenly distributed among the 18 categories. For more details about the dataset, see University of California, Irvine (n.d.-a). The dataset is also available for download from that reference. From here on, this problem will be referred to as simply chess, or the chess dataset.

2.2.5 Localization of Proteins in Yeast

This dataset, called the yeast dataset, is also quite well-known. It is a challenging dataset, meaning it is hard to get a very high accuracy. The set consists of 1,484 examples, each with eight features, and belonging to one out of ten categories. All features are real numbered values, which can be fed directly into the model. For a detailed description of the

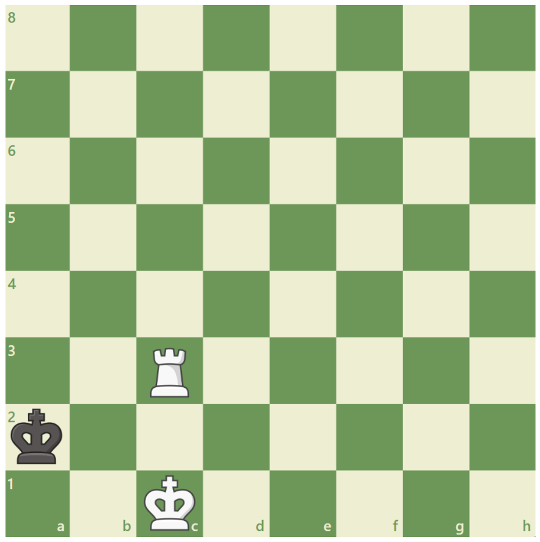


Figure 2.6: An example of a position from the chess dataset. The black king must move downward into the corner, and white can checkmate with the rook.

dataset, including what each feature and category represents, see Horton & Nakai (1996). In summary, the task concerns localizing proteins in yeast based on data from various analyzes and measurements. The dataset is available from University of California, Irvine (n.d.-b). From here on, this problem will be referred to as simply yeast, or the yeast dataset.

2.3 Summary

This chapter has presented relevant background theory for understanding the rest of this thesis. The two main parts of any neuroevolution system, neural networks and genetic algorithms, have been thoroughly introduced. Thereafter, the five datasets which have been used to test the systems were presented. This includes the three image datasets MNIST, EMNIST and fashion-MNIST, as well as a chess rook endgame dataset and a yeast dataset.

Chapter 3

Related Work

In this chapter, the most relevant previous work will be presented. First, in Chapter 3.1, the process of finding related literature, a structured literature review (SLR), will be explained. Then, the most relevant literature is presented in Chapter 3.2. Chapter 3.3 summarizes the chapter.

This chapter is based on the midterm report written for this master's thesis (Børstad, 2019, pp. 24-33), as the main part of the literature review was completed early in the work with this thesis.

3.1 Structured Literature Review

Here, the structured literature review (SLR) which has been carried out will be described. Neuroevolution is a field of research which has been studied for a long time, meaning a lot of research papers are available. Even though deep learning has first become popular over the last few years, general work on combining genetic algorithms with less complex neural nets have been conducted all the way back to the 1980s. See Montana and Davis (1989), Miller, Todd and Hegde (1989), and the review paper of Yao (1999) for examples of such. These early papers could introduce useful concepts for today's systems within deep learning, even though they were originally used on a much smaller scale. In summary, a large quantity of relevant papers are available, meaning it is not realistic to read them all. Therefore, it is necessary to take a structural approach when it comes to selecting what should be read in order to avoid important papers from being excluded.

The reading of relevant literature primarily has two purposes: to gather information and inspiration which could be useful for the thesis, and to ensure nobody else has done anything too similar before. For the first of these purposes, the more read, the more knowledge was available before the practical part of the thesis should be executed. For the latter, one can never be completely sure nobody has done something similar without reading all available papers, but the SLR reduces the risk of missing a paper describing something very similar.

The starting point of the literature review was two papers sent from the advisor, namely Stanley, Clune, Lehman and Miikkulainen (2019) and Miikkulainen et al. (2019). These were read thoroughly, and then references in them which seemed interesting and relevant were followed for further readings. In some of the new papers found this way, new references were found interesting and relevant, and from this a tree spanned from the two original papers by following references from each paper to new papers. A lot of papers were found this way. Also, dialogue with the advisor at later times during the thesis has resulted in recommendations for more papers which could be worth reading.

Complementary to this approach, search engines were used to find relevant papers. The search engine used the most was Google Scholar. Both general terms related to neuroevolution, and more specific terms discovered by reading other papers, have been queried. Table 3.1 shows the queries which have been used in Google Scholar, and the number of papers which showed up for each of them.

Query	**Number of Papers**
AutoML	2,330
DeepNEAT	65
Evolutionary Algorithm Neural Net	278,000
Evolving Artificial Neural Nets	323,000
Evolving Neural Networks	384,000
Genetic Algorithm Neural Net	440,000
HyperNEAT	1230
Incremental Complexification	5,770
Neural Architecture Search	2,020,000
Neuroevolution	7,580
Neuroevolution Aging	486
Neuroevolution Author:Miikkulainen	217
Neuroevolution Author:Stanley	195
Neuroevolution Chess	615
Neuroevolution CNN	521
Neuroevolution Deep Neural Net	3,370
Neuroevolution Ensembles	1,410
Neuroevolution Evolve Learning Rate	4,690
Neuroevolution Hyperparameters	748
Neuroevolution MNIST	277
Neuroevolution Novelty Search	1,220
Neuroevolution Of Augmenting Topologies	3,070
Neuroevolution Permutation Problem	676
Neuroevolution Reinforcement Learning	3,750
Neuroevolution Review Article	2,930
Neuroevolution Yeast Dataset	37

Table 3.1: Queries in Google Scholar.

As one can see, far more papers were found than what could possibly be read. First, the number was reduced by the following two criteria:

- Only the 10 first (and hopefully most relevant) papers for each query were considered.
- The title of the paper was used to estimate its relevance.

For the papers which still seemed relevant, the summary was read. This goes for both the papers found by using search engines, and by the other methods mentioned above. Based on the summary, each paper was classified into one of the following three categories:

- Paper will be read
- Paper will be skimmed
- Paper will not be read

In the next subchapter, the most relevant of the read papers found through the SLR will be presented.

3.2 Summary of Related Work

Many of the papers introduced here describe components which have been used as inspiration and starting points for the systems implemented during this thesis. Chapter 3.2.1 presents two papers which are quite different from each other, but which both give an introduction to the field of research. Then, in Chapter 3.2.2, NEAT, one of the most recognized works within neuroevolution, and some of its extensions, will be discussed. Systems focusing on diversity in the population will be presented in Chapter 3.2.3, and systems evolving other hyperparameters than the topology are introduced in Chapter 3.2.4. In Chapter 3.2.5, three papers which contain interesting ideas, but which do not really fit into any of the other subchapters, are discussed.

3.2.1 Introducing Papers

The two papers presented here, give an introduction to neuroevolution in different ways. The first does so by describing the broader field called AutoML, which neuroevolution is a part of, and the second because it is one of the very first papers written about neuroevolution.

He et al. (2019) does not solely focus on the evolution of neural nets, but gives an overview of the field of AutoML. AutoML is the idea of automating every aspect of machine learning, including data preprocessing, feature engineering, selection of model and hyperparameters, and model evaluation. Neuroevolution is a subfield of AutoML, and more specifically underneath the part of selecting hyperparameters. The paper reviews the most common methods of using genetic algorithms for this purpose. Additionally, alternatives to genetic algorithms to determine hyperparameters of neural nets are discussed, that is; grid search, reinforcement learning, Bayesian optimization and using gradient descent directly. All of these methods are part of the subfield of AutoML called NAS (Neural Architecture Search). The most promising systems from each of the approaches mentioned are compared based on various image classification problems. The best results are achieved with the system "RL NAS", which uses reinforcement learning. Generally, reinforcement learning and genetic algorithms achieve very promising results, and considerably better ones than the other methods. From this, the claim that genetic algorithms are well-suited for tuning hyperparameters are strengthened, but it also shows this is not the only approach which can give good results.

Miller et al. (1989) was one of the very first papers published regarding neuroevolution. Here, the topologies of the networks are evolved, before the weights are tuned with backpropagation. A matrix is used to represent what connection exists between each pair of neurons. The paper suggests several types of connections, but in the tested system, only a binary representation is used: either two neurons are connected, or they are not. The relation between genes and phenotype is therefore very directly encoded. The system is only tested for small neural nets, but show good results on both the XOR problem and the four quadrant problem, among others.

3.2.2 NEAT

In this subchapter, what is probably the most wide-known system within neuroevolution, NEAT, is introduced. Thereafter, a couple of extensions of NEAT are presented.

In Stanley and Miikkulainen (2002), the innovative system NeuroEvolution of Augmenting Topologies (NEAT) is presented. NEAT evolves both the network weights and the topology by using a direct encoding, and the paper presents results better than the state-of-the-art at that time for a pole balancing problem, a reinforcement learning problem where the goal is to balance a pole in an environment with gravity by adjusting some parameters. The system applies three very interesting ideas in order to perform well:

1. The use of historical markers
2. Guarantying diversity with species
3. Incremental complexification from a minimal structure

NEAT uses genomes of variable length. To be able to execute crossover in a sensible way due to this, historical markers are used (idea 1). Every time a genome is extended through mutation, a number is saved with the new gene. This number represents when the gene was added. By adjusting the genes in a way that makes the components with the same numbers, and therefore are of the same type, be switched during crossover, the genes will not be damaged, like they could have been otherwise. If there exists no gene with a specific number in the genome of one individual, there will be an empty space there instead. Figure 3.1 illustrates this.

To restrict the population from converging toward one specific type of network too early, the population is divided into species (idea 2), which primarily compete internally, as opposed to competing with every other individual in the population. By letting the individuals within a species share their fitness value, one does not risk one species becoming too large. The more individuals within a species, the more must share the fitness value, meaning it will decrease for each individual. At some point the fitness value will become low enough for individuals from other species to score higher. The historical markers are used to determine what individuals belong to the same species. More similar historical markers

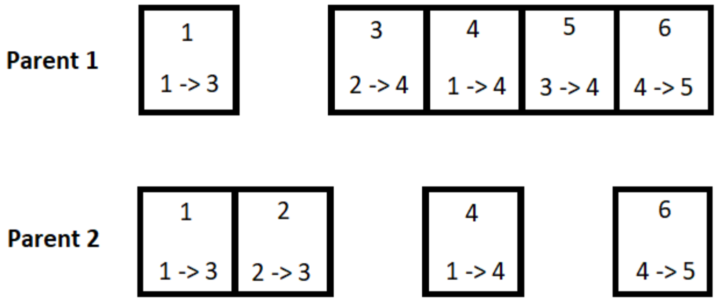


Figure 3.1: Usage of historical markers in NEAT. The upper number is the historical marker, making sure the similar genes are placed underneath each other during crossover. The lower value represents a connection between two neurons.

mean the individuals are more similar, and therefore could be of the same species.

Incremental complexification from a minimal structure (idea 3) is to let the initial population only consist of individuals with small, simple topologies. In NEAT's case, every net begins without any hidden neurons. Thereafter, such hidden neurons will be added through mutations. The advantage of this approach is that one will not miss out on small, simple networks which might work well. Larger nets can be more likely to overfit to the training data, and also usually require a longer time to train. For these reasons, it seems like a good idea to try small networks first. Such algorithms are called constructive. The opposite are destructive algorithms, which begin with a more complex topology, and gradually simplifies it (Yao, 1999, p. 1428). Research question 4 is inspired by this idea of incremental complexification.

Stanley et al. (2009) presents HyperNEAT, an extension of NEAT. The difference between the two is that HyperNEAT uses an indirect encoding to represent the neural nets, instead of the direct encoding used by the original NEAT. The word "hyper" comes from the fact that a hypercube is used to determine the weights of the nets. This cube works as a function taking the coordinates of the two neurons with a connection as arguments, and returning the weight for this connection. This is called a Compositional Pattern Producing Network (CPPN). One of the advantages with this method is that it can take advantage of the location information from the features (e.g. how the pixels of an image are located relatively to each other). Such information is not available for a normal, dense neural net. The encoding turns from direct to indirect by evolving the function (i.e. hypercube) which determines the weights, instead of evolving the weights directly. The system is tested on a visualization problem and a food gathering problem, and performs well on both. The paper states two reasons why indirect encodings are better than direct ones: they are more scalable, and have a higher potential for finding patterns in the connections. The reason for the first is that every weight does not need its own part of the genes, meaning the genome will be smaller, and more weights can be evolved without increasing the search space at the same rate. The reason for the second is that patterns can be found and reused as the same parts of the genome are used to determine more than one weight.

Many more extensions of NEAT have been published. Examples of such are real time NEAT (rtNEAT) (Stanley, Bryant & Miikkulainen, 2005) and Adaptive HyperNEAT (Risi & Stanley, 2010). Here, only two more versions of NEAT will be presented, both of which are much newer. These are named DeepNEAT and CoDeepNEAT, and are both introduced by Miikkulainen et al. (2019).

DeepNEAT is probably the related work most similar to this thesis. The method is based on the same ideas as NEAT, with the exception that every part of the genome no longer represents a single connection between two neurons, but instead a whole layer in the net. This makes the method much more scalable than NEAT, and better suited for evolving deep neural nets with many neurons and layers. For every layer, several hyperparameters are evolved, including the layer type (convolutional layer, dense layer or recurrent layer), the number of neurons, the activation function, and so on. The layers do not necessarily need to be ordered in one long series either, which is the most common, but could instead hold a more complex structure. Furthermore, global hyperparameters which are not related to a single layer are also evolved, like a learning rate and momentum. The networks are trained and evaluated with the backpropagation algorithm, unlike NEAT where the weights are determined evolutionarily. Unfortunately, DeepNEAT is only presented as a step toward CoDeepNEAT in Miikkulainen et al. (2019), meaning no more details about how the system works, or results from experiments with DeepNEAT, are shown.

CoDeepNEAT (Coevolution DeepNEAT) brings the ideas from DeepNEAT one step further, and uses two different types of populations: one with modules, and one with blueprints. Every module represents a small neural network, while the blueprints connect several of these modules into larger nets. In other words, it is the blueprints which create the final neural nets, but they depend on favorable modules to use in order to perform well. The advantage of such a design is that it has a high degree of freedom and creates a large search space which has the potential to discover brand new modules and network topologies which no one has tried before. The system is tested on a task where it should automatically caption images, and achieves promising results.

3.2.3 Diversity

Diversity in the population could be important in order to find satisfying solutions, and is a big part of this thesis as it is covered by research question 3. Here, three papers about the topic are introduced.

Novelty search, a method which completely ignores the actual objective and solely focuses on increasing the diversity of the population, was first introduced in Lehman and Stanley (2008), and is further discussed in Lehman and Stanley (2011). The main arguments for using novelty search are that the individuals often become very similar if only the final objective is used to evaluate fitness, and that the final objective not necessarily is a good heuristic along the way toward that objective. Lehman and Stanley (2011, p. 45) also claims that the diversity of the population should be evaluated based on the phenotype of the individuals instead of the genes directly, in order to work well.

Furthermore, Lehman and Stanley (2011) suggests open-ended search, where a population is evolved without any final objective at all, with the hope that such unrestricted evolution will lead to the discovery of individuals with new, interesting capabilities. This is very much inspired by biological evolution, where complex creatures have emerged, even though no specific objective has been set to achieve such. Open-ended search is closely linked to novelty search, as they are both about ignoring the objective. By using the introduced methods, the authors can refer to impressive results on a maze task, among others. This includes results better than those achieved by evolution lead by a normal fitness function directed toward the objective.

In Such et al. (2018), genetic algorithms are used instead of backpropagation to adjust the weights of neural nets, which in turn are used for reinforcement learning. One version uses the objective as fitness function directly, while another version uses pure novelty search. The first version is tested on a variety of Atari games, a maze task, and a humanoid locomotion task. The second version, with novelty search, is only tested on the maze task. The neural nets produced by the first method perform well on the two first tasks, but disappoint on the last one. The novelty search version does even better than the first one on the maze task.

Chandra and Yao (2006) suggests DIVACE (DIVerse/ACcuratE), a system which takes both the accuracy of the neural networks and their diversity into account. This is different from traditional novelty search as it also includes the actual objective, i.e. the accuracy. It is a form of multi-objective learning as accuracy and diversity can be viewed upon as two different objectives, and the system searches a Pareto front (explained in Ngatchou, Zarei and El-Sharkawi (2006)) to find the best networks. The system is tested on two different datasets: the Australian Credit Card Assessment Dataset and the Diabetes Dataset, and shows promising results on both compared to similar previous methods.

The paper also uses simple forms of ensembles. In general, ensemble methods are the idea of combining more than one model, because they together are more accurate than any model alone. The main reason for this is that different models often do not make the same mistakes. By using several models, the mistakes are therefore evened out, meaning the plurality of models predict correctly more often. For this to be true, two criteria must be fulfilled: the models must be accurate and diverse. Accurate means they are better than random guessing. Diverse means there is not a (high) correlation between the mistakes the models make. As can be seen, the two requirements for a good ensemble are exactly the same two things as this system focuses on when evolving the nets. The use of ensembles with neuroevolution is quite natural, as the method already evolves several neural networks, and hence have many models available without any extra work required.

Pairwise Failure Crediting (PFC) is introduced in the same paper as a new measurement of diversity. Every neural network gets a binary string representing what validation examples were predicted correctly and wrongly. Every pair of individuals in the population are compared by their binary strings, and get a score based on how different the two individuals of the pair are. This score is summed for every pair a specific individual is part of, and the individuals can from this be ranked by their diversity. Mathematically, it looks like this:

$$div_{i,j} = \frac{diffPreds(i,j)}{wrongPreds(i) + wrongPreds(j)}$$

$$pfc_i = \frac{\sum_{j=0}^{N} div_{i,j}}{N-1}$$

Here, $div_{i,j}$ is the diversity score of the pair of individuals i and j, $diffPreds$ returns the number of different predictions made by two individuals, $wrongPreds$ returns the number of wrong predictions made by an individual, and N is the population size. $div_{i,i}$ will always be 0 (i.e. every individual is identical to itself). The PFC value will always be in the interval [0, 1], where a higher value indicates a more unique individual. PFC can be regarded as a less extreme version of shared fitness values, which were introduced earlier with species in NEAT.

The paper also discusses Lamarckian evolution, which is an alternative to Darwinism. In the traditional, biological Darwinism, only the genome of the parent is used as basis for how the child turns out. Anything an individual learns during its life will not be transmitted to the child. In Lamarckian evolution, this is possible; not only genes, but also knowledge, can be transmitted to the children. This means it contradicts biological evolution, but could still work well for genetic algorithms. When it comes to neural networks, Lamarckian evolution can be used by letting the weights of a child be initialized with the values of the trained weights of the parent(s).

Another interesting idea used in the paper, is to use three parents instead of two for crossover. Additionally, not all three parents have the same importance for how the child turns out. The child gets most of its genes from parent number 1. It is uncertain whether or not this idea works well generally, but the fact that genetic algorithms are not limited to a specific number of parents, and that not every parent needs to have the same importance for how the child turns out, can create inspiration for new ideas.

3.2.4 Evolving Various Hyperparameters

Most of the work within neuroevolution has focused on evolving network weights and/or topologies. In this subchapter, papers where other hyperparameters, such as the learning rate and learning rules, have been evolved, will be addressed. This is relevant for the thesis as research question 1 focuses on evolving several different hyperparameters.

In Kim, Jung, Kim and Park (1996), the learning rate is evolved, and it can hold different values for different layers and training steps. The values are random in the initial population, and then children are created through mutation by adding a value drawn from a normal distribution. Crossover is omitted. The normal distribution which the mutations are drawn from, have mean 0 and standard deviation determined by a perturbation factor. This is defined so that individuals with low fitness values get a larger standard deviation, as they most likely need to be mutated more in order to perform well. The paper presents good results on a modulo problem.

Patel (1996) tries to evolve different types of hyperparameters. In the version described in the paper, this includes the learning rate and the number of hidden neurons (only one hidden layer is used). However, it is mentioned that the system easily can be expanded to evolve more hyperparameters, like momentum and the number of hidden layers. The values to be evolved are converted to a binary representation, and are initialized with random values within certain boundaries. Both crossover with two parents and mutation are used. The paper does not specify exactly how these methods are applied, but it can be assumed they are standard methods for binary representations (see Chapter 2.1.2). The method is used to make predictions for the stock value of the company British Telecom. As stock data is created over time, the paper also spends some time discussing time series predictions. The results are accurate for predictions in the near future.

Chalmers (1991) discusses the evolution of learning rules, i.e. how the weights in the neural network should be updated. The method rediscovers the known delta rule for a set of linearly separable problems, but does not find any new, revolutionary learning rules. Backpropagation is such a well-established method making it unlikely that a better rule will be discovered, but the paper discusses several interesting concepts which should be taken into account for other systems within neuroevolution, including those in the next paragraphs.

The search space is limited by only looking at specific parameters as input for the learning rule, only considering linear combinations of these and the product of each pair of these, and only considering a finite number of possible values for the coefficients which are evolved and multiplied by the features. Such limitations have the advantage that one can find good solutions faster, if they are still part of the limited search space. The disadvantage is of course that new, revolutionary learning rules might be located outside of this limited search space.

A crossover rate is used, in addition to a mutation rate. This determines the rate of children generated by crossover with two parents. The rest of the children are created by copying the genes of a single parent, meaning mutation will make the only difference between parent and child.

Elitism, which is also used in the paper, is to let the best individual from the previous generation always survive until the next generation. This means one will never forget a solution from earlier generations which turns out to be the best one in total.

3.2.5 Other Ideas

The three papers discussed here do not fit well under any of the other subchapters, but nonetheless introduce interesting ideas for neuroevolution.

Real et al. (2017) uses evolution to find architectures for image classifiers, but many of the techniques introduced might work well for other types of neural nets as well. Crossover is not used, meaning all change is caused by mutations. There exists a list of possible mutations, and for every child one of these is drawn randomly. Most of the mutations concern the network topology, but additionally, one of them is weight inheritance. That means the child will begin with its parent's weights, and further adjust these through backpropagation. This is only possible because no other mutations happen at the same time, meaning the topology will remain the same. If not, there would not have been any simple way of determining how the weights should be inherited. Weight inheritance can be considered a type of Lamarckian evolution (see Chapter 3.2.3). In addition to determining the topology through evolution, the proposed system also evolves the learning rate. The paper presents results comparable to state-of-the-art within image classifiers on the well-known datasets CIFAR-10 and CIFAR-100.

Instead of dividing the algorithm into generations, new individuals are produced continually. This has the advantage that there will not be any waiting time at the end of each generation when most of the individuals are done training, while some are not. All available computational resources can instead start generating new children. Instead of sorting all individuals by their fitness values, only two random individuals are compared each time. The worst will be removed from the population, while the best will be the parent of a new individual. This makes the method very well-suited for running distributed; the processes can run completely independently of each other, and do not even need to know every individual's fitness value at the same time. This method is an example of tournament selection and steady-state evolution.

Gomez and Miikkulainen (1997) describes incremental neuroevolution. This is to first evolve nets which solve simpler versions of the actual problem, and then gradually increase the problem difficulty. The motivation for this is the fact that it might be hard to find good solutions by looking for the final objective already at the start of the algorithm's run. A comparison from biological evolution is the following: In the beginning, only bacteria and simple creatures existed, which only had to do very simple tasks in order to survive. Then, gradually, more complex creatures were evolved, all the way up to humans. For evolution, humans are of course not the end goal, but there are still similarities here. If biological evolution tried to evolve humans right from the start, without going through the simpler steps first, it would probably have failed. Incremental neuroevolution is quite similar to, and might easily be confused with, incremental complexification which NEAT used (see Chapter 3.2.2), and which research question 4 concerns. The difference is that incremental neuroevolution increases the problem difficulty gradually, while incremental complexification increases the complexity of the neural networks gradually.

The system is tested on a hunter problem, where a simulated robot tries to catch prey by navigating an environment, as well as a pole balancing problem. On both problems the results shown are better than those achieved using direct evolution. A challenge with incremental neuroevolution is that not every problem easily can be reduced to simpler subproblems.

Real, Aggarwal, Huang and Le (2019) mainly concerns image classifiers, but introduces a technique that might be interesting in general for genetic algorithms. This is called aging, and is to associate every individual with an age, and favor young individuals. This means younger individuals will have a higher chance of being selected as parents for the next generation, and that older individuals will die and be removed from the population. This forces a continual evolution where the individuals must be able to pass their well-performing properties through their genes over generations. Aging can also help making the population more diverse, as it is no longer possible that some well-performing, but very similar individuals will stay in the population for a very long time.

3.3 Summary

This chapter has covered related papers for the work in this master's thesis. First, the Structured Literature Review (SLR) was explained, and then the most relevant papers found were presented. For a more detailed overview of what has been done within neuroevolution, three review articles are recommended. Yao (1999) thoroughly covers the field up until the year it was written. Floreano, Dürr and Mattiussi (2008) describes what was considered most exciting at that time. Stanley et al. (2019) covers more modern breakthroughs and ideas.

From the literature, many interesting ideas have been found, some of which have been used as inspiration for the systems implemented in this thesis. The details behind the thesis' systems are covered in the next chapter. However, already at this point it can be said that these systems differ from what has been done previously, as it does not seem like anyone has attempted to evolve all the different hyperparameters which are normally set by the user, directly. The previous work which seems most similar to the systems of this thesis, is DeepNEAT (Miikkulainen et al., 2019) (see Chapter 3.2.2).

Chapter 4

Methodology

This chapter will describe the systems implemented in order to run tests to gather results which in turn have been used to answer the research questions. Chapter 4.1 describes the main system used for most of the tests. Thereafter, the two other versions of the system created, one using subpopulations and one creating convolutional neural networks, are presented in Chapter 4.2 and 4.3, respectively. Chapter 4.4 summarizes the chapter.

4.1 Main System

The main system evolves deep neural networks consisting of dense layers, for classification tasks. The evolved hyperparameters are those usually set manually by the user before training a neural network. This search space is thoroughly described in Chapter 4.1.1. The system utilizes many of the techniques commonly used by genetic algorithms, including population initialization, parent selection, crossover, mutation, individual evaluation and individual removal. These main components of the system are described in Chapter 4.1.2. Thereafter, in Chapter 4.1.3, other aspects of the system are discussed. Finally, chapter 4.1.4 discusses the fact that a neuroevolution system like this one introduces as many new parameters for the genetic algorithm as it removes from the neural network, and why this is not a big problem. For a more technical description of how the system can be used, see appendix 1. The source code of the main system is found in appendix 2.

4.1.1 Search Space

Eight different hyperparameters are evolved by the genetic algorithm to describe the neural networks which the population consists of. These are:

- Topology
- Activation function
- Cost function
- Optimizer
- Learning rate

- Learning rate decay
- Dropout probability
- Batch size

The different hyperparameters are encoded separately, meaning they are created and mutated independently of each other. The encoding of every hyperparameter will now be discussed, one by one. For details about how the different hyperparameters are set when initializing the population and mutating an individual, see Chapter 4.1.2.

Topology

The topology, or network architecture, describes the number of neurons in each layer. Only dense layers are evolved in this main system. The encoding of the topology can be considered a list of integers, describing the number of neurons in each hidden layer. The length of the list represents the number of hidden layers. The number of neurons in the input and output layers are not evolved, as these are determined by the problem attempted to be solved, and cannot be changed freely by the algorithm.

The use of skip connections was tested on an early version of the system, but resulted in slightly worse results, and is therefore not used in the final system. Skip connections lets layers be connected to other layers than the ones right before and after itself (if one thinks of the layers as a one-dimensional chain), hence letting parts of the signal flow "skip" layers. Further testing of this could be a subject for future work.

Activation Function

Eight different activation functions can be evolved by the system. These are:

- **Sigmoid** $\frac{1}{1+e^{-x}}$
- **Tanh** $\frac{e^x - e^{-x}}{e^x + e^{-x}}$
- **Softsign** $\frac{x}{1+|x|}$
- **ReLU** $max(x,\ 0)$
- **Leaky ReLU** $max(x,\ \alpha x),\ \alpha = 0.2$
- **Swish** $x \cdot Sigmoid(x)$
- **Softplus** $ln(1+e^x)$
- **SeLU** $\lambda \begin{cases} \alpha(e^x - 1) \text{ for } x < 0 \\ x \text{ for } x \geq 0 \end{cases},\ \alpha \approx 1.67,\ \lambda \approx 1.05$

The choice of exactly these functions are partly because they are among the most popular activation functions, and partly because early testing suggested this set worked well. The three first functions are quite similar, as they all follow S-shapes, and so are the last five, which all have a rectified shape. The activation functions are encoded atomically, meaning that from the system's point of view, none of the activation functions are more similar to one than to another. This is done to avoid adding an unpredictable bias as discussed in Chapter 2.1.2.

Cost function

The cost function is encoded in the same way as the activation function, and the possible functions are selected for the same reasons; they are common and performed well during early testing of the system. Three different cost functions are considered by the system:

- **Mean Square Error (MSE)** $\frac{1}{n}\sum_{i=1}^{n}(c_i - p_i)^2$
- **Cross Entropy** $-\sum_{i=1}^{n} c_i \cdot ln(p_i)$
- **Huber** $\sum_{i=1}^{n} \begin{cases} \frac{1}{2}(c_i - p_i)^2 \text{ for } |c_i - p_i| \leq \delta \\ \delta|c_i - p_i| - \frac{1}{2}\delta^2 \text{ for } |c_i - p_i| > \delta \end{cases}$, $\delta = 1$

In the formulas, n represents the number of categories, c the correct values, and p the predicted values.

Optimizer

Also the optimizer is encoded atomically, and the possible techniques are selected for the same reason as the activation functions and cost functions. Five different optimizers can be evolved:

- **Gradient Descent**
- **Adam** with $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$
- **Adagrad** with $initialAccumulator = 0.1$
- **RMSProp** with $decay = 0.9$, $momentum = 0$ and $\epsilon = 10^{-10}$
- **FTRL** with $learningRatePower = -0.5$, $initialAccumulator = 0.1$, $l1RegularizationStrength = 0$, $l2RegularizationStrength = 0$ and $l2ShrinkageRegularizationStrength = 0$

The parameters of the optimizers are set to common default values. These parameters could of course also have been evolved, alongside the other hyperparameters of the neural nets, but this has not been done in order to reduce the size of the search space. See Ruder (2017, pp. 4-9) for details about these optimizers (except for FTRL), and Reddy (2019, "Follow the Leader (FTL)") for details about FTRL.

Learning rate

The learning rate is encoded as a continuous value. The only restriction is that it must be larger than 0.

Learning rate decay

The learning rate decay hyperparameter is used as part of a learning rate schedule to decrease the learning rate over time (see chapter 2.1.1). A time-based learning rate schedule is selected, which updates the learning rate by the following formula:

$$\eta_{n+1} = \frac{\eta_n}{1 + dn}$$

η_{n+1} is the learning rate for training step $n + 1$, meaning η_0 is the original learning rate set by that hyperparameter. d is the evolved learning rate decay hyperparameter.

Just as with the learning rate, the learning rate decay is encoded as a continuous value, and must be larger than zero. In practice, the learning rate decay will be $<< 1$. If not, the learning rate will approach 0 too fast, and most of the training process will be negligible. This is however not enforced by the encoding, but is instead handled by the way it is being initialized and mutated (see Chapter 4.1.2).

Dropout probability

Also the dropout probability is a continuous value. It must be within the interval [0, 1] as it is a probability. Its value determines the probability that any neuron in a hidden layer will be dropped for the upcoming training step. The same dropout probability is used for every hidden neuron and for every training step.

Batch size

Mini-batch gradient descent is used to train the neural networks (see Chapter 2.1.1), and hence, a batch size must be determined. The batch size must ultimately be a positive integer, but is encoded as a continuous value in the genome. This is because it allows a simpler and smoother mutation of the value. The encoded value is rounded to the closest integer before the neural network is trained. The positive, continuous value could be rounded to the illegal batch size of 0, or be larger than the number of training examples. If so, it is simply set to 1 or the maximum legal value, respectively, during training. Mathematically, that is:

$b = median(1, round(\hat{b}), t)$

Here, b is the integer batch size finally used, $\hat{b}$ is the continuous batch size evolved by the system, and t the number of training examples (i.e. the maximum batch size).

The training is divided into epochs, meaning every training example will be used once before any example is used for a second time (see Chapter 2.1.1).

Other Hyperparameters

Finally, a few more hyperparameters are treated by the system, without being explicitly part of the genes of the individuals.

Instead of a constant number of training steps, a constant training time is used by the system, determining the number of seconds each neural network will be trained. This means the number of training steps is determined implicitly by the values of the other hyperparameters. For example, larger topologies have more weights, meaning a longer time is needed to execute each training step. Hence, larger topologies will reduced the number of training steps. If a constant number of steps would have been used instead, large topologies would severely impact the system's running time, as each training step would take a lot of time. This also makes the competition between the networks more fair, as they cannot simply grow extremely large to achieve good results, as it then will not be time to adjust the weights properly.

In addition to topology, the batch size also affects the number of training steps considerably. A higher batch size means each step will consume more time. The other hyperparameters might also have a slight impact on the number of steps, but to a much smaller degree. E.g. the time used to calculate the different activation functions are approximately equal, or at least very small compared to the other parts of the training. It is the CPU time used which determines when the training is terminated, and not the wall time. This means the time available for training should be fair between the networks in a run, and even networks in different runs, independently of the current capacity of the hardware.

It would also have been problematic to evolve the number of training steps as an independent part of the genome, as a larger number of steps often give better results, but at the cost of a longer running time, naturally. Without penalizing networks training longer, one could risk the networks training practically infinitely. A penalty could have been possible, but it is hard to find a balanced penalty so that the optimal strategy will not either be to not train at all, or to train for a very long time anyway. Stanley and Miikkulainen (2002, pp. 105-106) also discusses this briefly. Therefore, having the number of training steps determined by a training time which is the same for all nets, seems like the best approach.

The distribution from which the initial weight values are drawn was attempted evolved in a previous version of the system. However, it has been found that using a simple, parameter-free approach like Glorot works just as well (Glorot & Bengio, 2010). With Glorot, the initial weights are drawn from a normal distribution with mean μ and standard deviation σ set as follows for layer l:

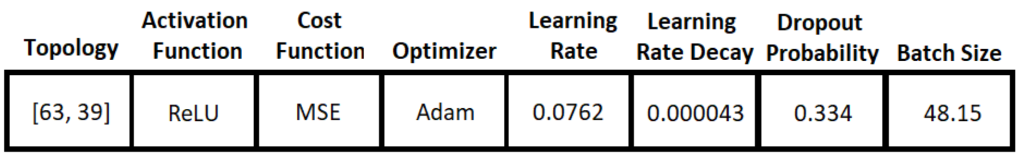
$$\begin{aligned} \mu_l &= 0 \\ \sigma_l &= \sqrt{\frac{2}{n_{l-1} + n_l}} \end{aligned}$$

Here, n_l is the number of neurons in layer l. The main advantage of Glorot is that it reduces the search space by removing an additional hyperparameter.

Batch normalization is also used by the system. However, as there are rarely any drawbacks with this method, it is used for every neural net, and therefore does not add any hyperparameters to the search space. Batch normalization is used for every hidden layer.

Every hyperparameter, except for the number of neurons in each layer, has the same value for every hidden layer and neuron. An early version of the system attempted to let some of the hyperparameters evolve independently for different layers (e.g. one could have different learning rates for each layer), like some of the related work have done (e.g. Miikkulainen et al. (2019) (Chapter 3.2.2) and Kim et al. (1996) (Chapter 3.2.4)). However, the initial results were rather poor. This is most likely because of the severe increase in the size of the search space.

In summary, no part of the genome is binary encoded. Instead, it consists of a list of integers (the topology), three atomic values (the activation function, cost function and optimizer), and four continuous values (the learning rate, learning rate decay, dropout probability and batch size). The main reason for this is that crossover and mutation operators working on individual bits probably would have performed poorly in this context (discussed in Chapter 2.1.2). The chosen encoding leans more toward a direct one than an indirect one, as it is the hyperparameter values themselves that are evolved, instead of a function etc. which later will determine the hyperparameter values. However, the abstraction level is higher than for the traditional directly encoded neuroevolution systems (e.g. NEAT), where individual neuron connections are encoded explicitly in the genome. Figure 4.1 shows a graphical representation of the gene encoding. The order of the different hyperparameters does not matter, as no spatial dependent crossover or mutation operators are used. The search space is infinite because the topology can keep growing indefinitely.



Topology	Activation Function	Cost Function	Optimizer	Learning Rate	Learning Rate Decay	Dropout Probability	Batch Size
[63, 39]	ReLU	MSE	Adam	0.0762	0.000043	0.334	48.15

Figure 4.1: Gene encoding in the system.

It is possible to set further restrictions for the hyperparameters in the implemented system, like setting a maximum learning rate or a specific number of hidden layers. However, for the main portion of tests described in Chapter 5, no such restrictions have been used.

To evolve the different hyperparameters at the same time in such a direct manner differs from most of the previous work within neuroevolution, including those discussed in Chapter 3. It is a straightforward approach of evolving the hyperparameters, without abstracting it within indirect encoded functions or module-like components, and is therefore a natural approach to try. This is tied to research question 1.

4.1.2 Main Components

This subchapter describes the main components of the genetic algorithm used by the system. The population is first initialized, and then, for every generation, parents are selected, children are created and mutated, neural networks are trained and evaluated, and poorly performing individuals are removed from the population. Figure 4.2 describes this process through a high-level pseudo-code, and Figure 4.3 gives a more graphical representation of the same.

```
population = initializePopulation()

for generation in numberOfGenerations:

    for child in numberOfChildren:

        parents = selectParents(population)

        child = createChild(parents)

        child = mutate(child)

        trainedNet = trainNet(child)

        evaluation = evaluate(trainedNet)

        population = population.append(child, evaluation)

    population = removeIndividuals(population)

finalEvaluation = evaluateFinalPopulation(population)
```

Figure 4.2: High-level pseudo-code for the main system.

In the next sections, each component will be explained in richer detail.

Initialization of the Population

At the start of the genetic algorithm's run, the population needs to be initialized. This is done almost completely randomly within the search space, in order to get a diverse initial population. However, it differs a bit for the different hyperparameters.

The topology is not initialized randomly. Instead, every individual starts with exactly one hidden layer with one hidden neuron in it. This is because incremental complexification

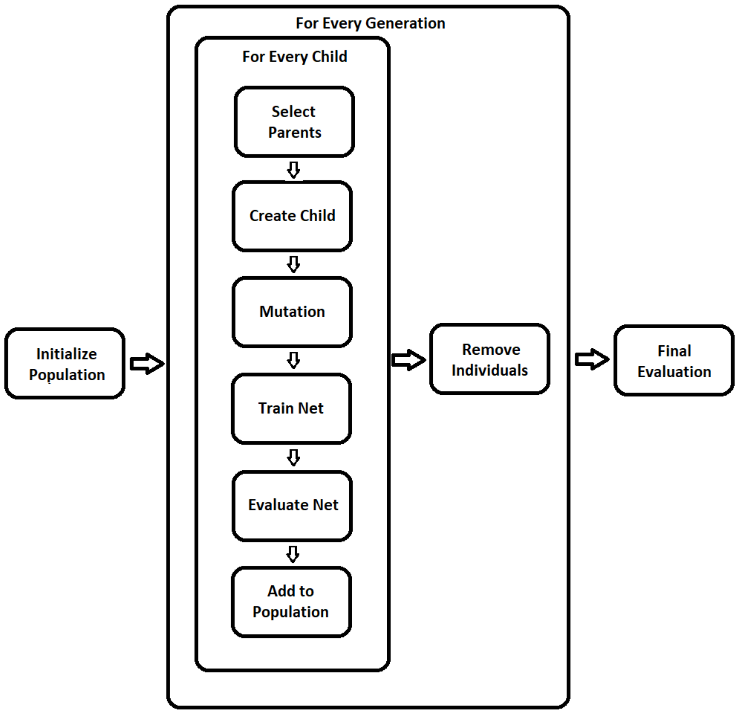


Figure 4.3: Graphical illustration of the main system.

is used, as research question 4 concerns. Starting from a minimal structure and thereafter executing incremental complexification is inspired by NEAT (see Chapter 3.2.2).

The activation function, cost function and optimizer are initialized to random values for each individual. As there are few possible values for these hyperparameters, it is likely they will all be represented in the initial population.

Initial values for the learning rate and batch size are drawn from exponential distributions (i.e. $f(x) = \lambda e^{-\lambda x}$). This has the advantage that the values will always be positive, and that small values are more likely, but larger ones are also possible. For the learning rate, $\lambda = 8$ in the exponential distribution, and for the batch size, $\lambda = \frac{1}{64}$. These parameter values are selected to get many values in the areas which usually perform well for these hyperparameters (around [0.01, 0.1] for the learning rate, and [32, 128] for the batch size). This gives a small human bias, but it is almost negligible (see Chapter 4.1.4).

The initial values for the learning rate decay could have been drawn in the same fashion, but as it requires very small values which might depend a lot on the time each neural net is allowed to train (short training time means fewer steps, which in turn means the decay may be larger), another approach is chosen. A random coefficient in the continuous interval [1, 8], and a random exponent in the discrete interval [-12, -2], are both drawn. The learning rate decay lrd will be:

$$lrd = coeff \cdot 8^{exp}$$

Here, $coeff$ is the coefficient drawn and exp the exponent drawn. This means the learning rate decay will explore values in a wide range, from extremely small and almost negligible, to relatively large ones.

Finally, the dropout probability is drawn from a beta distribution with $\alpha = \beta = 2$. This distribution has the advantages of always drawing a value within the interval [0, 1], and that it is more likely to draw a value close to the middle, making more extreme values appear rarer. Figure 4.4 illustrates this distribution.

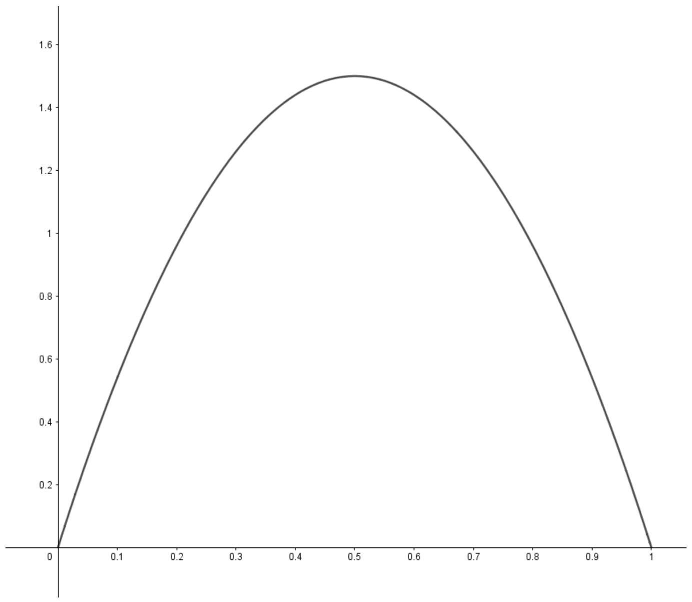


Figure 4.4: Beta distribution with $\alpha = \beta = 2$. The x-axis represents the possible values to be drawn, and the y-axis represents the probability densities.

Parent Selection and Child Creation

The system can create individuals in four different ways. The first is during initialization, as explained above. Later, during every generation, three other methods for creating children are used. These are:

- Children created from two parents with uniform crossover
- Children created from a single parent
- Children created as random immigrants

The first of these methods require two parents to be selected. Parent selection will be explained soon. The genes of the two parents are combined with uniform crossover. This means each part of the genome is drawn from a random parent, completely independent of what parent the other parts are drawn from (see Chapter 2.1.2). For the system, a "part" is considered one hyperparameter, meaning the whole hyperparameter value is inherited from the same parent. That means, for instance, that the whole topology comes from one of the parents, instead of the child inheriting some layers from one parent and some from the other. This choice has been made based on early tests, indicating this was the better approach. After crossover, the child will be mutated, something which will be explained in the next section.

The second method requires only one parent to be selected. The child will originally get an exact copy of its parent's genome. This means mutation becomes particularly important, as the only thing making the child different from its parent.

A random immigrant is an individual put into the population without a base in another individual in the population, i.e. no parents (Tinós & Yang, 2007, p. 256). A random immigrant is created in almost the same fashion as the individuals in the initial population. The only hyperparameter that differs from the approach used there, is the topology. This is necessary as the initial population only contained networks with a single hidden neuron. This is too few to be competitive with more complex structures for most problem domains, meaning the random immigrants would be outperformed by the other individuals, which most likely would have grown larger through mutation by now. To solve this issue, the topology of a random immigrant will contain a random number of layers drawn between the smallest and largest number of layers in the population, and a random number of neurons in each layer drawn between the fewest and largest number of neurons in any layer in any individual in the population. This means the random immigrants are not completely independent of the other individuals in the population, but are still close to being so.

Both children with one and two parents are used by the system as early testing showed they both worked well. Also, as children with only one parent become more similar to their parent, which most likely performs rather well, these individuals could be regarded more exploitatory, while those of two parents could be more exploratory. At least, that was the initial idea, but the results in Chapter 5.2.3 indicate it is the other way around. Random immigrants are even more exploratory, and might help if the individuals of the population have become very similar and trapped in a local minimum, by introducing entirely new individuals. During early testing, it was also tried to use three parents, and to have a larger probability that each part of the genome will be drawn from one parent than from another. This was inspired by Chandra and Yao (2006) (see Chapter 3.2.3). All of these methods worked approximately equally well to the methods used by the system, but have not been implemented in the final system to keep it simple, and because they did not add anything in particular either.

What method will be used to create a specific child, is drawn randomly and independently. The probabilities for the different methods are set through two genetic algorithm parameters (see Chapter 4.1.4 for details). In general, the probability of creating a random immigrant should be rather small, as it should only be used to prevent the population from becoming too uniform.

The two methods above which require parents, need these to be selected somehow. A rank-based selection method is used (see Chapter 2.1.2), where the probability is proportional to the individual's fitness rank. This method is used because if the individuals have quite similar fitness values, selecting parents with probability proportional to the value itself will make an almost uniform distribution. The fitness value which the selection is based upon, is explained in the section below about network training and evaluation.

Mutation

Of the three methods for creating children used by the system, the first two use mutation. Mutation is especially important when only using one parent, to make sure the child differs from its parent. The mutations are selected dependently of each other, in a way so that exactly one mutation is selected for each child. Random immigrants do not need mutations, as they are already randomly created.

A mutation in the gene for activation function, cost function or optimizer will simply draw a new, random value with uniform probability. This means the new value is independent of the old value, and is a consequence of the fact that the system treats all values for these hyperparameters as equally dissimilar, in order to avoid a human bias, as discussed before.

The learning rate, learning rate decay and batch size is mutated by first drawing a random value from a beta distribution with $\alpha = \beta = 2$, and then transforming this value to the interval [-1, 1] by multiplying it by 2, and subtracting 1. This new value is set as an exponent with 2 as base, and the hyperparameter value is finally updated by multiplying the current value with the calculated one. Mathematically, that is:

$$\begin{aligned}
&v_1 = beta(2,2)\\
&v_2 = 2v_1 - 1\\
&v_3 = 2^{v_2}\\
&value_{new} = value_{old} \cdot v_3
\end{aligned}$$

This means that for these hyperparameters, the new value is dependent on the old one, and that the maximum change is a doubling or halving of the previous value.

The dropout probability is also mutated by drawing a random value from a beta distribution with $\alpha = \beta = 2$. However, this value is reduced by $\frac{1}{2}$ to transform it to the interval [-$\frac{1}{2}$, $\frac{1}{2}$]. This value is added to the current probability to produce the new one. If the value is below 0 or above 1, it is simply set to that respective extreme value, as it must be a probability. It can be described mathematically like this:

$$\begin{aligned}
&v_1 = beta(2,2)\\
&v_2 = v_1 - \frac{1}{2}\\
&value_{new} = median(0, value_{old} + v_2, 1)
\end{aligned}$$

For the topology, there are four possible types of mutation available for the system:

- Add neurons: A random hidden layer is selected, and neurons are added to this layer. The number of neurons are drawn with uniform probability between 1 and the number of neurons already in the layer, meaning the number of neurons in a layer can maximally be doubled with a single mutation.
- Add layer: A single layer is added. Its position relative to the other layers is drawn uniformly. A random, already existing layer is drawn to determine the maximum possible number of neurons in the new layer. The number of neurons is drawn uniformly between 1 and the number of neurons in that randomly selected layer. This means a new layer will never contain more neurons than the already largest layer in the net. In the spirit of incremental complexification, one could consider only adding new layers with a single neuron in them, like when initializing the population. However, this would not work well because all the information from the layers before would then have to be sent through a single neuron, and a lot of information would be lost. Therefore, such networks would have been outperformed, and never have a chance to grow larger.
- Remove neurons: This works in the same fashion as adding neurons, by selecting a single layer to mutate. The number of neurons to be removed is drawn between 1 and half the number of neurons in the layer.
- Remove layer: Removes a random layer, selected with uniform probability.

It is these topology mutations that make the incremental complexification of research question 4 happen. By adding neurons and layers over time, the structure grows from the minimal one which all the individuals in the initial population had. The mutations removing neurons and layers are used rarely, and are only part of the system to make it possible to discover less complex structures. Without this, the population could be trapped in a search space of minimum the complexity of the least complex individual in the population. The input and output layers of the neural networks are of course not mutated, as the number of neurons in these are set by the problem attempted to be solved.

The probabilities for the different mutations to be selected are not all equal. This is because some mutations are more important than others. The topology mutation is critical, because without it, there would not be any incremental complexification. Also, as the hyperparameters holding continuous values (i.e. the learning rate, learning rate decay, dropout probability and batch size) have a far larger search space than the ones holding atomic values (i.e. the activation function, cost function and optimizer), the former have a higher mutation probability than the latter. As exactly one mutation is always selected, the mutation rate can be regarded as 1. This is quite high for a genetic algorithm, but early testing indicated it worked well for this system. The exact probability for each mutation is shown in Table 4.1.

An exciting idea attempted early on, is to let the probabilities for the different mutations vary dynamically based on what has worked well earlier in the run of the system. The initial results were not too promising, but it could be an interesting subject for a future work.

Mutation	**Probability**
Topology, Add Neurons	25 %
Topology, Add Layer	22.5 %
Topology, Remove Neurons	1.5 %
Topology, Remove Layer	1 %
Sum, Topology	**50 %**
Activation Function	4.55 %
Cost Function	4.55 %
Optimizer	4.55 %
Learning Rate	9.09 %
Learning Rate Decay	9.09 %
Dropout Probability	9.09 %
Batch Size	9.09 %
Sum, Total	**100 %**

Table 4.1: Probabilities for the different mutations in the main system.

Network Training and Evaluation

In order to acquire a fitness value for each individual, which in turn will be used for parent selection and individual removal, the neural networks represented by the genes must be trained and evaluated. The nets are only trained and evaluated in the generation where they are created, and then the results are saved for calculating the fitness value in future generations. This means every neural network is trained exactly once.

Training is done using the backpropagation algorithm on the training part of the dataset. Even though the networks are trained in parallel, this part of the system takes up the majority of its runtime. This also means spending a few additional resources on the other parts of the system can be justified, as it does not affect the total runtime by a lot. The total runtime of the system, t_{run}, can be approximated as follows:

$$t_{run} \approx t_{train} \approx \frac{gens \cdot c \cdot t_{net}}{CPUs}$$

Here, t_{train} is the total time used to train all neural nets, $gens$ is the number of generations, c is the number of children created per generation, t_{net} is the time each neural network is allowed to train, and $CPUs$ is the number of CPUs used by the system. The time each neural network is allowed to train is a parameter of the system (see Chapter 4.1.4 for details). The system could train the nets on GPUs instead of CPUs, however, tests have indicated it runs faster on CPUs. The main reason for this is probably that the networks need to be larger in order for a GPU to outperform a CPU.

During training, a second partition of the dataset, called the early stopping set, is used to evaluate the training progress independently of the training set. Two parameters of the system, a validation interval and a patience, is used for this. The validation interval determines the number of training steps between each time the network is evaluated by the early stopping set. If no improvement has been found in the number of steps specified by the patience parameter, training is stopped in order to save time. Additionally, when training is completed, regardless of whether it is because the training time is exceeded or there has not been an improvement for too long, the weights of the network are rewound to how they were when they performed the best on the early stopping set. All of this is done in order to avoid overfitting if the network is trained for too long (see Chapter 2.1.1).

When training is completed, the network is evaluated on a third part of the dataset, called the validation set. The results from this evaluation is what is used to calculate the fitness of the individual. This means the fitness value is only based on data independent of that having been used during training, preventing the genetic algorithm from overfitting to the training data.

The fitness value is based on three different properties of the individual:

- The accuracy of the network
- The diversity of the network
- The age of the individual

The accuracy of the network is simply the ratio of correctly classified validation examples. This works well for fairly balanced classification datasets. For unbalanced datasets, a measure like F1 score might work better, however, early testing indicated it worked worse for the balanced datasets selected for testing in this thesis. Also, in the case of regression problems, it is necessary to modify this evaluation to some degree. These topics are further discussed in Chapter 6.2.

The diversity of the network is also taken into account, in order to avoid an overly uniform population, and is covered by research question 3. This is inspired by novelty search (see Chapter 3.2.3). Pairwise Failure Crediting (PFC) (also explained in Chapter 3.2.3) is used to measure diversity. As it measures the diversity for every pair of individuals, this must be done in every generation. However, the neural net predictions which the measure is based on only have to happen once. Therefore, these results are stored so that the whole evaluation of the neural network will not have to be repeated. PFC evaluates the diversity based on the phenotype (i.e. neural net) rather than the genome directly. This is also suggested by Lehman and Stanley (2011). However, a couple of techniques for measuring diversity directly on the genome were attempted during early testing. Measuring difference between hyperparameters directly provided poor results, while building a family tree and using distance measures to find the diversity was more promising. That being said, it has not been brought into the final system as PFC is both simpler and performs slightly better.

Taking the age of the individual into account, called aging, is inspired by Real et al. (2019) (see Chapter 3.2.5). The age starts at 0 in the generation where the individual is created, and is incremented by 1 for every generation. As the age affects the fitness value negatively, the algorithm will prefer younger individuals, hence forcing the system to stay innovative and be able to pass the individuals' genetic information from generation to generation, as old but well-performing individuals will be removed over time. The system stores the individual with the best accuracy overall, even if it is removed from the actual population, meaning there is no risk of losing the best individual due to aging. This can be considered a type of elitism (see chapter 3.2.4). Just as the traditional diversity measured by PFC can be regarded as a spatial diversity ensuring the population is not too uniform at any given time, aging resembles a temporal diversity ensuring the population changes over time to explore a larger part of the search space. As incremental complexification pushes the population continually toward larger, more complex topologies, one could think that the best individuals are discovered in early generations for problems were the best solutions are found with small nets. However, as mentioned above, as long as the best performing individual overall across all generations are stored, this is not a real issue, as the best solution will never be forgotten.

The relative importance between the three properties listed above is determined by an α-parameter. Its value will always be in the interval [0, 1], where 0 means only accuracy is taken into account, and 1 means only diversity and aging is taken into account. In early generations, it makes sense to prioritize diversity in order to explore a large part of the search space. In later generations, when the algorithm is getting closer to the end, it seems better to focus on accuracy in order to find the very best configurations. Because of this, α_i is defined as follows, where i is the current generation ($i = 0$ when initializing the population), $gens$ is the total number of generations, and r is the ratio of final generations where only accuracy should be taken into account:

$$\alpha_i = max(0, 1 - \frac{i}{gens \cdot (1 - r)})$$

This means α starts at 1, and is lowered linearly until it is 0. Then, the system parameter r determines the number of generations with $\alpha = 0$ before the algorithm is done running. As an example, with $gens = 40$ and $r = 0.25$, α will be lowered by $\frac{1}{30}$ for each of the first 30 generations, before the last 10 generations will be run with $\alpha = 0$. Other schedules for the value of α were tested early in the project. These tests indicated that it was valuable to lower α gradually, and to run some additional generations with $\alpha = 0$ in the end. However, whether α was decreased linearly, or e.g. followed an S-shaped curve, did not really affect the results. The linear version is picked mostly because it is very simple, yet works well.

Gradually moving from a diversity focused to an accuracy focused population differs from the previous related work found using some kind of diversity measure (see Chapter 3.2.3). Lehman and Stanley (2008, 2011) and Such et al. (2018) look solely for diversity, and Chandra and Yao (2006) does not let the focus gradually change from diversity to accuracy.

As the accuracy of the different networks are often quite similar (e.g. ranging from 95 % to 97 % for MNIST), using this value directly might create a problem for the fitness function. If the diversity varies a lot more, this part will be of much more importance than the accuracy, even if the α-parameter says otherwise. E.g. if the accuracy ranges from 95 % to 97 %, while the diversity ranges from 30 % to 80 %, the diversity can be regarded as 25 times more important for the final fitness value. To avoid this problem, the accuracy and diversity values for each individual are scaled to the range [0, 1], so that the lowest value in each generation will be 0, and the largest 1.

Finally, the formula for the fitness f of an individual i is:

$$f_i = (1 - \alpha) \cdot s(acc_i) + \alpha \cdot (s(pfc_i) - \frac{age_i \cdot a}{gens})$$

Here, α is defined as above, s scales the argument to the range [0, 1], acc_i is the accuracy of i (i.e. the ratio of correctly classified validation examples), pfc_i is the PFC diversity score of i, age_i is the age of i, and a is the age factor, a parameter of the system determining the relative importance of the age. This is divided by $gens$, the number of generations, so that age matters more for runs with few generations, which have a shorter time to find good solutions. pfc_i is defined as in Chapter 3.2.3:

$$div_{i,j} = \frac{diffPreds(i,j)}{wrongPreds(i) + wrongPreds(j)}$$

$$pfc_i = \frac{\sum_{j=0}^{N} div_{i,j}}{N - 1}$$

The fitness value can be maximum 1, but there is no strict minimum value as the aging penalty grows linearly with the age. If there is no aging penalty (i.e. $a = 0$), the minimum value is 0.

Individual Removal

As a final part of every generation, poorly performing individuals are removed from the population. As many individuals are removed as were created, making the population size constant through the run of the algorithm. Individuals are removed stochastically, where the probability of surviving until the next generation is proportional to the individual's fitness rank. This is just as with parent selection; individuals ranked high have a higher probability of being selected as parents, and also to survive until the next generation.

4.1.3 Other Aspects

Here, some additional aspects of the main system will be discussed.

In the previous subchapter, three parts of the dataset were introduced; a training set, an early stopping set, and a validation set. The system splits the dataset into four parts in total, where the last one is a testing set. The purpose of this set is to give an objective, independent evaluation of the best networks when the genetic algorithm is finished running. It should be emphasized that this part of the dataset is only used for a final evaluation, and never used by the genetic algorithm to guide the search.

Whether or not these four sets are enough for the system, depends on the problem to be solved. There are two main types of problems which neuroevolution can attempt to solve:

1. If the objective is to find optimal *hyperparameters*, the nets must in the end be trained on at least one independent training set which has not been used by the genetic

algorithm previously. This is because the evolved hyperparameters might not work well in general, but just did so by luck for the training set used by the genetic algorithm. By training on an independent set, an objective evaluation is achieved. Preferably, the training should also be run multiple times on the same set, as it is not deterministic.

2. If, on the other hand, the objective is to find the optimal *trained network*, it is enough to train the net once. This is because one can argue that even though the evolved hyperparameters might not work well on general training sets, they did work well with the one used by the genetic algorithm, and the trained network does execute the classification task with a high accuracy.

For the thesis, the objective has been defined as one of type 2, meaning the four different parts of the dataset described above will be enough. The main disadvantage with the first problem type is that as it requires more dataset partitions, each partition will become smaller.

The system tries to minimize the use of data preprocessing and feature engineering, as it is not relevant for the research questions to be answered. In the future, the system could be enhanced by such methods, and probably improve the results to some degree. Currently, only the following is done with the data before being fed into the neural network:

- One-hot encoding of categorical features
- One-hot encoding of labels
- Shuffling
- Scaling of features to the interval [0, 1]

The first three are almost essential in order to create a working classification net, while the last one is a very simple method for making the nets converge faster (Brownlee, 2019a).

A simple ensemble method has been implemented in the system. Every individual in the final population gives a weighted vote for what category the current testing set example belongs to. The weight of the vote is the accuracy of that neural network on the validation set, meaning a better performing network will give a vote weighing more. Instead of giving its full vote to one category, each individual gives a value to every category; that value being the predicted probability of the example belonging to that category. Mathematically, the actual vote v given to category i by individual j, where p is the predicted probabilities and acc is the accuracies, is:

$$v_{i,j} = p_{j,i} \cdot acc_j$$

The category with the highest score in total will be predicted by the ensemble. As the weighing is based on the validation set, while the ensemble is used to make predictions on the test set, the weighing is neutral, hence making the results representable. This is a simple form of ensemble to illustrate how easily they can be implemented in a neuroevolution system like this one, which already has a lot of different models (i.e. neural networks) available. Chapter 6.2, future work, discusses how this can be extended with even more powerful ensembles.

4.1.4 A Hyperparameter Change of Basis

While the system has been described in the previous subchapters, many parameters have been introduced, like the ratio of final generations where the fitness function should only consider accuracy, and the age factor. The observant reader might already have asked the question: "What is the point of a system like this one, if it introduces as many new parameters to be set by the user as it tries to remove by evolving them instead?". This subchapter will discuss this issue, while taking a closer look at the new parameters introduced by the system. For this discussion, and the rest of the thesis, the term "hyperparameter" refers to those evolved for the neural networks, while the term "parameter" refers to those used by the genetic algorithm in the system implemented.

There is a big difference between the hyperparameters in a neural net, and the parameters used by this system and genetic algorithms in general: The parameters in the latter are much more intuitive, and each one of them has a clear trade-off between two different aspects. Furthermore, the choice of value for parameters of a genetic algorithm usually affect the results less, and default values usually work fairly well. Hence, a neuroevolution system like this one does not reduce the number of parameters, but makes it much simpler for the user to set their values. Having a lot of parameters are not negative by itself, as long as one is not dependent on a lot of trial-and-error before achieving good results.

The concept of clear trade-offs should be explained in some more details. None of the hyperparameters of a neural network have such a trade-off. As an example, one can look at the topology. What are the effects of increasing the number of neurons in a layer? More weights take more time to update, and might represent more complex functions, but could also increase the chance of overfitting. But except for these very general remarks, not much can be said with certainty. What about the activation function? How does it affect the results if it is switched from ReLU to Softplus? Hardly anything can be said in general about this. Additionally, it is often the relationships between the hyperparameter choices that matter for a neural network. For example, what activation function that will work well in a setting might be highly dependent on the choice of cost function. The conclusion is that one simply has to try different values, and wait for the results.

On the other hand, all of the parameters which *can* be set by the user in this system, and in most neuroevolution systems, have a clear trade-off. Here, *can* is use instead of *must*, as the default values will usually provide good results. There are mainly two types of trade-offs for the parameters in this system:

1. Trade-off between accuracy and time usage
2. Trade-off between exploration (i.e. diversity) and exploitation

A trade-off between accuracy and time usage means that setting a value close to one end of the scale (e.g. a high value) will prioritize a high accuracy but use more time getting there, while setting a value in the other end of the scale (e.g. a low value) will have the opposite effect. One could therefore set these values based on the amount of time one has available, and the required accuracy for the solution. An example of a parameter with this trade-off is the number of generations, where a larger value might achieve better results, but takes a longer time to run.

A trade-off between exploration and exploitation means whether the system should focus on diversity, or primarily stay in the areas of the search space where the most promising solutions found so far already exist. An example of a parameter with this trade-off is the rate of children to be created as random immigrants. A higher rate means more random individuals spread across the search space, hence more exploration. A lower rate will create more individuals as children of the individuals already performing well, hence more exploitation. This might not be as easy to determine as the accuracy-time usage trade-off, but at least the parameter values determine a one-dimensional property (i.e. the exploration-exploitation trade-off), where an adjustment in one direction should have a predictable effect (unlike the neural network hyperparameters).

Table 4.2 lists every parameter used by the genetic algorithm of the system, and what trade-off they concern. As one can see, every parameter has a clear trade-off of one of the two types listed above. Most of the parameters have already been explained in the previous subchapters. Its quite obvious that the first four (i.e. generations, population size, children per generation and training time per individual) have an accuracy-time usage trade-off. A lower validation interval means the exact best version of the network is more likely to be found, hence increasing the accuracy, but also increasing the time usage, as the evaluations must be done more frequently. A higher patience means the training will stop early rarer, hence giving the nets the chance of improving, but at the cost of a longer training time. The PFC sample rate is the fraction of the validation examples which should be used to calculate the PFC diversity value, and the PFC individuals to compare with is the number of individuals each individual should be compared with to calculate the PFC diversity value. A lower value will for both parameters give a less precise estimate for the actual PFC value, but will also save some time.

Parameter	**Trade-off**
Generations	Accuracy-Time Usage
Population Size	Accuracy-Time Usage
Children per Generation	Accuracy-Time Usage
Training Time per Individual	Accuracy-Time Usage
Validation Interval	Accuracy-Time Usage
Patience	Accuracy-Time Usage
PFC Sample Rate	Accuracy-Time Usage
PFC Individuals to Compare With	Accuracy-Time Usage
Crossover Rate	Exploration-Exploitation
Random Immigrant Rate	Exploration-Exploitation
Age Factor	Exploration-Exploitation
Final Generations with Accuracy Only Ratio	Exploration-Exploitation

Table 4.2: Parameters of the main system.

It is natural to think that a higher crossover rate will make the system more exploratory, as children with two parents will be more novel than children with just one (as the latter will be almost identical to its parent, except for the mutation). However, tests run indicate it is the other way around (see Chapter 5.2.3). That said, this does not remove the fact that there is a clear trade-off. A higher random immigrant rate will make the system more exploratory, naturally. The same goes for the age factor, as a higher value will force the old individuals out of the population faster, hence making room for exploring more new individuals. A higher number of final generations with accuracy only will make the system more exploitatory, as diversity is disregarded in these generations. For more information about how to set the different parameters, see appendix 1. Tests attempting to show these clear trade-offs can be found in Chapter 5.2.3.

To summarize, the clear trade-offs make it much easier to set the values for the parameters. One could argue some of the numerical values presented earlier in this chapter, like the different mutation rates, also are a kind of parameters, and that the value of these are not trivial to set or have a clear trade-off. However, as long as these parameters do not have to be set by the user, as they have default values which work well for most or all problems, this is not really an issue. Also, even if these values are not fine-tuned, they do not affect the results by much. This also indicates the robustness of the system; minor changes to the system only cause minor changes to the results. This is further discussed together with the results in Chapter 5.2.3.

The fact that genetic algorithms have parameters which are easy to set also explains why neuroevolution, with a genetic algorithm on top of a neural network, is a good idea. Why exactly these two types of machine learning, and why exactly two levels on top of each other? What about putting a third algorithm on top of that, to tune the parameters of the genetic algorithm? As neural networks generally perform very well in a wide area of problem domains, it makes sense to have these on the lowest level; the level which is actually going to solve the task. It is also not necessary to put another algorithm on top of the genetic one, as the parameters of that one can easily be set by a user, and the search for good values does not need to be automated by another algorithm on top. Furthermore, the time complexity grows exponentially with respect to the number of levels, meaning more than two levels will be far too computational expensive for most practical tasks. This is derived in appendix 3.

4.2 Merging Subpopulations System

The merging subpopulations system is inspired by distributed genetic algorithms, where several subpopulations are subject to evolution in parallel. See Tanese, Holland and Stout (1989) and Arellano-Verdejo, Godoy-Calderon, Alonso-Pecina, Arenas and Cruz-Chavez (2017) for examples of distributed genetic algorithms in general. There are two main motivations for using subpopulations: the different subpopulations can run independently of each other in a distributed environment, and they might evolve different types of individuals, hence ensuring more diversity in the system overall. It is primarily the latter of these reasons which have motivated the implementation of this additional system during this project. Even with the diversity stimulating techniques used in the main system, there is still a tendency that the individuals only explore a rather small part of the search space.

This is also relatable to the usage of species, like in NEAT (see Chapter 3.2.2). Each subpopulation could be regarded as a species which is evolved independently of the other species. The difference lies in the fact that there are no restrictions on how dissimilar the individuals of one subpopulation can be, while the individuals of a species usually have to be quite similar.

It is only the main idea of using subpopulations in general which has been inspired by the field of distributed genetic algorithms, and the papers listed above. The details about how the system works have been designed from scratch in this thesis' work through intuition and testing. The source code of the merging subpopulations system can be found in appendix 4, while the system manual in appendix 1 covers all three systems.

Most of this system works in the same fashion as the main system. The same hyperparameters are evolved, and every generation consists of the same components (i.e. child creation, network training and evaluation, and individual removal). However, it happens in more than one population at once. One additional parameter is required by this system, determining the number of starting populations. There are three requirements for the value of this parameter, here represented by s:

1. $s = 2^n$ (It should be a power of 2)
2. $inds \equiv 0 \ (mod \ s)$ (The total number of individuals across all populations should be divisible by the number of starting populations)
3. $gens \equiv 0 \ (mod \ log_2(s) + 1)$ (The total number of generations should be divisible by the logarithm of the number of starting populations plus 1)

Here, n is any positive integer, $inds$ is the number of individuals in total, and $gens$ is the total number of generations. The second of these requirements is quite obvious, as the number of individuals must be divisible by the number of starting populations in order to make all populations equally large. This is not extremely important, but makes sense in order to give all populations the same starting point. The other two requirements will be explained soon, but first one should know how the algorithm works.

All of the starting populations are initialized as normal, and first run separately for $\frac{gens}{log_2(s)+1}$ generations. Thereafter, pairs of populations are merged together, doubling the size of each population, but also halving the number of populations. Then, the new populations run for another $\frac{gens}{log_2(s)+1}$ generations, before another merge happens. This repeats until there is only one population left. This final population runs for a last $\frac{gens}{log_2(s)+1}$ generations, before the algorithm has completed its run.

Figure 4.5 illustrates this for the case of a total number of 80 individuals, and a total number of 80 generations, with 8 starting generations. For the first 20 generations, there will be 8 populations with 10 individuals in each. For generations 20 to 40, there will be 4 populations with 20 individuals each. Thereafter, generations 40 to 60 will have 2 populations, each of size 40 individuals. Finally, the last 20 generations will only have 1 population, with 80 individuals in it.

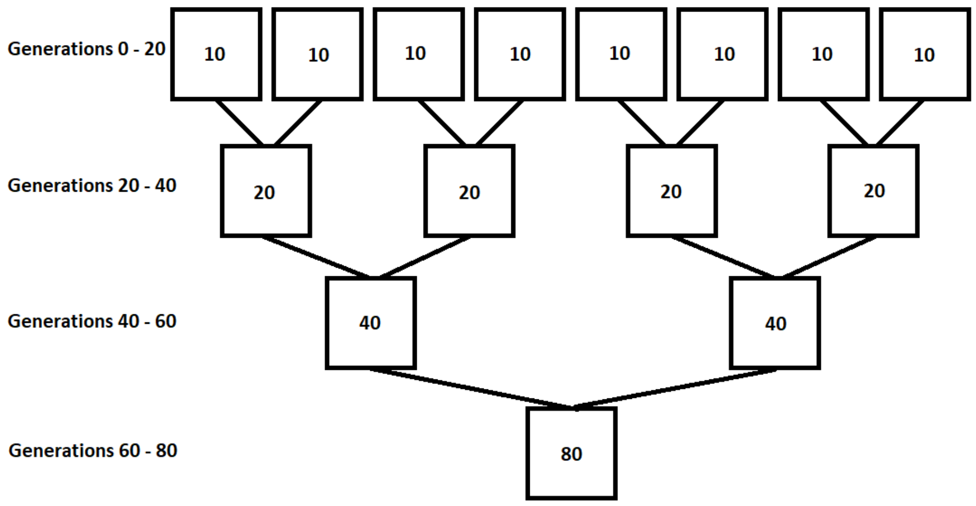


Figure 4.5: The merging subpopulations system with 80 generations, 80 individuals and 8 starting generations. Each box is a population, with its size within.

Now, the two other requirements for the s parameter can be explained. If its value is not a power of 2, it cannot be merged down to a single population by always merging pairs of populations. If it is not divisible by $log_2(s) + 1$, there will not be the same number of generations between each merge.

The number of starting populations has a trade-off of type exploration-exploitation (see chapter 4.1.4). More populations will explore a larger part of the search space, hence more diversity and exploration, while fewer populations means the size of each will be larger, hence allowing more promising individuals to be created and therefore more exploitation.

4.3 Convolutional System

The convolutional system evolves convolutional neural networks (CNNs) instead of normal, dense networks. It is based on the main system, with the only real difference in how the topology hyperparameter is treated. Also, the system requires the feature data to have a three-dimensional structure, with a width, height and depth, where the depth can be regarded as color channels for an image. The source code of the convolutional system is found in appendix 5, and how it is run is explained alongside the other systems in the system manual of appendix 1.

Because the topology of a CNN can hold several types of layers (e.g. convolutional layers, pooling layers, and dense layers), and each of these have several possible configurations, the search space would grow extremely large without any restrictions. It is certainly interesting to evolve such nets freely to see if new innovative architectures would be discovered, however, with the limited time and computational resources of this thesis, some restrictions have been set in order to reduce the search space. First, the topology will always have a structure like the one illustrated in Figure 4.6.

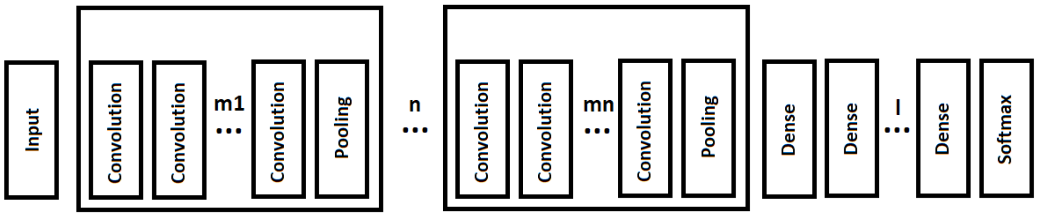


Figure 4.6: Convolutional topology.

As can be seen, it first consist of n layer groups, where each group i consists of m_i convolutional layers and one max pooling layer in the end. Every m_i can be different. After the n layer groups, l normal, dense layers follow, before the softmax output layer. Before the dense layers, the output from the last pooling layer must be converted into one-dimensional data. Ahire (2018, p. 132) claims an architecture like this one works well in general for convolutional networks. Batch normalization is executed at the end of every convolutional and dense layer, and the activation function is applied after these types of layers as well. Dropout is only used for the dense layers, as it does not work as well for convolutional layers directly (Ghiasi, Lin & Le, 2018).

Each layer has hyperparameters which determine its behavior. In the current system, the following hyperparameters are evolved depending on layer type:

Convolutional Layer

- Number of filters
- Filter size
- Stride
- Padding

Pooling Layer

- Size

Dense Layer

- Number of neurons

If the reader is unfamiliar with these concepts, see Ahire (2018, Chapter 8). The number of filters can be any positive integer. The filter size is an odd integer of value at least 3, as odd filter sizes usually perform better (Sahoo, 2018, paras. 7-9). The stride can be any positive integer. The padding is just a binary value determining if zero-padding should be used for the layer. The size of the pooling layer is 2 or 3, as these values are almost the only ones used in practice (Ahire, 2018, p. 128). The stride of the pooling layer is always set to 2, while the type is max pool, as other configurations rarely work better than this (Ahire, 2018, p. 128). The number of neurons in a dense layer is just as it is explained for the main system. Of course, the values of these hyperparameters could have been set even more freely, and more hyperparameters could have been added, but for the same reasons as explained above, it has not been done in order to reduce the search space.

The rest of this subchapter will explain how the topology is initialized and mutated for this system, as the rest works in the same fashion as in the main system.

The idea of starting from a minimal structure and gradually mutating it larger through incremental complexification has been inherited from the main system. However, as the search space is so large, the topologies may be initialized to slightly larger than minimal structures (but still quite small). Every initial individual contains only one layer group (as explained above, a layer group consists of convolutional layers before a final pooling layer), with one convolutional layer in it. This convolutional layer will have between 1 and 32 filters, drawn with uniform probability. The filter size f and stride s are drawn from geometric distributions g, like this:

$$f = 2g(\tfrac{1}{2}) + 1$$
$$s = g(\tfrac{1}{2})$$

This means there is a higher chance for lower values, but there is no theoretically maximum value that can be drawn. Also, the filter size will always be an odd number, as was discussed previously. Finally, the value of the binary padding type of the convolutional layer is drawn uniformly. For the pooling layer, the size which can have values 2 or 3 is drawn with uniform probability.

Only one dense layer will be available in the initial population, just like in the main system. However, in order to speed up the evolution, the individuals might have more than one neuron in that layer. The number of neurons are drawn from a geometric distribution with $p = \frac{1}{32}$.

Random immigrants are created in the same fashion as the initialized individuals, except for the fact that they might contain more layer groups, convolutional layers per layer group, filters per convolutional layer, and dense layers. The number for each of these are drawn randomly between 1 and the number of the individual in the population with the highest of that respective number. This resembles how random immigrants are created in the main system.

Ten different topology-related mutations are available to the system. Each of these will be explained briefly.

Add a Layer Group

This group is created in the same way as during initialization, except that it might contain more than one convolutional layer and more than 32 filters per layer. The maximum number for these values are equal to the maximum currently present in any layer group of this individual. The layer group is added in a random location of the topology (but before the dense layers).

Remove a Layer Group

A random layer group is removed from the topology.

Add a Convolutional Layer

A convolutional layer is added at a random position within a random layer group. Its hyperparameters are drawn in the same way as during initialization, except that it may contain as many filters as the maximum already present anywhere in this individual.

Remove a Convolutional Layer

A random convolutional layer is removed from a random layer group.

Modify a Convolutional Layer

A random convolutional layer in a random layer group is selected for modification. The filter size, stride and padding are drawn in the same way as during initialization. When it comes to the number of filters, a random integer in the interval $[-floor(\frac{f_c}{4}), f_c]$ is added to the current number of filters, where f_c is the number of filters currently in the layer. In this way, the number of filters should gradually increase, but it is also possible for the number to be reduced, if that becomes necessary.

Modify a Pooling Layer

A random layer group is selected, and the pooling layer of this group is modified. As the size is the only hyperparameter of a pooling layer available to the system, all that is done is drawing the value 2 or 3 with uniform probability again.

Add Neurons to a Dense Layer, Add a Dense Layer, Remove Neurons from a Dense Layer, and Remove a Dense Layer

These four mutations are equal to the ones used by the main system (see Chapter 4.1.2).

The probabilities for the different mutations are selected so that those complexifying the topology are selected more often than those reducing it, as incremental complexification is used. Also, more drastic mutations are selected rarer than less drastic ones (e.g.

adding a layer group changes the current topology much more than adding a single convolutional layer). Because there are so many mutations available, the system works a bit differently than the main system. More than one mutation can be picked at once for the same child, though some mutations are mutually exclusive (to e.g. ensure that a layer that will be removed is not modified at the same time). Table 4.3 summarizes the probabilities and dependencies for the mutations in the convolutional system.

Mutation	**Probability**	**Total Probability**	**Mutual Exclusion**
Activation Function	$\frac{1}{16}$	$\frac{1}{16}$	
Cost Function	$\frac{1}{16}$	$\frac{1}{16}$	
Optimizer	$\frac{1}{16}$	$\frac{1}{16}$	
Learning Rate	$\frac{1}{8}$	$\frac{1}{8}$	
Learning Rate Decay	$\frac{1}{8}$	$\frac{1}{8}$	
Dropout Probability	$\frac{1}{8}$	$\frac{1}{8}$	
Batch Size	$\frac{1}{8}$	$\frac{1}{8}$	
Topology	$\frac{1}{2}$	$\frac{1}{2}$	
Add a Layer Group	$\frac{1}{24}$	$\frac{1}{48}$	a
Remove a Layer Group	$\frac{1}{96}$	$\frac{1}{192}$	a
Modify a Layer Group	$\frac{3}{4}$	$\frac{3}{8}$	a
Add a Convolutional Layer	$\frac{1}{6}$	$\frac{1}{16}$	b
Remove a Convolutional Layer	$\frac{1}{24}$	$\frac{1}{64}$	b
Modify a Convolutional Layer	$\frac{5}{8}$	$\frac{15}{64}$	b
Modify a Pooling Layer	$\frac{1}{6}$	$\frac{1}{16}$	
Add Neurons to a Dense Layer	$\frac{2}{5}$	$\frac{1}{5}$	c
Add a Dense Layer	$\frac{1}{5}$	$\frac{1}{10}$	c
Remove Neurons from a Dense Layer	$\frac{1}{5}$	$\frac{1}{10}$	c
Remove a Dense Layer	$\frac{1}{10}$	$\frac{1}{20}$	c

Table 4.3: Probabilities for the different mutations in the convolutional system. Mutation groups in italic represent the probability that the below indented mutations will be considered. Total probability is the probability of that mutation multiplied by the probability that the mutation group it belongs to is picked. Mutations with the same letter in the mutual exclusion column cannot be picked for the same child.

All these choices for the convolutional system might seem a bit arbitrary, and to some degree they are. However, they are partly supported by previous research regarding convolutional networks, as described above, and partly based on the results of early tests. They could certainly be tested and tuned further, however, as this is only a side system of this thesis, the focus has been on creating a simple convolutional system that works quite well. Also, the robustness of the genetic algorithm (see Chapter 4.1.4) argues that tuning of these choices probably will not affect the performance of the system by a lot.

4.4 Summary

This chapter has described the three systems which have been implemented for this thesis. Most of the focus is devoted to the main system. The eight hyperparameters evolved are explained, as well as the main components of the genetic algorithm used. It has also been discussed that a system like this one is useful even though it introduces as many new parameters as it removes, because the new ones have a clearer trade-off and affect the results less, and hence are much easier to set by the user. Finally, the merging subpopulations system which use many separate populations to find the best neural networks, and the convolutional system evolving convolutional neural networks, are explained.

Chapter 5

Results and Analysis

The results obtained by running the systems will be presented and analyzed in this chapter. First, in Chapter 5.1, the environment in which the systems have been run is briefly described. Thereafter, in Chapter 5.2, the results are presented and analyzed. Finally, the chapter is summarized in Chapter 5.3.

5.1 Environment

A brief presentation of the environment in which the systems have been run is necessary to get a full understanding of the results. Chapter 5.1.1 shows the key aspects of the used hardware, while Chapter 5.1.2 explains the most important software components used.

5.1.1 Hardware

Neuroevolution is very resource demanding by nature, due to the large number of neural nets which must be trained. Many of the experiments of the related work described in Chapter 3.2 use powerful supercomputers with plenty of GPUs to achieve impressive results. During this thesis, only a limited number of such resources have been available, through a supercomputer. For details about this hardware, see NTNU: High Performance Computing Group (n.d.). However, this is a shared computer, meaning only a small fraction of its resources have been available for this thesis. For most runs, 20 CPU cores have been requested, and a total of 100 GB RAM. At times it has been possible to run multiple tests at the same time.

With these resources, a large number of neural nets could be trained and evaluated, but it is still only a fraction of what some other systems have used, like Miikkulainen et al. (2019) and Real et al. (2019).

5.1.2 Software

The systems have been written in Python 3, and use the machine learning library TensorFlow to quickly build and train the neural networks. Python is chosen primarily because of its efficiency when it comes to mathematical computations, easiness of running on both Windows and Linux, and compatibility with TensorFlow. The choice of TensorFlow is based on its efficiency and simplicity for creating neural nets, as well as its rich expressibility. It is only used for the neural net part of the systems, as it does not support genetic algorithms. Some other library could have been used to run genetic algorithms, but as these are quite simple to implement, and also require a large degree of flexibility to create the pursued systems, they have instead been written from scratch. For more details regarding TensorFlow, see Google Brain (n.d.).

5.2 Results

In this subchapter, the results gathered from running the three systems described in Chapter 4 will be presented and analyzed. The tests have been split into four different phases:

1. Early, informal tests determining what components should be part of the final systems
2. Testing of the final systems
3. Ablation tests related to the specific research questions
4. Trade-off tests

The first phase was conducted while the systems were being implemented, in order to make the best choices for the systems to perform well. Some of the observations from this phase were presented in Chapter 4 while describing the systems, to justify some of the choices made. The EMNIST and fashion-MNIST datasets were not used in this phase, meaning the systems have not been changed since these datasets were included. If all datasets used for the final tests had been used while tuning the systems, there would have been a risk of overfitting the systems to the selected datasets. By letting a couple of the datasets be held out during this phase, this risk is reduced. The results of this phase were quite informal, and will not be presented in greater detail.

The second phase tested the three final systems on each of the five datasets described in Chapter 2.2 (except for the yeast dataset on the convolutional system; see explanation below). These were the tests running for the longest time, and the results from these tests have the highest statistical significance, as each test was repeated for a total of 10 times. This phase is described in more depth in Chapter 5.2.1, and is primarily used to answer research question 1.

An ablation test is to remove a component of the system, in order to see if the system performs worse without it. If not, it might not be necessary for the system to hold this component. For each of the research questions, certain ablation tests have been run. The ablation tests are presented in Chapter 5.2.2.

The trade-off tests try to illustrate the clear trade-offs which in Chapter 4.1.4 are claimed to be the advantage of neuroevolution systems like these ones. These results are also tied to research question 1, because if the clear trade-offs are working as expected, the systems are easy to use and hence even more attractive. The trade-off tests are presented in Chapter 5.2.3.

The ablation tests and trade-off tests have only been run on the main system, due to the limited computational resources available.

5.2.1 Testing of the Final Systems

The purpose of these tests is to measure the performance of the systems implemented, and will in turn be used to answer research question 1 partially. A total of 140 tests have been run, each one running for approximately 70 hours (several tests have been run in parallel). The 140 test runs are divided into 14 different types of tests, meaning each type has been run 10 times. This is necessary in order to get statistically significant results, as the systems are not deterministic. The 14 types of tests are for the 3 different systems, with the 5 different datasets for each. The only combination which has not been run is the yeast dataset for the convolutional system, as this system requires image data as input. On the other hand, the convolutional system *has* been run with the chess dataset, as it is possible to treat a chessboard as an image, where each square is a pixel, and the different types of pieces can be regarded as different colors. When it comes to the parameters of the systems, Table 5.1 shows the values of these for the test runs.

Parameter	**Value**
Generations	150
Population Size	120
Children per Generation	120
Training Time per Individual	240 seconds
Validation Interval	100 steps
Patience	10000 steps
PFC Sample Rate	1
PFC Individuals to Compare With	all
Crossover Rate	0.5
Random Immigrant Rate	0.1
Age Factor	2
Final Generations with Accuracy Only Ratio	0.25
Fraction of Dataset Used in Total	See Table 5.2 below
Fraction of Used Dataset as Early Stopping Set	0.1
Fraction of Used Dataset as Validation Set	0.2
Fraction of Used Dataset as Testing Set	0.15
Restrictions on Evolved Hyperparameters	None

Table 5.1: Parameter values for testing of the final systems.

In addition, the merging subpopulations system has been run with 8 starting populations, and with 152 generations instead of 150. This is done in order to assure the same number of generations between each merge (see Chapter 4.2).

Due to memory limitations on the hardware used, only a fraction of the data have been loaded and used for some of the datasets. Table 5.2 shows the fraction for each dataset. This applies to all tests run. Also, it should be mentioned that the MNIST and fashion-MNIST datasets originally consist of 60,000 training examples and 10,000 testing examples each. Only the training part has been used during these tests for both datasets (this part has again been split into different partitions by the system). These reductions in the number of data examples might result in poorer performance than if the whole sets were used.

Dataset	**Fraction**	**Data Examples**
MNIST	0.2	12,000
EMNIST	0.1	13,160
Fashion-MNIST	0.2	12,000
Chess	1	28,056
Yeast	1	1,484

Table 5.2: Fraction of the different datasets used.

Now, the results will be presented for each dataset. The results for all three systems are presented simultaneously.

MNIST

Table 5.3 presents the results for MNIST for all three systems. Both the accuracy of the best network evolved, and the ensemble accuracy, is shown. The accuracy is the ratio of correctly classified examples on the independent test partition of the dataset. What is considered as the best network is the one with the highest accuracy on the validation set. This means there might be other nets evolved performing better on the testing set, but in order to keep this partition neutral and independent, the systems cannot select the best network based on this. The 10 runs for each system are sorted by descending accuracy for the best net. MSS is used as an abbreviation for the merging subpopulations system. All of this also applies for the tables presenting the results for the other datasets later in this chapter.

	Main System		**MSS**		**Convolutional System**	
	Best Network	**Ensemble**	**Best Network**	**Ensemble**	**Best Network**	**Ensemble**
Run 1	97.8 %	98.1 %	97.6 %	98.1 %	99.0 %	99.2 %
Run 2	97.5 %	97.9 %	97.6 %	97.6 %	99.0 %	99.1 %
Run 3	97.4 %	97.8 %	97.5 %	98.0 %	99.0 %	99.0 %
Run 4	97.2 %	97.7 %	97.4 %	97.9 %	98.9 %	99.4 %
Run 5	97.1 %	98.0 %	97.3 %	98.1 %	98.9 %	99.3 %
Run 6	97.0 %	97.3 %	97.3 %	97.8 %	98.9 %	99.3 %
Run 7	96.9 %	97.5 %	97.2 %	97.8 %	98.9 %	99.3 %
Run 8	96.8 %	97.3 %	97.2 %	97.8 %	98.8 %	99.4 %
Run 9	96.7 %	97.5 %	97.2 %	97.8 %	98.7 %	99.1 %
Run 10	96.7 %	97.4 %	97.0 %	97.9 %	98.6 %	99.0 %
Mean	**97.1 %**	**97.7 %**	**97.3 %**	**97.9 %**	**98.9 %**	**99.2 %**
Median	**97.1 %**	**97.6 %**	**97.3 %**	**97.9 %**	**98.9 %**	**99.3 %**
Std. Dev.	**0.348 %**	**0.277 %**	**0.185 %**	**0.147 %**	**0.127 %**	**0.145 %**
Range	**1.1 %**	**0.8 %**	**0.6 %**	**0.5 %**	**0.4 %**	**0.4 %**

Table 5.3: Results for MNIST.

The results show that the convolutional system outperforms the other two. This is logical, as a convolutional architecture should be able to exploit the image characteristics of a dataset like MNIST in a good way. Also, the merging subpopulations system performs slightly better than the main system on average. Therefore, based on this dataset, it seems like the use of separate populations to ensure diversity is valuable. For 28 out of the 30 runs across all three systems, the ensemble of all individuals in the final population performs better than the best network alone, and for the last two (run 2 with MSS and run 3 with the convolutional system), the best net and ensemble scores equal. That means even such a simple ensemble as the one implemented by the systems (see Chapter 4.1.3) is valuable, at least when it comes to MNIST. The standard deviations and ranges of the results are rather small for all three systems, meaning the systems can be considered quite stable for MNIST, as they are always capable of evolving networks with approximately the same accuracy.

The best performing model overall is found by the ensemble of run 4 with the convolutional system, with an accuracy of 99.4 % correctly classified testing examples. This is not comparable with the state-of-the-art results, where Wan, Zeiler, Zhang, LeCun and Fergus (2013) has achieved an accuracy of 99.79 %. However, this neural network is not evolved, but specifically created for the task at hand. Also, feature engineering and data preprocessing have been available. With evolution, Xie and Yuille (2017) is able to achieve 99.66 % accuracy. That being said, the nets are evolved in a way specializing toward this kind of image classification task, where a lot of previous human knowledge is used by the system. This is also done by the convolutional system presented in this thesis, but not to the same extent (see Chapter 4.3).

To conclude, the convolutional system is capable of evolving quite well-performing nets for the MNIST dataset, but not ones that can challenge state-of-the-art. This is far from surprising, though, as 1) MNIST is one of the most famous datasets in the world, making it very unlikely this system should be able to come up with something on the same level as what many research teams have spent years creating and improving, 2) the nets evolved here have only been allowed to train for 4 minutes each, meaning they could possibly have become better with a longer training time, and 3) only 20 % of the dataset have been used, something which could lead to worse performance. In practice, one would probably not consider using a system like this one for the MNIST dataset, as so many well-performing nets already exist. For a new dataset, on the other hand, where no information about what neural nets work well is available, an automatic system like this one might be of more value.

One of the runs above are selected for further investigation and analysis. This will be done for every dataset. For MNIST, run 1 with the main system is selected. Figure 5.1 presents details about this run.

The average diversity (i.e. PFC value) increases through the first generations, and then decreases again toward the end of the run. It is natural that it increases early on, as the initial population is random, and then the fitness function will reward diversity a lot in early generations, making the population more diverse. Toward the end of the run, diversity will be less and less prioritized, and for the last 25 % of the generations, only accuracy is considered by the fitness function (see Chapter 4.1.2). Therefore, it makes sense that the average diversity drops at this point.

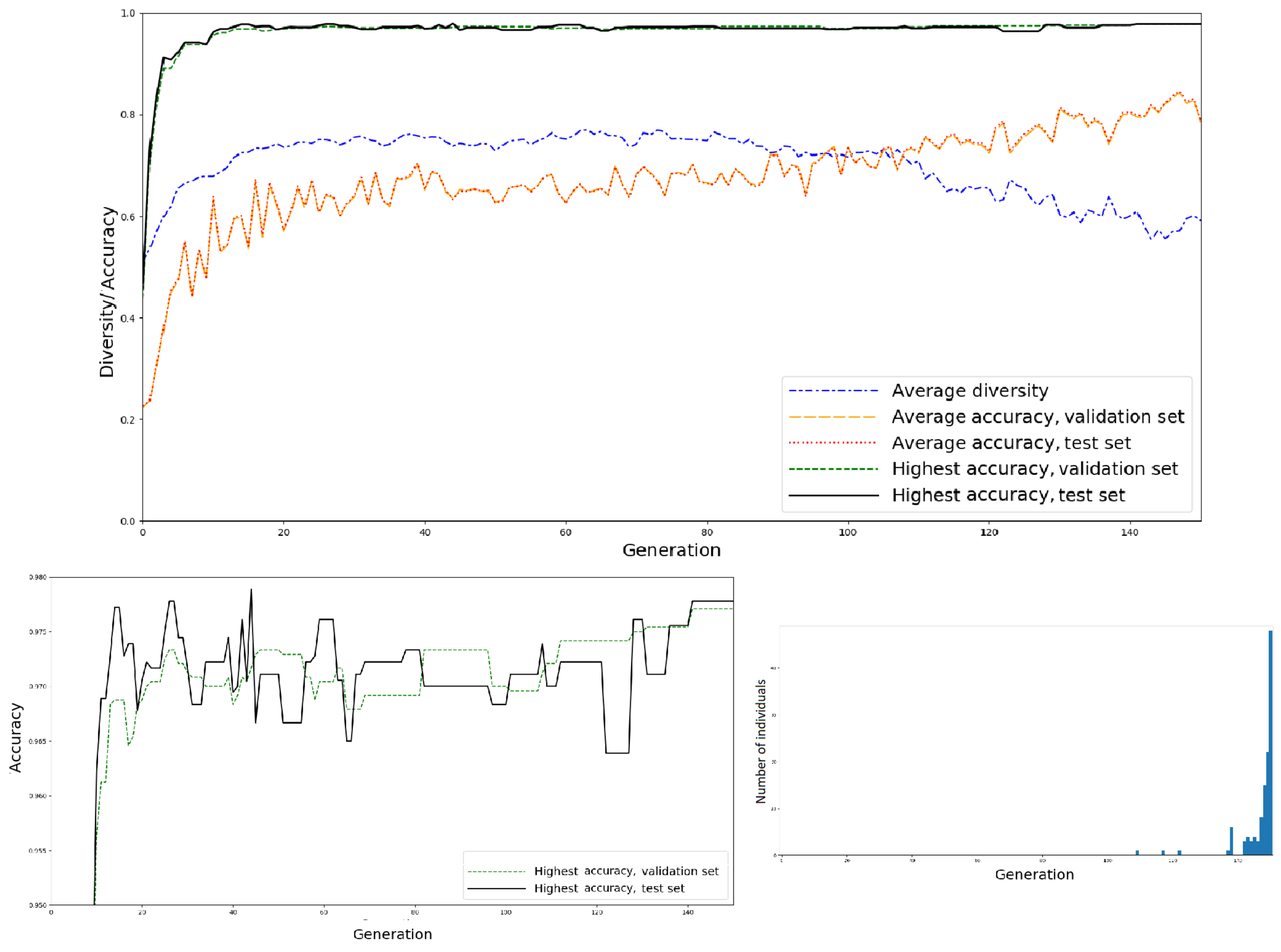


Figure 5.1: A run with the main system on MNIST. The upper graph shows the average diversity measured by PFC, and the average and highest accuracy for the validation and testing sets throughout the generations. Alongside is a magnified graph to easier show the highest accuracy on the validation and test set, and a distribution of what generation the individuals in the final population were created in.

The average accuracies for both the validation and testing set grow throughout the run, as accuracy is continually more prioritized by the fitness function. These two graphs are almost identical. By looking at the magnified version of the graph, it can be seen that the accuracy of the best net on the validation set varies a bit through the generations. In early generations it drops sometimes, as diversity is the main concern for the fitness function. Later on, the validation set accuracy improves a lot, and it reaches its highest value throughout the run just a few generations before the end. This means the best network produced is still alive in the final population.

The accuracy of the best network (where best is defined as performing the best on the validation set) on the testing is more unstable, and sometimes even better than for the validation set (this must be due to randomness). The best individual on the testing set is found already around generation 40, but as it performs poorer on the validation set, it is not considered working so well by the system. This network scores 97.9 % on the testing set, 0.1 % better than the network considered best by the system. A higher correlation between the validation set accuracy and testing set accuracy would reduce this problem. It seems like the cause of the difference is not that the genetic algorithm overfits toward the validation set, but that it is just noise. It does not seem to be much overfitting as the graphs for the average accuracies on the two dataset partitions are almost identical, meaning the evolved nets do not perform better on the validation set on average. With this many nets, the noise is evened out, making a higher correlation than for the graphs of the highest accuracies (which is based on a single net at each time). Also, for the highest accuracy plots of the two set partitions, the testing accuracy is about just as often better than the validation accuracy as the other way around, also suggesting there is no overfitting present. The simplest way of achieving a higher correlation between the two set partitions for each net is to use more data examples, supporting the claim made above that the system might perform slightly worse due to only using a fraction of the dataset available. This in turn means the system performs best with large datasets with lots of examples available, but therefore also requires a lot of system memory.

The average accuracies are much lower than the ones of the best network. Even in the last generation, the average network is only capable of correctly classifying about 75 % of the testing examples, which is a quite horrible performance for MNIST. Of course, this does not really matter as long as the best networks are doing well, but for the ensemble using all individuals in the final population, this could lead to a lower accuracy. Therefore, a more sophisticated ensemble might perform much better, but as it is not a main focus of the thesis (see Chapter 4.1.3), this has not been done. However, this shows the potential of stronger ensembles.

The distribution of when the individuals in the final population were created shows that most individuals are created in the last or close to last generations. It is natural that no individuals from early generations survive, due to aging. However, for the last 25 % of the generations, age is not penalized as only accuracy is considered by the fitness function. Therefore, it could have been possible for more individuals from earlier generations to survive, but it seems like the system is capable of improving all the way until the last generation, making many of the individuals created there surviving instead, as they perform well. This is also supported by the continual increase in average accuracy seen in the main graph.

Table 5.4 describes the three best individuals in the final population of this run. For the topology, only the number of neurons in the hidden layers are shown in the table (i.e. not the input and output layers). All three nets have only one hidden layer, but with a various number of neurons in it. For the activation function, cost function and optimizer, all three nets use the same ones. This is probably partly because these values work very well, but also partially because in the last generations, the population becomes less diverse. That

	Net 1	**Net 2**	**Net 3**
Topology	[305]	[502]	[252]
Activation Function	Swish	Swish	Swish
Cost Function	Cross Entropy	Cross Entropy	Cross Entropy
Optimizer	RMSProp	RMSProp	RMSProp
Learning Rate	0.00773	0.460	0.494
Learning Rate Decay	$7.39 \cdot 10^{-8}$	$6.17 \cdot 10^{-8}$	$7.39 \cdot 10^{-8}$
Dropout Probability	49.4 %	46.0 %	49.4 %
Batch Size	165	165	165
Training Steps	6423	4979	7129
Generation Created	142	150	144
Creation Method	One Parent	One Parent	Two Parents
Validation Set Accuracy	97.7 %	97.7 %	97.7 %
Testing Set Accuracy	97.8 %	97.6 %	97.6 %

Table 5.4: Three best networks for the same run of the main system with MNIST. For this and the upcoming tables, the topology is represented by a list of integers representing the number of neurons in each dense layer.

means that given these values work well, it is likely they will spread, making e.g. Swish be used by a lot of individuals, even though ReLU could work just as well. The fact that Swish is the preferred activation function is interesting, as it is not that well-known, compared to e.g. ReLU and Sigmoid. The learning rate for net 2 and 3 have a very large value, but as it is lowered during training due to the decay, it can be allowed to start at a larger value than if it had been held constant during the training. Nets 1 and 3 have the same dropout probability, and all three nets have the same batch size, suggesting they all have a common ancestor which they have inherited this property from. Also, the net with the largest topology (i.e. net 2) has been able to train for the fewest number of steps, and the one with the smallest (i.e. net 3) has been able to train for the longest, due to the constant training time. All three networks are created within the last 10 generations, suggesting that the system is capable of improving all the way throughout the run.

EMNIST

Table 5.5 shows the results for the 10 runs with EMNIST for each of the three systems. The comments that were presented with the table of MNIST results also applies here, as well as for the other datasets presented later.

As with MNIST, the convolutional system outperforms the other two for EMNIST, as it is an image dataset. However, opposite of with MNIST, here the main system performs better than the merging subpopulations system. This indicates diversity might not be as important for finding well-performing hyperparameter values for EMNIST as it is with MNIST, and that it is better here to use one population which directs the focus toward exploitation. It also means that the choice between a system using only one population, and one using merging subpopulations, is not obvious, as it varies what system performs the best for various datasets.

	Main System		**MSS**		**Convolutional System**	
	Best Network	**Ensemble**	**Best Network**	**Ensemble**	**Best Network**	**Ensemble**
Run 1	81.2 %	82.0 %	81.2 %	80.8 %	86.7 %	89.1 %
Run 2	81.2 %	81.9 %	80.3 %	80.8 %	86.6 %	87.7 %
Run 3	81.0 %	82.0 %	80.0 %	80.5 %	86.5 %	88.4 %
Run 4	81.0 %	81.1 %	79.9 %	80.3 %	86.4 %	89.0 %
Run 5	80.8 %	81.7 %	79.8 %	80.6 %	86.4 %	88.2 %
Run 6	80.7 %	82.0 %	79.4 %	80.3 %	86.2 %	87.3 %
Run 7	80.7 %	81.8 %	79.0 %	80.2 %	86.0 %	87.5 %
Run 8	80.3 %	81.3 %	78.9 %	79.9 %	85.9 %	87.3 %
Run 9	80.2 %	81.6 %	78.7 %	79.9 %	85.9 %	86.9 %
Run 10	80.0 %	81.1 %	78.1 %	79.7 %	85.4 %	86.3 %
Mean	**80.7 %**	**81.7 %**	**79.5 %**	**80.3 %**	**86.2 %**	**87.8 %**
Median	**80.8 %**	**81.8 %**	**79.6 %**	**80.3 %**	**86.3 %**	**87.6 %**
Std. Dev.	**0.399 %**	**0.344 %**	**0.851 %**	**0.363 %**	**0.379 %**	**0.854 %**
Range	**1.2 %**	**0.9 %**	**3.1 %**	**1.1 %**	**1.3 %**	**2.8 %**

Table 5.5: Results for EMNIST.

Just as with MNIST, the ensemble outperforms the best network alone for all three systems. The standard deviations and ranges are a bit higher than for MNIST, meaning the results vary slightly more. An accuracy range of 3.1 % between the best and worst run, as it is for the best network of the merging subpopulations system, might be considered a bit much. One is not guaranteed to get a network performing optimally from a single run. It suggests that it is more challenging to find optimal hyperparameter values for EMNIST than for MNIST, as optimal ones are not being found in every single run. It is slightly surprising that it is the merging subpopulations system that has the highest standard deviation and range, as the separate subpopulations should ensure a more diverse population exploring a larger part of the search space. That being said, this also means the most promising parts of the search space are explored less, and could be what makes this system perform slightly worse and less stable than the main system.

State-of-the-art for the balanced version of EMNIST is found in Jayasundara et al. (2019), with an accuracy of 90.46 %. This is higher than the networks evolved by even the convolutional system here (the best ensemble has 89.1 %), but can be partly explained by the same arguments as those that were discussed when comparing the MNIST results to state-of-the-art.

Run 1 with the convolutional system has been selected for further investigation. Figure 5.2 shows this run. The main difference from what happened in the run with the main system on MNIST described above, is that the networks perform better right from the first generation. This is because the convolutional system does not start with an absolute minimal structure, but with larger, well-performing topologies right from the start (though still pretty close to minimal; see Chapter 4.3). This will be further discussed with the ablation tests related to research question 4 in Chapter 5.2.2. Even though the system performs well right from the first generation, the highest accuracy on the validation set

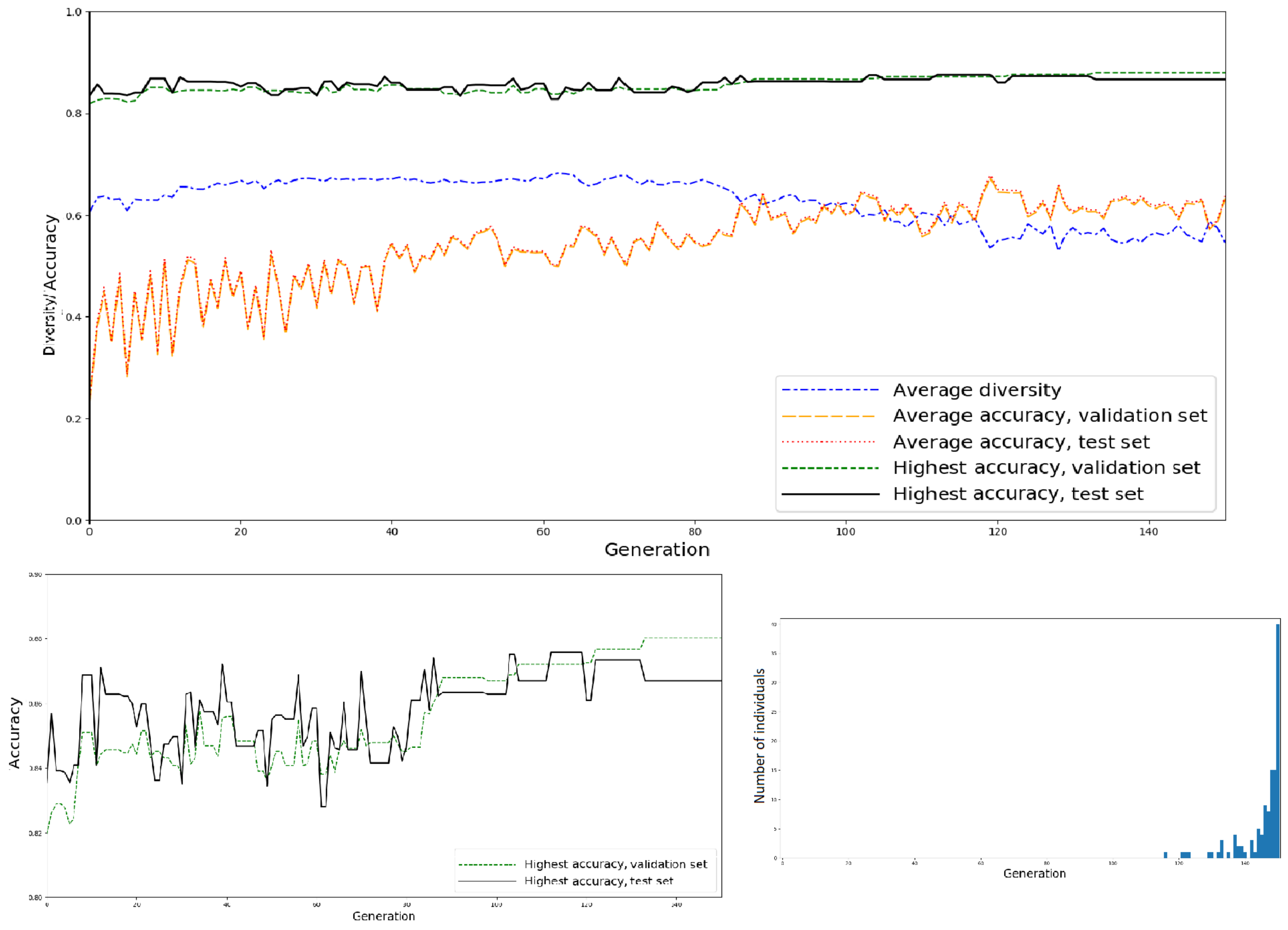


Figure 5.2: A run with the convolutional system on EMNIST. The upper graph shows the average diversity measured by PFC, and the average and highest accuracy for the validation and testing sets throughout the generations. Alongside is a magnified graph to easier show the highest accuracy on the validation and test set, and a distribution of what generation the individuals in the final population were created in.

grows throughout the run, until the best net is found in generation 129. Unluckily, this particular network performs worse on the testing set than some of the previous best networks do. The best network on the testing set, found on the graph around generation 115, has an accuracy of 87.6 %, 0.9 % better than the net which the system claims is the best one. This further emphasizes the suggestion that larger datasets are needed to increase the correlation between the accuracy of the validation and testing sets, as discussed with the MNIST results. Also here, the graphs showing average accuracies for the two set partitions are almost identical, meaning there is close to no overfitting by the genetic algorithm toward the validation set.

	Net 1	**Net 2**	**Net 3**
Topology	Conv(28, 3, 1, true) → Pool(3) → Conv(56, 3, 1, true) → Pool(3) → Conv(161, 15, 1, true) → Pool(3) → [406, 477]	Conv(16, 3, 1, true) → Pool(3) → Conv(56, 3, 1, true) → Pool(3) → Conv(171, 3, 1, true) → Pool(3) → [237, 271]	Conv(16, 3, 1, true) → Pool(3) → Conv(56, 3, 1, true) → Pool(3) → Conv(171, 3, 1, true) → Pool(3) → [237, 266]
Activation Function	Swish	Swish	Swish
Cost Function	Cross Entropy	Cross Entropy	Cross Entropy
Optimizer	RMSProp	Adam	Adam
Learning Rate	0.00746	0.00746	0.00315
Learning Rate Decay	$8.13 \cdot 10^{-7}$	$1.13 \cdot 10^{-6}$	$1.10 \cdot 10^{-6}$
Dropout Probability	13.9 %	13.9 %	13.9 %
Batch Size	16	21	24
Training Steps	2767	2729	2549
Generation Created	129	147	144
Creation Method	One Parent	Two Parents	One Parent
Validation Set Accuracy	88.0 %	88.0 %	87.9 %
Testing Set Accuracy	86.7 %	87.1 %	87.0 %

Table 5.6: Three best networks for the same run of the convolutional system with EMNIST. The syntax for presenting the topology is as follows: Conv(a, b, c, d) is a convolutional layer with a filters of size b, with stride c, and with zero-padding if d is true. Pool(e) is a max pooling layer with size e, while the dense layers are represented in the same way as before.

Table 5.6 lists the properties of the three best networks in the final population. All three networks contain 3 pooling layers with only 1 convolutional layer in between, followed by 2 hidden dense layers. Zero-padding is always preferred, and the same applies for a filter size of 3 (except for one layer with size 15), a stride of 1, and a pooling size of 3. Some of the layers are identical for more than one of the networks, suggesting a common ancestor. It is interesting that just as with MNIST, the Swish activation function is preferred. Both RMSProp and Adam seem to be well-performing optimizers for EMNIST, as they are both present in the top 3 networks. The batch sizes are quite low for all three nets. The fact that the convolutional networks have a more complex topology than the single hidden layer networks used in the MNIST run analyzed above, results in a much lower number of training steps, as each one takes up more time. This might also partially be the cause for the lower batch sizes; if they were too large, it might not have been enough time (i.e. training steps) to train the nets properly. Opposed to MNIST, the networks perform considerably better on the validation set than on the testing set. However, this does not necessarily indicate that the system overfits a lot toward the validation set; as the three nets with the highest validation set accuracy are investigated, it is natural that they perform especially well on this partition. There might be just as many individuals in the population which do better on the testing set. This is also supported by the fact that the average accuracies are approximately equal for the two partitions, as discussed above.

Fashion-MNIST

Table 5.7 presents the results for fashion-MNIST, in the same manner as the results for MNIST and EMNIST were presented.

	Main System		**MSS**		**Convolutional System**	
	Best Network	**Ensemble**	**Best Network**	**Ensemble**	**Best Network**	**Ensemble**
Run 1	87.7 %	88.3 %	88.2 %	88.7 %	91.2 %	92.0 %
Run 2	87.7 %	88.0 %	88.1 %	88.6 %	91.1 %	91.8 %
Run 3	87.6 %	87.8 %	88.1 %	88.3 %	91.0 %	91.7 %
Run 4	87.5 %	88.3 %	88.0 %	88.7 %	90.9 %	91.9 %
Run 5	87.5 %	88.2 %	87.9 %	88.2 %	90.7 %	91.6 %
Run 6	87.5 %	87.9 %	87.9 %	88.2 %	90.6 %	91.4 %
Run 7	87.4 %	88.1 %	87.8 %	88.6 %	90.5 %	91.5 %
Run 8	87.3 %	88.1 %	87.8 %	88.4 %	90.4 %	91.4 %
Run 9	87.3 %	88.1 %	87.6 %	88.5 %	90.3 %	91.7 %
Run 10	87.0 %	88.0 %	87.6 %	88.3 %	90.3 %	91.5 %
Mean	**87.5 %**	**88.1 %**	**87.9 %**	**88.5 %**	**90.7 %**	**91.7 %**
Median	**87.5 %**	**88.1 %**	**87.9 %**	**88.5 %**	**90.7 %**	**91.7 %**
Std. Dev.	**0.201 %**	**0.154 %**	**0.195 %**	**0.186 %**	**0.316 %**	**0.196 %**
Range	**0.7 %**	**0.5 %**	**0.6 %**	**0.5 %**	**0.9 %**	**0.6 %**

Table 5.7: Results for fashion-MNIST.

Just as for the other two image datasets, the convolutional system outperforms the ones evolving dense nets, and the ensembles outperform the single best networks for all systems. Just as with MNIST, the merging subpopulations system do slightly better than the main system. There is a low standard deviation for all systems, and the range between the best and worst run are below 1 % for all as well. This suggests the systems are capable of always finding quite well-performing solutions for fashion-MNIST. State-of-the-art for fashion-MNIST is reported in Zhong, Zheng, Kang, Li and Yang (2017), with an accuracy of 96.35 %. The systems are not competitive with this, probably for the same reasons as those discussed with the MNIST results.

For this dataset, run 5 of the merging subpopulations system has been selected for further investigation. Figure 5.3 presents this run. As the merging subpopulations system contain several populations, the averages and highest values are calculated across all populations.

The same tendencies can be seen as for the other datasets. The diversity drops slightly during the run, the accuracies increase in general, and most of the individuals in the final population are created in the last couple of generations.

The merging from first 8 to 4 populations, then 4 to 2, and finally 2 to a single population, is done after generation 38, 76 and 114. It is hard to spot such a critical action by studying the graphs (at least except for the final merge, discussed next); nothing particular seems to happen at those points. This can be considered an advantage as the

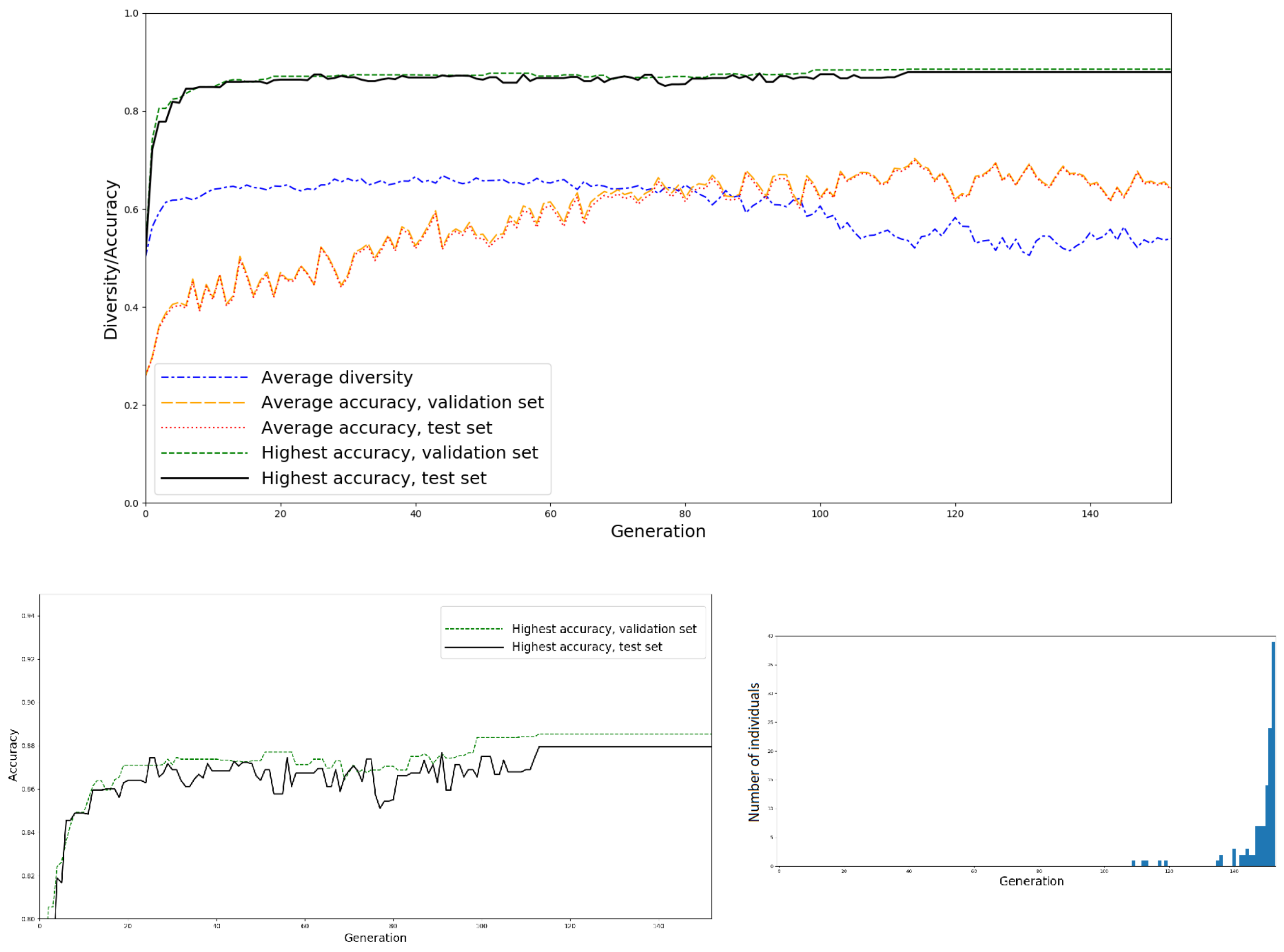


Figure 5.3: A run with the merging subpopulations system on fashion-MNIST. The upper graph shows the average diversity measured by PFC, and the average and highest accuracy for the validation and testing sets throughout the generations. Alongside is a magnified graph to easier show the highest accuracy on the validation and test set, and a distribution of what generation the individuals in the final population were created in.

progression keeps running smoothly, instead of the whole dynamics of the diversities and accuracies of the population changing by a lot at these points. The exception is that the highest accuracies vary a lot for the first 113 generations, but at that point it stabilizes as the best individual is found. The fact that this happens just before the final merge might be coincidental, but it could also be that when left with a single population, the system becomes slightly less innovative and has more trouble inventing new, even better networks. This claim is strengthened by the fact that the main system, which always has only a single population, performs slightly worse than the merging subpopulations system for the fashion-MNIST dataset in general (presented above). It has been investigated if this was the case also for the other runs with the merging subpopulations system, and with the other datasets, and it was a general tendency. That being said, it was particularly clear for this specific run, and also more for the fashion-MNIST dataset than for the other datasets.

	Net 1	**Net 2**	**Net 3**
Topology	[263, 155, 16]	[263, 155, 287, 16, 44, 85]	[124, 263, 155, 17]
Activation Function	Swish	Swish	Swish
Cost Function	Huber	Cross Entropy	Cross Entropy
Optimizer	RMSProp	RMSProp	Adam
Learning Rate	0.00203	0.00203	0.00203
Learning Rate Decay	$2.24 \cdot 10^{-10}$	$2.24 \cdot 10^{-10}$	$2.24 \cdot 10^{-10}$
Dropout Probability	19.2 %	19.2 %	24.1 %
Batch Size	122	86	128
Training Steps	6355	5739	5775
Generation Created	113	109	152
Creation Method	Two Parents	Two Parents	One Parent
Validation Set Accuracy	88.5 %	88.4 %	88.4 %
Testing Set Accuracy	87.9 %	87.0 %	87.7 %

Table 5.8: Three best networks for the same run of the merging subpopulations system with fashion-MNIST.

Table 5.8 shows the three best networks evolved by the merging subpopulations system for this run. All networks prefer rather large topologies, ranging from 3 to 6 hidden layers. Once again, Swish is the preferred activation function, beating the quite similar but much more popular ReLU. Cross Entropy is a popular cost function (just as with MNIST and EMNIST), but the best network uses Huber instead. RMSProp and Adam both seem to work well as optimizers, a tendency continued from the MNIST and EMNIST datasets. All three nets have the same learning rate and decay, most likely from the same ancestor. The fact that these values often are equal for the best networks suggest they might be troublesome to set, meaning most mutations to these values create poorly performing nets. When a value working well first is found, it is quickly spread to most of the population, as it outperforms other values. The two best individuals are created quite early on, in generations 113 and 109, but the system is still capable of evolving well-performing nets until the very last generation, as network 3 shows.

Chess

Table 5.9 shows the results with the chess dataset. They are presented in the same way as with the tables above for the other datasets.

Contrary to the image datasets, the convolutional system is not able to compete with the two others here. With an accuracy of just below 50 %, the convolutional networks evolved are capable of extracting a lot of information from the features, as it performs much better than random guessing, but it is still much worse than what the dense nets can do. Figure 5.4 shows how the chess board has been converted to an image-like structure which can be used by the convolutional system. Interpreting a chess board as an image is very interesting,

	Main System		MSS		Convolutional System	
	Best Network	**Ensemble**	**Best Network**	**Ensemble**	**Best Network**	**Ensemble**
Run 1	90.4 %	91.8 %	89.7 %	91.7 %	46.9 %	50.3 %
Run 2	90.3 %	92.0 %	89.5 %	91.7 %	46.4 %	50.2 %
Run 3	90.3 %	91.8 %	89.5 %	91.5 %	46.1 %	49.9 %
Run 4	90.2 %	91.9 %	89.4 %	91.4 %	46.0 %	50.1 %
Run 5	90.1 %	91.7 %	89.2 %	91.7 %	46.0 %	50.0 %
Run 6	90.0 %	91.5 %	89.1 %	91.4 %	45.9 %	49.2 %
Run 7	89.9 %	92.0 %	89.0 %	91.2 %	45.8 %	50.1 %
Run 8	89.8 %	92.3 %	88.8 %	91.1 %	45.1 %	49.7 %
Run 9	89.7 %	91.9 %	88.5 %	90.7 %	44.5 %	49.8 %
Run 10	89.5 %	91.6 %	88.3 %	91.0 %	44.2 %	49.0 %
Mean	**90.0 %**	**91.9 %**	**89.2 %**	**91.3 %**	**45.7 %**	**49.8 %**
Median	**90.1 %**	**91.9 %**	**89.2 %**	**91.4 %**	**46.0 %**	**50.0 %**
Std. Dev.	**0.279 %**	**0.216 %**	**0.433 %**	**0.320 %**	**0.798 %**	**0.405 %**
Range	**0.9 %**	**0.8 %**	**1.4 %**	**1.0 %**	**2.7 %**	**1.3 %**

Table 5.9: Results for the chess dataset.

but is of course not the main task for a convolutional network. Dense networks are generally more flexible when it comes to the usage of its neurons and connections, as none of them share weights, and it turns out this is well-exploited when it comes to the chess dataset.

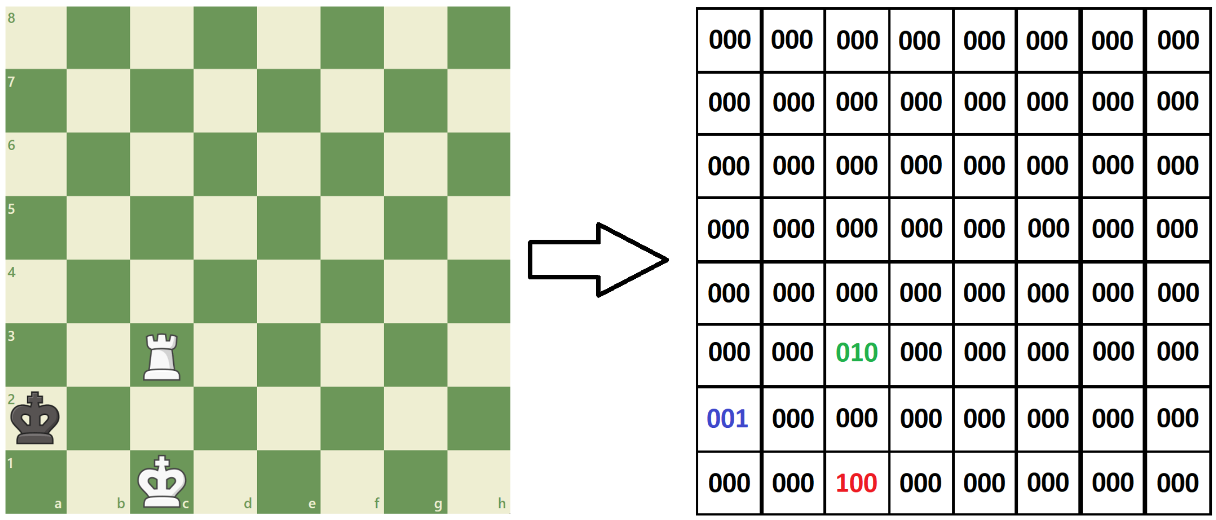


Figure 5.4: An example chess position turned into an image representation. Each square is converted to a pixel, each with three color channels. If a square is occupied by the white king the first channel gets value 1, if it is occupied by the white rook the second channel gets value 1, and if it is occupied by the black king the third channel gets value 1.

The main system performs clearly better than the merging subpopulations system, just as with EMNIST, but opposite of MNIST and fashion-MNIST. It is starting to become clear that there is not a clear winner when it comes to the choice between the main system and merging subpopulations system, as so far, they perform best on two datasets each. It is also hard to find any general guidelines to predict what system works the best for a particular dataset (e.g. not all image datasets prefer the same, and MNIST and EMNIST, probably the two most similar datasets, work the best on different systems). This is unfortunate as it would be easier for a user if one system always worked the best, but that simply does not seem to be how it is.

The ensembles outperform the best networks alone, just as for every other dataset. The standard deviation and ranges are largest for the convolutional system, but is not really comparable to the other systems as it performs so much worse. For the other systems, well-performing networks are being evolved fairly stable from run to run, but with some minor variations (with a range about 1 %). This dataset is not as famous as the other ones, resulting in that no comparable state-of-the-art results have been found.

Run 1 with the main system will now be studied further. Figure 5.5 shows details about this run. Both the average and highest accuracies grow a lot in the first 25 generations, and the highest ones keep increasing for the rest of the run. Also, there is a higher correlation between the validation and testing accuracies than for the previous datasets. The main reason for this is most likely that there are many more data examples used, meaning each partition will have more data and be subject to less noise.

	Net 1	**Net 2**	**Net 3**
Topology	[412, 197, 393, 302, 87]	[443, 197, 440, 178, 125]	[611, 337, 221, 178, 87]
Activation Function	Swish	Swish	Swish
Cost Function	Cross Entropy	Cross Entropy	Cross Entropy
Optimizer	Adam	Adam	Adam
Learning Rate	0.00316	0.00271	0.00812
Learning Rate Decay	$1.16 \cdot 10^{-6}$	$1.16 \cdot 10^{-6}$	$1.79 \cdot 10^{-6}$
Dropout Probability	0 %	0 %	0 %
Batch Size	333	431	431
Training Steps	1881	1516	1470
Generation Created	127	134	132
Creation Method	One Parent	Two Parents	Two Parents
Validation Set Accuracy	90.0 %	90.0 %	89.9 %
Testing Set Accuracy	90.4 %	89.9 %	89.9 %

Table 5.10: Three best networks for the same run of the main system with the chess dataset.

Table 5.10 presents the three best networks evolved in this run. Very large topologies are preferred, all with 5 hidden layers. At some point it should stop being a surprise that Swish is the activation function of the best individuals, as not a single other activation function has been used by any of the three best nets for the selected runs of any of the datasets presented so far. The dropout probability is 0 % for all networks, and this is also a general tendency for every run with the chess dataset. It seems like these nets are formed in a way that makes them dependent on the value of every neuron in order to make some sense out of it, and be capable of performing well. The batch sizes are very large. Such large values would most

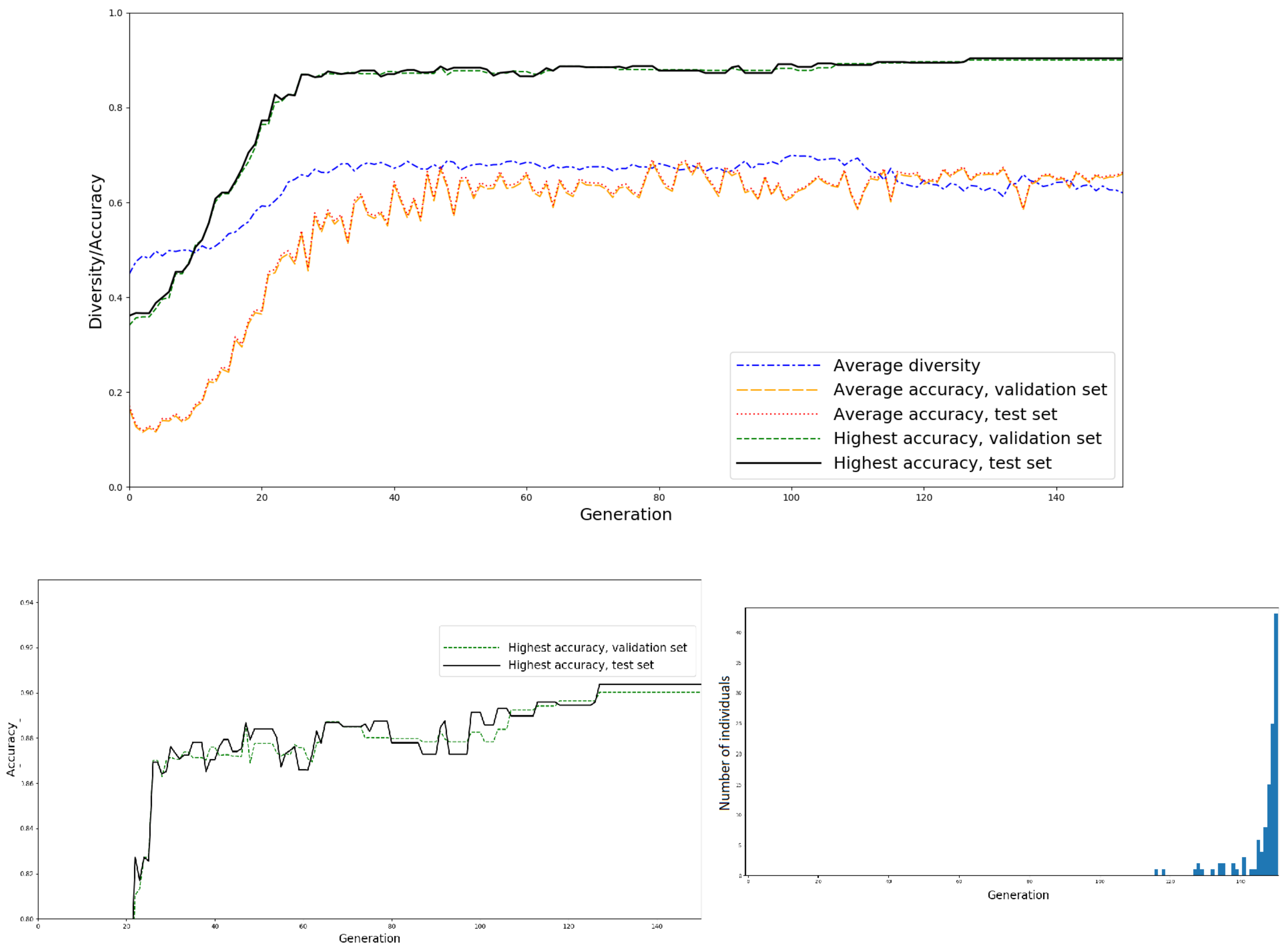


Figure 5.5: A run with the main system on the chess dataset. The upper graph shows the average diversity measured by PFC, and the average and highest accuracy for the validation and testing sets throughout the generations. Alongside is a magnified graph to easier show the highest accuracy on the validation and test set, and a distribution of what generation the individuals in the final population were created in.

likely not have been tried by a human, showing the value of evolution. The nets are trained for a relatively small number of steps, as there are large topologies and large batch sizes, but the results show it is enough for the weights to be adjusted into well-performing states. The best network actually performs 0.4 % better on the testing set than on the validation set. This must be due to randomness, and can be regarded as quite lucky when it comes to the performance of the system run. It shows that even though the increase in number of data examples increases the correlation between the validation and testing accuracies (as discussed above), there is still some variance between them.

Yeast

Table 5.11 presents the results of the runs with the yeast dataset. Because the convolutional system is not compatible with this dataset, as the features cannot be interpreted as images in any meaningful way, only results for the main system and merging subpopulations system are available.

	Main System		**MSS**	
	Best Network	**Ensemble**	**Best Network**	**Ensemble**
Run 1	63.1 %	62.6 %	64.9 %	63.1 %
Run 2	61.7 %	60.4 %	64.4 %	64.0 %
Run 3	59.9 %	60.4 %	62.6 %	63.5 %
Run 4	59.9 %	56.3 %	60.4 %	61.7 %
Run 5	59.0 %	58.1 %	59.9 %	60.8 %
Run 6	59.0 %	57.7 %	59.9 %	59.9 %
Run 7	58.6 %	58.6 %	58.6 %	61.7 %
Run 8	56.3 %	58.1 %	58.1 %	63.5 %
Run 9	55.4 %	57.7 %	56.8 %	55.4 %
Run 10	53.6 %	52.3 %	55.0 %	55.9 %
Mean	**58.7 %**	**58.2 %**	**60.1 %**	**61.0 %**
Median	**59.0 %**	**58.1 %**	**59.9 %**	**61.7 %**
Std. Dev.	**2.72 %**	**2.61 %**	**3.02 %**	**2.92 %**
Range	**9.5 %**	**10.3 %**	**9.9 %**	**7.2 %**

Table 5.11: Results for the yeast dataset.

These results are significantly different than those for the other datasets. Firstly, the standard deviations and ranges are about 10 times larger than for the other datasets, meaning there is no guarantee a single run will be able to discover a well-performing net. The merging subpopulations system both performs slightly better than the main system, and has a slightly smaller variance, probably because separate populations can help exploring more types of individuals, some of which perform better. Secondly, the ensemble is worse than the single best net for the main system on average, and just barely better for the merging subpopulations system. The ensemble probably does not perform so well because as there are more poorly performing individuals in the population, these will sabotage for the better ones in the ensemble.

The main cause for the problems seems to be the low correlation between a high validation set accuracy and a high testing set accuracy (this can be seen in the results for the specific run presented below). Even though much better networks for the test set might have been evolved, the ones evaluated as well-performing by the systems (because they have a high validation accuracy) are not necessarily the best on the independent test set. This again is caused by the fact that the dataset has only 1,484 examples. With a validation fraction of 20 % and a testing fraction of 15 %, this only results in 296 validation examples and 222 testing examples, making more noise to the results and less correlation between the two accuracies. This indicates what has already been discussed for the other datasets; that more data examples are required for the systems to perform well and stable.

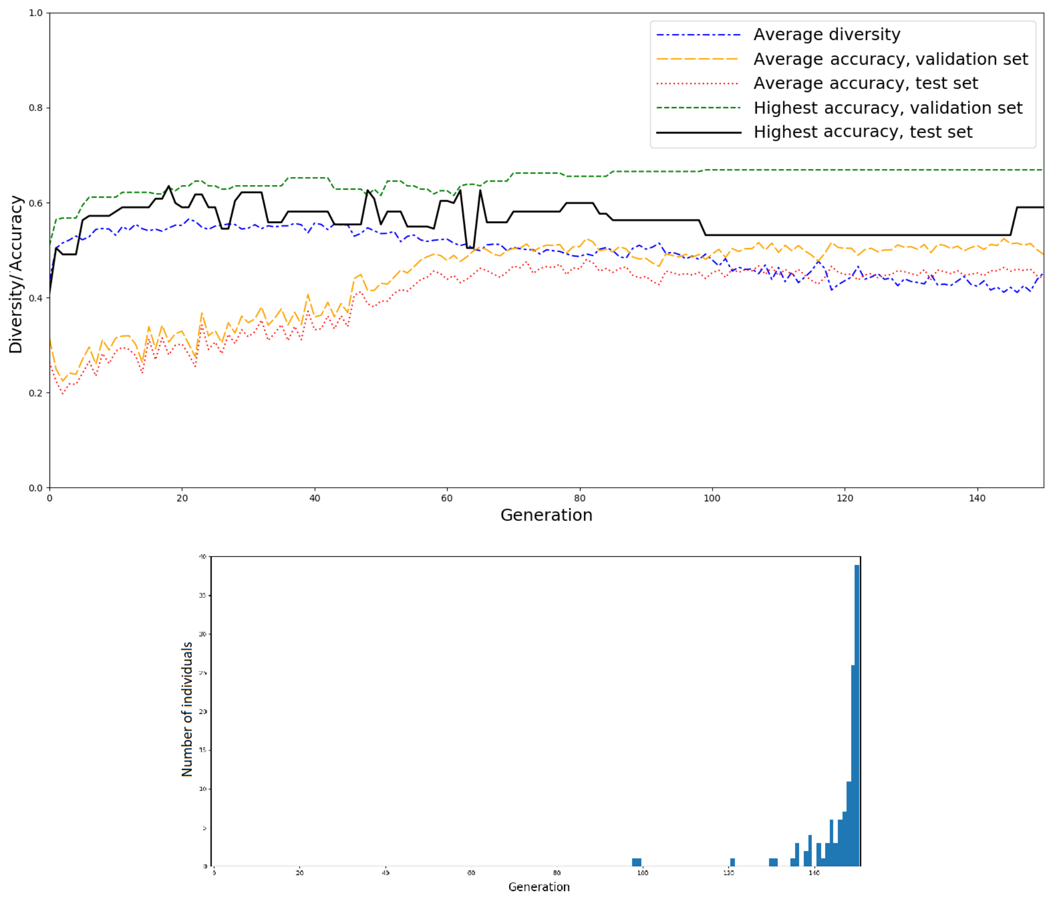


Figure 5.6: A run with the main system on the yeast dataset. The upper graph shows the average diversity measured by PFC, and the average and highest accuracy for the validation and testing sets throughout the generations. Alongside is a distribution of what generation the individuals in the final population were created in.

Run 5 of the main system is selected for further analysis. Figure 5.6 shows details about this run. The system only uses a few generations to find decent performing nets, but do not improve much from there on. For the average accuracies, one can see a clear correlation between the validation and testing set, but for a single network (i.e. the best network), there is little correlation between the validation and testing accuracies. Also, one can clearly see that the system overfits to the validation set for this dataset, as the average validation accuracy is constantly higher than the test set accuracy. The main reason for this could just be the lack of correlation, as the system will preserve more nets with higher validation accuracy, as that is what the fitness function considers.

Table 5.12 presents details about the three best networks of this run. All nets have many layers, but with only a few neurons in each. Contrary to all other runs investigated

	Net 1	**Net 2**	**Net 3**
Topology	[6, 7, 21, 5, 32, 20, 2]	[7, 2, 14, 27, 4, 2]	[9, 5, 38, 5, 87, 20, 2]
Activation Function	Softplus	Softplus	Softplus
Cost Function	Cross Entropy	Cross Entropy	Cross Entropy
Optimizer	Adam	RMSProp	Adam
Learning Rate	0.102	0.0422	0.0422
Learning Rate Decay	$5.29 \cdot 10^{-6}$	$1.53 \cdot 10^{-10}$	$5.29 \cdot 10^{-6}$
Dropout Probability	2.79 %	2.79 %	2.79 %
Batch Size	95	116	207
Training Steps	11200	27400	10900
Generation Created	146	98	146
Creation Method	One Parent	Two Parents	Two Parents
Validation Set Accuracy	66.9 %	66.9 %	66.9 %
Testing Set Accuracy	59.0 %	53.2 %	56.3 %

Table 5.12: Three best networks for the same run of the main system with the yeast dataset.

for the other datasets, Softplus is preferred as activation function instead of Swish. Cross Entropy keeps performing well, and so do Adam and RMSProp. These have generally been preferred by the other datasets as well. Dropout is used by all three networks, though the rate is so small it is almost negligible. The networks can train for very many steps, because the topologies are relatively small, and the dataset also contains few features (reducing the size of the input layer). For these best networks, the difference between the accuracy on the validation and training sets are extreme, with as much as 13.7 % difference for net 2. It is natural that the difference is particularly high for these nets, as those with the most extreme (i.e. high) validation accuracies are picked. That being said, it clearly illustrates the problem.

General Remarks

For all datasets, reasonably well-performing neural networks have been found. For the three popular image datasets which can easily be compared to other works, the nets evolved are not capable of challenging state-of-the-art, but still do decently well. The systems are relatively stable at consistently being capable of finding well-performing networks, except when it comes to the yeast dataset. This is probably mostly caused by the fact that this dataset consists of too few data examples, adding too much noise and too little correlation between the different dataset partitions. Also for the other datasets, more data examples and an even higher correlation would help the results. As only a fraction of the image datasets are used, this could easily be achieved as long as one has the computational resources available. The systems therefore seem to require at least a few thousand data examples in order to perform well, and probably about 50,000+ to work optimally, though this will of course vary from dataset to dataset.

The convolutional system performs the best on all three image datasets, making the choice of preferred system easy for future image datasets. On the other hand, when it comes to the choice between the main system and merging subpopulations system, it is not easy to predict which one does better. The main system performs best for the EMNIST and chess datasets, while the merging subpopulations system performs best on the other three datasets. The systems must be tested on many more datasets in order to figure out which of these systems works the best in general, or to find some general rules which can predict what works best for the dataset currently at hand.

By looking at the three best nets of each of the runs studied in depth, it is clear that a lot of different hyperparameter values can work well, depending on the problem at hand. This includes simple and complex topologies, and small and large learning rates, learning rate decays, dropout rates and batch sizes. The biggest surprise is probably Swish's dominance when it comes to the activation function, selected by 12 out of the 15 studied nets. Also, Cross Entropy was generally the most attractive cost function, while both RMSProp and Adam were frequently used optimizers by the best nets. Both nets created from a single parent and from two parents were found among the top three nets, but no random immigrants were. This is very natural, as it is very unlikely a completely random individual will beat the ones carefully evolved over time. Too many conclusions should not be made based on only these few networks, but these general tendencies also seem to apply for the other best nets evolved in the different runs, which have not been presented in depth.

Research question 1 concerns to what degree these kinds of systems are capable of evolving well-performing neural nets, and based on these tests, they are able to do so fairly well. They might not be comparable with state-of-the-art, which comes from methods that have required a lot of research and trial, but for a new dataset where one has no real guidelines to what kind of hyperparameter values will perform well, evolving networks in this manner might be of value. A final answer to research question 1 will not be given yet, as some of the results presented in the next two subchapters also will be used for this.

5.2.2 Ablation Tests

As explained above, an ablation test is to remove a component from the system to see how this changes the performance. If the system does just as well or better without the component, it could be considered irrelevant or detrimental. The idea of conducting such tests was inspired by Stanley and Miikkulainen (2002).

The ablations have been divided into four parts, one for each research question:

1. Ablation tests without evolution, in order to answer research question 1 partially
2. Ablation tests where hyperparameter evolution has been replaced by manually setting specific hyperparameters, in order to answer research question 2
3. Ablation tests related to diversity, in order to answer research question 3
4. Ablation tests related to incremental complexification, in order to answer research question 4

In total, this testing phase has consisted of 208 test runs, each running for about 15 hours. The phase has only been run on the main system, due to time and hardware limitations. In order to compare the ablation systems to the actual system, some base tests have been conducted. These are similar to the final tests of the main system described in Chapter 5.2.1, but have been run for a shorter period of time to be comparable to the ablation tests. These base tests have been run with the parameter values shown in Table 5.13.

Parameter	**Value**
Generations	80
Population Size	80
Children per Generation	80
Training Time per Individual	120 seconds
Validation Interval	100 steps
Patience	10000 steps
PFC Sample Rate	1
PFC Individuals to Compare With	all
Crossover Rate	0.5
Random Immigrant Rate	0.1
Age Factor	2
Final Generations with Accuracy Only Ratio	0.25
Fraction of Dataset Used in Total	See Table 5.2
Fraction of Used Dataset as Early Stopping Set	0.1
Fraction of Used Dataset as Validation Set	0.2
Fraction of Used Dataset as Testing Set	0.15
Restrictions on Evolved Hyperparameters	None

Table 5.13: Parameter values for the base tests.

The parameter values of Table 5.13 have also been used for every ablation test, if not anything else is specified. The fractions of the datasets used are similar to those used when testing the final systems, shown in Table 5.2 (Chapter 5.2.1).

As discussed in Chapter 5.2.1, the results of the yeast dataset were quite noisy. Due to this, and the fact that only a limited number of tests could be run, it has not been used during this phase. Therefore, each test has been run for each of the 4 other datasets, and usually 2 times per dataset. 2 runs are a bit marginal to give the individual results a very high statistical significance, but the resources to conduct more tests have not been available. However, as there are 4 datasets, each type of test has been run for a total of 8 times across all of these, giving some more statistical significance overall. Furthermore, some of the most important tests, like the ones used to investigate diversity and incremental complexification of research questions 3 and 4, have been run more times. This will be described when these results are presented, but if not otherwise specified, the number of runs per test is 2.

As there are a lot of results which will be discussed in this subchapter, a simple measure which can be used to evaluate performance quickly is needed. The average accuracies on the test set across all datasets have been chosen for this. Table 5.14 presents the results of the base tests. Every table in this subchapter will present the results in this way if not otherwise specified, with one value followed by another in parentheses. The first one represents the average accuracy on the test set of the best network evolved across all datasets, and the one in parentheses is the average accuracy on the test set of the ensemble of all networks in the final population across all datasets. Each base test has been run 6 times.

Dataset	**Average Accuracy**
MNIST	96.6 % (97.3 %)
EMNIST	79.1 % (80.6 %)
Fashion-MNIST	87.3 % (87.9 %)
Chess	88.2 % (91.1 %)
Overall Average	**87.8 % (89.2 %)**

Table 5.14: Results for the base tests. The first value represents the average accuracy on the test set for the best net, while the one in parentheses represents the average accuracy on the test set by the ensemble of all individuals in the final population.

The four ablation parts regarding each research question will be presented next.

Ablations Related to Research Question 1

The first research question concerns to what degree a system like this one can work well in general, and has already been partially answered with the results of Chapter 5.2.1. Here, only one additional ablation test is run in order to prove that evolution has a positive effect on finding well-performing neural networks. This is done by running a test which uses random search instead of evolution to create new networks. If the base system using evolution performs better than random search, evolution has a positive effect, but if random search performs better or just as good, evolution might not be the way to go for finding well-performing neural nets.

The main system implemented can easily be transformed into a random search system by letting all children be created as random immigrants (i.e. a random immigrant rate of 1). This way, no genetic information is passed from one individual to another, hence making all individuals being created randomly and independently of each other. The only problem by doing this directly, is that the topology of the individuals will stay minimal and never grow, due to the way random immigrants are created (see Chapter 4.1.2). Instead of letting the number of layers of a random immigrant be drawn randomly between the least and most numbers of layers in the population, and the number of neurons in the same manner (as they are in the standard main system), they are set as follows:

$$l = DU([max(l_{min} - 1, 1),\ min(l_{max} + 1, 5)])$$
$$n = DU([max(floor(\frac{n_{min}}{2}), 1),\ min(2n_{max}, 2500)])$$

Here, l is the number of layers, with l_{min} and l_{max} being the minimum and maximum number of layers of any individual currently in the population. n, n_{min} and n_{max} are the corresponding for the number of neurons in a layer. DU is a discrete uniform distribution.

This means the method of creating new individuals is not entirely independent of the other individuals in the population, but is still close to being so. A maximum number of 5 layers and 2500 neurons per layer are set to stop individuals from growing indefinitely large and using all memory and storage space on the hardware.

Table 5.15 shows the results of the random search, compared to the results of the base tests. As can be seen, evolution performs better for all datasets. The difference varies (e.g. the difference on MNIST is much less than for the chess dataset), and random search is able to find some decent neural nets, but the results show evolution does help finding well-performing networks.

Dataset	**Random Search**	**Evolution (base tests)**
MNIST	96.1 % (96.3 %)	96.6 % (97.3 %)
EMNIST	76.0 % (76.2 %)	79.1 % (80.6 %)
Fashion-MNIST	86.8 % (86.6 %)	87.3 % (87.9 %)
Chess	81.2 % (74.0 %)	88.2 % (91.1 %)
Overall Average	**85.4 % (83.2 %)**	**87.8 % (89.2 %)**

Table 5.15: Results for random search. The first value represents the average accuracy on the test set for the best net, while the one in parentheses represents the average accuracy on the test set by the ensemble of all individuals in the final population.

Another interesting property of these results is that the ensembles of random search either perform worse than the single best network, or at least improve the results less than what they do with evolution. It is natural that a population of random individuals does not perform as well on average, hence working worse together than those created by evolution.

Ablations Related to Research Question 2

Research question 2 concerns what hyperparameters are most critical to evolve for the neural networks to perform well. A hyperparameter can be considered more critical to evolve if a change in its value drastically changes the performance of the network, and if only a small fraction of its possible values make the network perform well. To test this by using the main system implemented, it has been tried to manually set the values of one hyperparameter at a time. If the neural networks evolving all hyperparameters but the one currently investigated performs as well as the neural networks where all hyperparameters are evolved, that means it is most likely easy to set the value of that hyperparameter, hence making it less critical.

The values set manually are based on some general rules for setting hyperparameters, as well as some arbitrariness. Of course, the results of other tests conducted during the thesis have not been used as a basis for selected hyperparameter values, as that would defeat the whole purpose. Table 5.16 shows the values that were tried for the different hyperparameters, one at a time. The only two hyperparameters set manually at the same time are the learning rate and learning rate decay, as they concern the same thing.

Hyperparameter	**Values**
Topology	Small, Medium and Large
Activation Function	Sigmoid and ReLU
Cost Function	MSE and Cross Entropy
Optimizer	Adam
Learning Rate and Learning Rate Decay	(0.01, 0), (0.1, 0) and (0.1, 10^{-7})
Dropout Probability	0 %, 50 % and 70 %
Batch Size	32, 64 and 128

Table 5.16: Manually set values for the different hyperparameters. For the learning rate and learning rate decay, the first value of the tuple represents the learning rate, and the second value represents the learning rate decay.

When it comes to the topologies, the same ones have not been used for every dataset. Instead, it has been attempted to set topologies that can be regarded as small, medium and large for the different datasets. E.g. what is considered small for MNIST might be considered large for the chess dataset, as there are so many more features for MNIST. Table 5.17 shows the selected topologies for the different datasets.

Dataset	**Small Topology**	**Medium Topology**	**Large Topology**
MNIST	[200]	[600, 100]	[1000, 800, 100]
EMNIST	[200]	[600, 100]	[1000, 800, 100]
Fashion-MNIST	[200]	[600, 100]	[1000, 800, 100]
Chess	[30]	[60, 40]	[100, 100, 50]

Table 5.17: Manually set values for the topology.

Table 5.18 presents the results, compared with the base tests. As there are a lot of results, the ensemble accuracies are not shown for these tests. Values performing better than the base tests are put in bold.

For the topology, none of the overall averages for the different sizes can compete with dynamical evolution. As can be seen, manually set topologies actually perform better for some particular tests, like a small topology for MNIST or a medium one for EMNIST. This is probably because the manually set values happen to work well for these cases, and these values did not happen to be tested during evolution. The noisy nature of the results might also be a partial explanation for this. On the other hand, especially the chess dataset performs much worse with manually set values. Overall, the topology seems quite important to evolve in order to ensure good results. This makes a lot of sense since the topology has such a large search space, hence making it more difficult to manually pick a value that works well.

For activation functions, using only ReLU actually improves the results for three of the datasets, but on the other hand decreases the accuracy for EMNIST by quite a lot. Overall for the four datasets, the average accuracy ends up equal to the base tests. Using only Sigmoid also works decently, but not as well as ReLU or evolution. To summarize, it seems

Hyperparameter	MNIST	EMNIST	Fashion-MNIST	Chess	Overall Average
All Hyperparameters Evolved (base tests)	96.6 %	79.1 %	87.3 %	88.2 %	87.8 %
Topology, Small	**97.1 %**	76.3 %	86.3 %	72.0 %	83.0 %
Topology, Medium	**97.2 %**	**79.5 %**	86.6 %	81.1 %	86.1 %
Topology, Large	**96.9 %**	78.7 %	85.5 %	86.3 %	86.8 %
Activation Function, Sigmoid	**96.7 %**	77.4 %	87.0 %	86.8 %	87.0 %
Activation Function, ReLU	**97.0 %**	78.3 %	**87.4 %**	**88.7 %**	87.8 %
Cost Function, MSE	96.3 %	76.4 %	**87.9 %**	86.7 %	86.8 %
Cost Function, Cross Entropy	96.4 %	**79.7 %**	87.3 %	**88.6 %**	**88.0 %**
Optimizer, Adam	**97.1 %**	78.5 %	87.0 %	**88.7 %**	87.8 %
Learning Rate and Decay, (0.01, 0)	96.2 %	78.1 %	86.9 %	**88.3 %**	87.3 %
Learning Rate and Decay, (0.1, 0)	**96.8 %**	76.5 %	85.5 %	86.8 %	86.4 %
Learning Rate and Decay, (0.1, 10^{-7})	**96.8 %**	78.8 %	**87.9 %**	87.1 %	87.6 %
Dropout Probability, 0 %	95.8 %	76.9 %	86.6 %	88.1 %	86.8 %
Dropout Probability, 50 %	**97.1 %**	78.5 %	86.6 %	66.0 %	82.0 %
Dropout Probability, 70 %	96.6 %	76.8 %	86.5 %	57.0 %	79.2 %
Batch Size, 32	**97.1 %**	76.7 %	86.5 %	86.8 %	86.8 %
Batch Size, 64	96.4 %	77.7 %	86.1 %	86.3 %	86.6 %
Batch Size, 128	**96.9 %**	**80.4 %**	**88.1 %**	87.2 %	**88.1 %**

Table 5.18: Results for manually set hyperparameters. Values in bold are better than the base tests.

like the activation function is not very important to evolve, as setting it manually to ReLU works quite well (Swish might work even better, based on the results of Chapter 5.2.1). The reason why only using ReLU sometimes improves the results, is because it makes the search space smaller, meaning more configurations of the other hyperparameters can be tested, instead of wasting individuals with activation functions that do not work well.

Cross Entropy works better than evolution overall, while MSE performs quite a bit worse, especially for the EMNIST and chess datasets. As Cross Entropy generally works well for classification problems, this might change if the system should handle other types of problems as well. However, for classification problems, it seems like setting the cost function to Cross Entropy works at least as well as letting it be evolved.

Adam performs better than evolution for MNIST and the chess dataset, but worse for EMNIST and fashion-MNIST. The overall average scores exactly the same with Adam as with evolution. Hence, evolution seems not to be very important for the optimizer, but might sometimes have an advantage over manually setting it to Adam.

For the learning rate and learning rate decay, evolution performs better than manually set values. Even though some configurations of manually set values can compete with evolution (e.g. learning rate of 0.1 and decay of 10^{-7} for fashion-MNIST, and learning rate of 0.01 and no decay for the chess dataset), no configuration works better than evolution in general across all datasets.

Manually setting the dropout probability gives some interesting results. Using a dropout probability of 50 % works well for all image datasets, but is a disaster when it comes to the chess dataset. The safe option is therefore to not use dropout (i.e. a dropout probability of 0 %), but as can be seen by the overall average, this does perform worse than evolution. Therefore it seems like it is quite valuable to evolve the dropout probability instead of setting it manually.

Finally, the system performs better when setting the batch size to 128 than when letting it be set through evolution. An interesting explanation for this can be that a CPU can process batch sizes which are a power of 2 faster, hence being able to train for more steps, compared to a batch size of a random integer, like it is evolved by the system. A possible improvement can therefore be to let the system only evolve batch sizes which are a power of 2, in order to speed up the training.

In summary, it seems like the topology, learning rate and decay, and dropout probability are the most important hyperparameters to evolve, as they cannot easily be replaced by default values. The least important hyperparameters to evolve are the activation function, cost function and optimizer, with the batch size somewhere in the middle. There is a clear correlation between the hardness of setting a hyperparameter manually and the size of the search space of that hyperparameter. The three hyperparameters which can be set easily are also those whose search space consist of only a few values, while those which are harder to set consist of infinitely large search spaces.

Ablations Related to Research Question 3

Research question 3 concerns the importance of diversity as part of the fitness function. For the system, this is done through the PFC value and aging. There are of course also other, more common ways to ensure diversity used by the system, listed here:

- Random immigrants
- Crossover
- Mutation
- Stochastic selection of parents
- Stochastic removal of individuals
- Random initialization of individuals
- *For the merging subpopulations system:* Separate populations

However, this research question concerns the explicit usage of diversity in the fitness function, meaning only PFC and aging will be discussed. By letting the value of the α-parameter of the system be constantly at 0 (done in practice by setting the parameter determining the ratio of final generations where only accuracy should be considered by the fitness function to 1), diversity will not be considered by the fitness function, hence creating an ablation test which will measure the system's performance without diversity. The fitness value will now be equal to the ratio of correctly classified validation examples (i.e. the accuracy). Each test has been run 4 times, and the results are presented in Table 5.19.

Dataset	**Without Diversity**	**With Diversity (base test)**
MNIST	96.8 % (96.9 %)	96.6 % (97.3 %)
EMNIST	79.8 % (81.3 %)	79.1 % (80.6 %)
Fashion-MNIST	86.9 % (87.3 %)	87.3 % (87.9 %)
Chess	89.3 % (91.5 %)	88.2 % (91.1 %)
Overall Average	**88.2 % (89.2 %)**	**87.8 % (89.2 %)**

Table 5.19: Results for using only accuracy in the fitness function. The first value represents the average accuracy on the test set for the best net, while the one in parentheses represents the average accuracy on the test set by the ensemble of all individuals in the final population.

The results show that overall, the system actually performs better without diversity. This probably means that the more built-in, implicit diversity ensuring measures listed above are enough for the system to create a diverse population, and that bringing diversity explicitly into the fitness function is not necessary. It might create a larger problem by removing well-performing, but less diverse individuals, than it does good by ensuring diversity. When it comes to the ensembles, the exact same overall average is achieved with and without diversity. The reason why diversity is more important for ensembles than for a single best network might be that diversity is one of the two criteria for a well-performing ensemble (see Chapter 3.2.3).

One should be careful concluding evolution of neural networks performs worse with diversity as an explicit part of the fitness function in general, as these results are only based on one type of diversity measure, using PFC and aging. That being said, as mentioned in Chapter 4.1.2, other types of diversity measures were tried while the main system was being implemented, including a diversity measure based directly on the genomes, and one building a family tree measuring hierarchical distance. None of these performed as well as PFC, but it is important to emphasize that only a few tests were run at this stage, hence giving the results little statistical significance. Additionally, as the trade-off tests in the next subchapter will show, not using aging does not make the results better either. Therefore, none among several diversity methods performed better than simply ignoring diversity for the fitness function.

Ablations Related to Research Question 4

Research question 4 concerns incremental complexification starting from a minimal structure. In order to see if this approach has been efficient, two alternative approaches have been tested: a destructive one and a hybrid one.

The destructive approach starts from a very complex structure, and gradually decreases that complexity. In order to be the complete opposite of incremental complexification, it should start from a maximal complex structure. However, no such maximum exists, as the number of layers and neurons can grow indefinitely. The starting topology has therefore been set to 5 hidden layers of 2,500 neurons each. The probabilities for the different mutations have also been changed, as shown in Table 5.20, to make the mutations removing neurons and layers more probable.

Mutation	**Probability**
Add Neurons	3.5 %
Add Layer	2.5 %
Remove Neurons	31.5 %
Remove Layer	12.5 %
Sum, Topology	**50 %**

Table 5.20: Mutation probabilities for the destructive approach.

The hybrid approach tries to start with topologies somewhat in between of a minimal and maximal complex structure. The number of layers is drawn from a geometric distribution with $p = 0.5$, and the number of neurons in each layer is drawn from geometric distributions with $p = 0.1$. Because this approach might need to either increase or decrease the topology size, the probabilities for the mutations for adding and removing neurons and layers are set equal. However, as Table 5.21 shows, there is still a higher probability for altering the number of neurons than the number of layers, as the latter is a more drastic mutation.

Mutation	**Probability**
Add Neurons	17.5 %
Add Layer	7.5 %
Remove Neurons	17.5 %
Remove Layer	7.5 %
Sum, Topology	**50 %**

Table 5.21: Mutation probabilities for the hybrid approach.

Each version was run 4 times for each of the datasets. Table 5.22 summarizes the results.

Dataset	**Destructive**	**Hybrid**	**Incremental (base test)**
MNIST	96.5 % (97.1 %)	97.0 % (97.3 %)	96.6 % (97.3 %)
EMNIST	78.7 % (80.4 %)	79.5 % (80.7 %)	79.1 % (80.6 %)
Fashion-MNIST	87.1 % (87.9 %)	87.5 % (87.8 %)	87.3 % (87.9 %)
Chess	88.4 % (91.5 %)	88.1 % (90.8 %)	88.2 % (91.1 %)
Overall Average	**87.7 % (89.2 %)**	**88.0 % (89.1 %)**	**87.8 % (89.2 %)**

Table 5.22: Results for alternatives to incremental complexification. The first value represents the average accuracy on the test set for the best net, while the one in parentheses represents the average accuracy on the test set by the ensemble of all individuals in the final population.

As can be seen from these results, the hybrid approach scores slightly better overall for the single best neural network, while the destructive and incremental approaches are minimally better for the ensembles. However, the differences are so small they could partially be due to noise. Therefore, it seems like this choice of topology approach is not very important for the performance of a neuroevolution system. That being said, there are a few aspects to be discussed.

Incremental complexification was primarily used in this thesis because it was recommended by previous literature, like Stanley and Miikkulainen (2002). However, in these relatively old papers, the systems were usually tested on simple problems, where the optimal solutions often were relatively close to a minimal structure. Therefore, the idea of incremental complexification might have worked primarily because the starting point was already close to the solution, and not because the approach worked particularly well in general.

A big disadvantage with the destructive approach is that in the early generations, the very large networks will use a lot of storage space and memory. Furthermore, if a constant number of training steps is used, the training time will be very long in the early generations as the nets are so large (this was not a problem for the system of this thesis as a constant training time was used). The storage and memory problem, and possibly the training time problem, makes an incremental or hybrid approach more attractive.

The hybrid approach has two advantages compared to the other ones. Firstly, the initial population starts somewhat in the middle of the search space, and can from there spread in every direction. In general, this should make the system able to explore a larger part of the search space faster than an extreme approach starting on the edge of the search space, like the incremental and destructive ones do. Secondly, as there is some randomness in the initial topologies for the hybrid approach, not all initial individuals will be identical in terms of the topology. Therefore, a larger part of the search space will be explored right from the start. That being said, such diversity of the initial population could also be implemented in an incremental or destructive approach, by letting the topologies vary slightly, even though they are all yet very simple or very complex.

Even though these advantages of the hybrid solution are not reflected that much by the final results shown above, they are illustrated by looking at plots of the accuracy of the best network in the population in every generation. Figure 5.7 shows this for a run using the hybrid approach for the EMNIST dataset, while Figure 5.8 shows a similar run using incremental complexification (i.e. a base test). As can be seen, the hybrid approach performs decently right from the start, while incremental complexification uses about 10 generations before the performance reaches a decent level. After this, both approaches follow the same trend, by slowly improving. But one could argue the first 10 generations with incremental complexification are somewhat wasteful. If the system will only run for a few generations, this difference might severely affect the results. This tendency was also present for the other runs and with the other datasets. The destructive approach gave results with approximately the same tendency as the incremental one, as it used some generations to reduce the topologies enough to have time to train them properly. Hence, the hybrid approach seems to be the best one overall.

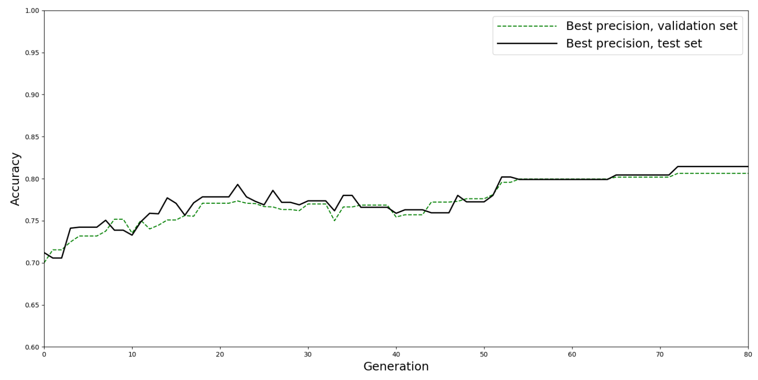


Figure 5.7: Performance with EMNIST using the hybrid approach. The system performs quite well right from the first generation.

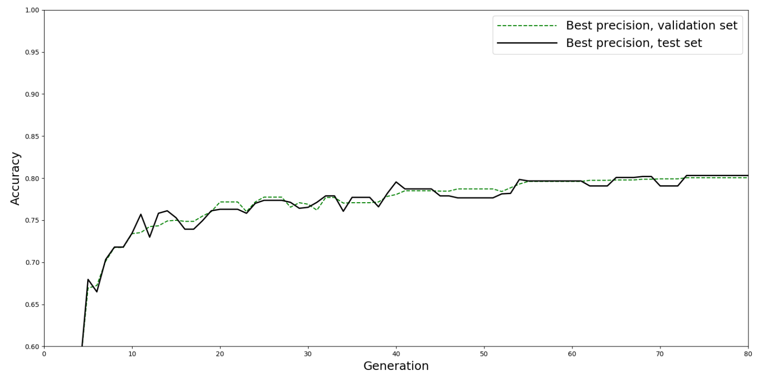


Figure 5.8: Performance with EMNIST using incremental complexification. The system needs about 10 generations before reaching a decent level of performance.

5.2.3 Trade-off Tests

The twelve parameters of the main system have been run with different values in order to try to show the clear trade-off between either accuracy and time usage, or exploration and exploitation (see Chapter 4.1.4). The results of these tests will be tied to research question 1, as clear trade-offs mean the system is easy to use, hence making it more suitable for evolving neural networks. In total, 288 tests have been run during this phase. A lot of what was said for the ablation tests, also applies here. The yeast dataset is dropped for the same reason, and each test is run twice for each dataset. For every parameter not currently being tested, the same values as for the ablation tests, found in Table 5.13, have been used.

First, the eight parameters with trade-off between accuracy and time usage will be considered. Those are:

- Generations
- Population Size
- Children per Generation
- Training Time per Individual
- Validation Interval
- Patience
- PFC Sample Rate
- PFC Individuals to Compare With

In order to show the trade-off for these parameters, the average final accuracy and average run time for the different tests are presented in Table 5.23. To make the results easier to read, only the overall average across all four datasets are presented in the table. For the same reason, the ensemble accuracies are dropped from this table.

For the number of generations, the accuracy increases drastically when the value is increased from 10 to 40, but from there on, the improvement is much smaller with even more generations. The time usage increases approximately linearly with the number of generations, and actually slightly faster, as in the early generations, there are more small networks, which in turn are more frequently subject to early stopping. Based on these results, there is indeed a clear trade-off here.

The population size is also subject to a clear trade-off, though the value chosen does not affect the accuracy or time usage as much as the number of generations does. This makes sense as the population size only determines the number of individuals to be kept in the population, and not the number of nets that must be trained in each generation (that is determined by the children per generation parameter). Therefore, the primary cause of the additional time used with a higher population size, is due to the quadratic time complexity for calculating the PFC diversity values of the individuals.

The number of children per generation affects the time usage a lot, as more individuals must be trained if its value is higher. A higher value also generally increases the accuracy, hence making the clear trade-off. It should be mentioned that the accuracy drops by 0.1 % when increased from 80 to 120, but this is most likely just due to noise in the results, because of the stochastic behavior of the system.

Naturally, the training time per individual follows the same tendency as the above parameters, as more training steps let the networks train better, but also spends more time doing so. Without the use of early stopping, there would be a risk of the accuracy dropping with a too long training time due to overfitting (see Chapter 2.1.1), but as long as early stopping is used, a longer training time should never hurt the performance.

Parameter	**Value**	**Accuracy**	**Time Usage (minutes)**
Generations			
	10	77.7 %	95
	40	87.2 %	396
	80 (base test)	87.8 %	797
	150	87.8 %	1504
Population Size			
	10	86.1 %	696
	40	87.6 %	735
	80 (base test)	87.8 %	797
	120	87.9 %	884
Children per Generation			
	10	84.5 %	189
	40	87.7 %	445
	80 (base test)	87.8 %	797
	120	87.7 %	1286
Training Time per Individual			
	10	83.7 %	314
	60	87.0 %	560
	120 (base test)	87.8 %	797
	200	87.8 %	1197
Validation Interval			
	1	88.0 %	803
	100 (base test)	87.8 %	797
	2,500	87.8 %	795
	∞	85.9 %	836
Patience			
	100	87.5 %	660
	1,000	87.8 %	778
	10,000 (base test)	87.8 %	797
	∞	87.7 %	842
PFC Sample Rate			
	0.01	87.5 %	648
	0.2	87.6 %	681
	0.5	87.8 %	721
	1 (base test)	87.8 %	797
PFC Individuals to Compare With			
	1	87.6 %	646
	10	87.7 %	645
	40	87.8 %	686
	all (79) (base test)	87.8 %	797

Table 5.23: Results showing the trade-off between accuracy and time usage. The results are the average for all four datasets.

A lower validation interval should statistically improve the results, as a more fine-grained validation means the best version of the network can be restored more precisely when the training is completed. At the same time, more frequent validations will use more time. This is also reflected by the results, but the time usage actually increases when the interval is set to ∞ (i.e. no validations during training). This is caused by the fact that no validation means no early stopping can happen, hence making all networks having to train for the whole training time. That means there is a clear trade-off, as long as the validation interval is not set extremely high.

The patience affects the accuracy and time usage only slightly, as it only determines how quickly early stopping happens. If its value is too low, the training might stop too early, hence making the results poor. Based on the results, it seems like a patience of about 1,000 training steps is enough for the system to perform well. The only reason for it to drop by 0.1 % when set to ∞ is due to noise.

The PFC sample rate affects the time used to calculate the PFC values in every generation approximately linearly, but as a much longer time is used to train the nets, it does not affect the total time usage by that much. The accuracy also slightly increases with a higher value, as the PFC value becomes more precise if more samples are used to calculate it. The trade-off is therefore clear.

Finally, the PFC individuals to compare with also give a better accuracy and a longer time usage when the value is higher, as it gives a more precise PFC value, but uses more time to calculate it. The results show a 1 minute lower average run time with 10 individuals instead of just 1, however, this must be due to noise.

To summarize, all eight parameters discussed above show a clear trade-off between accuracy and time usage. Some barely affect the accuracy (e.g. PFC sample rate and PFC individuals to compare with), while others are quite critical (e.g. generations and training time per individual). However, with these parameters, it should not be too hard for users to pick values which balance their needs for accuracy in the evolved nets with the time they have available.

The four parameters with a trade-off between exploration and exploitation are:

- Crossover Rate
- Random Immigrant Rate
- Age Factor
- Final Generations with Accuracy Only Ratio

A simple measure of exploration is the average diversity measured by the PFC value across all generations of a run. More information could be drawn from investigating the diversity at each generation separately, but in order to be able to summarize the results into a manageable quantity, the averages have been used. In the same way, the average accuracy on the validation set of all individuals across all generations can be used as a measure of exploitation. More exploitative approaches will most likely have a higher accuracy in early generations, and in a larger part of the population at any given time, hence also achieving a higher average accuracy. The accuracy of only the final generation should not be used, as exploitation is a measure of focusing on the most promising solutions along the way, and does not necessarily achieve better end results than a more exploratory approach. Table 5.24 summarizes the results of these tests. Just as with the results for the parameters with a trade-off between accuracy and time usage presented above, only the overall average across all four datasets are shown to make the results more readable.

Parameter	**Value**	**Exploration Score**	**Exploitation Score**	**Final Accuracy**
Crossover Rate				
	0	63.0 %	55.8 %	87.3 %
	0.2	62.5 %	56.0 %	88.0 %
	0.5 (base test)	61.8 %	56.1 %	87.8 %
	1	61.1 %	59.3 %	87.5 %
Random Immigrant Rate				
	0	61.2 %	59.1 %	87.8 %
	0.1 (base test)	61.8 %	56.1 %	87.8 %
	0.3	62.6 %	54.3 %	87.6 %
	0.9	57.9 %	42.6 %	84.6 %
Age Factor				
	0	61.7 %	56.5 %	87.6 %
	0.2	61.9 %	56.5 %	87.7 %
	2 (base test)	61.8 %	56.1 %	87.8 %
	10	63.0 %	55.5 %	88.0 %
Final Generations with Accuracy Only Ratio				
	0	63.9 %	53.8 %	87.5 %
	0.25 (base test)	61.8 %	56.1 %	87.8 %
	0.5	59.7 %	57.4 %	87.9 %
	1	56.7 %	61.8 %	88.2 %

Table 5.24: Results showing the trade-off between exploration and exploitation. Exploration score is the average PFC score across all generations, and exploitation score is the average accuracy on the testing set across all generations. All scores are the average across all four datasets.

For the crossover rate, the results clearly indicate that a higher value decreases the exploration score and increases the exploitation score. This is very surprising as children created by crossover with two parents should be more different from its parents than children created from a single parent. No plausible explanation has been found for why this happens, however, all results indicate that this is in fact the case. To investigate this further might be the basis of a future work. The takeaway from this is that a high crossover rate focuses more on exploitation, while a lower one focuses more on exploration, instead of the other way around as expected, at least for this system. The clear trade-off is still present, though.

For the random immigrant rate, there is a clear tendency of a higher value increasing the exploration and decreasing the exploitation, except for the last value of 0.9. More random immigrants should make the algorithm more exploratory, so why that value drops with a high random immigrant rate is quite a mystery. The most probable explanation can be found by also looking at the exploitation score and final prediction, as well as how the PFC value used to determine the exploration score is defined (see Chapter 3.2.3). As the system evolves quite poor networks with such a high random immigrant rate, the networks will make more mistakes than usual on average. As the number of wrong predictions are used as a denominator when calculating the PFC value, a significantly higher number of such might lower the PFC value, even though the system is not less exploratory. PFC is therefore not an ideal way of measuring exploration, but because it is the diversity measure used by the system, it is still used here. Furthermore, it works well enough for most of the results presented. Independently of this, the trade-off is clear as long as one does not use an extremely high value.

When it comes to the age factor, a higher value increases the exploration score slightly, and lowers the exploitation score, but it is not by much. The reason why it affects especially the exploration score by so little, is that it creates a temporal diverse population instead of a spatial one (see Chapter 4.1.2). As PFC measures spatial diversity in the population at any given time, the exploration measure does not really capture this type of diversity.

For the ratio of final generations with only accuracy considered by the fitness function, it is a clear tendency that a higher value lowers the exploration score, while increasing the exploitation score. This is very logical, as the fitness function focuses more on diversity and exploration for more generations if the value is low. Also, the fact that the final accuracy keeps increasing with a higher value of this parameter supports the answer of research question 3 given in the previous subchapter (i.e. that the system performs better without diversity as part of the fitness function).

In summary, all four parameters of this type show a quite clear trade-off between exploration and exploitation. The crossover rate behaves counterintuitively, but that still means there is a clear trade-off. A very high random immigrant rate breaks the trade-off pattern by having a low exploration score, but this is most likely due to the characteristics of the PFC measure. Additionally, one would never use a very high random immigrant rate in practice. The age factor does not affect the scores by much, but the trade-off can still be seen. Finally, the final generations with accuracy only ratio shows a very clear trade-off.

When it comes to the final accuracy of each test, it can be seen that the parameter values do not affect the results by a lot, except for when the values become quite extreme (e.g. a random immigrant rate of 0.9). This applies for both the parameters with an accuracy-time usage trade-off, and those with an exploration-exploitation trade-off. This strengthens the claim that was made in Chapter 4.1.4, that the parameter values of a neuroevolution system are much more robust, affect the results less, and are easier to set than those of a neural network.

5.3 Summary

This chapter started with a description of the hardware and software environment used to run the tests. Then, the results of the tests run were presented and analyzed. This included long running tests to evaluate the actual performance of the three systems on the different datasets, and more specialized tests on the main system to answer the research questions and to show the clear trade-offs for the parameters of the system.

Chapter 6

Conclusion and Future Work

In this part, the results and analysis will be used to answer the research questions. This is found in Chapter 6.1. Thereafter, in Chapter 6.2, possible future directions and remaining work will be discussed.

6.1 Conclusion

Now, after five chapters and a deep dive into the field of neuroevolution, it is time to answer the research questions presented in Chapter 1, and draw some conclusions. The four research question are repeated here:

- To what degree can well-performing neural networks be created by directly evolving most of the hyperparameters which normally must be set manually?
- Which hyperparameters are most critical to evolve for the neural networks to be well-performing?
- How important is it to include diversity as part of the fitness function when it comes to evolution of such neural networks?
- What are the advantages of starting with a minimal network structure, and then perform incremental complexification, instead of starting with more complex networks?

The tests presented in Chapter 5.2.1 are the basis for answering research question 1. For all datasets, the main system and merging subpopulations system are capable of evolving networks performing well, and the convolutional system evolves even better nets for the image datasets. For famous datasets like MNIST which have been subject to experiments over many years, this method is not really capable of challenging the state-of-the-art results, but for a new dataset with few benchmarked results, systems like these ones might have a value, and achieve better results in shorter time than what can be done through manual trial-and-error hyperparameter tuning. The major drawback with such a system is that it requires a lot of computational resources, meaning it might not be practically feasible in all cases. Also, rather large datasets with many data examples are needed in order to guarantee that well-performing networks will be evolved. That being said, with the probable continual increase in computational power over time, such neuroevolution systems might become progressively more attractive. In addition to these results, tests in Chapter 5.2.2 showed that the systems perform much better than random search for creating neural nets, meaning evolution is well-suited for this task, to the degree this was questioned. Finally, tests in Chapter 5.2.3 showed that all parameters of the new systems have a clear trade-off, meaning they are easy to set, have a more predictable behavior, and can usually perform well with default values. The parameter values selected also affect the results less than what the hyperparameter values for a neural network do, hence making the choice of value less important, and the system more robust.

To answer research question 2, each hyperparameter was replaced by manually set values, one at a time. If the networks performed better or just as well with the manually set values as with evolved values for a specific hyperparameter, that hyperparameter can be considered less critical for the neural networks to perform well. The results of these tests were presented and analyzed in Chapter 5.2.2. They indicate that the topology, learning rate and decay, and dropout probability are the most critical hyperparameters to evolve, while the activation function, cost function and optimizer are the least critical and easiest to replace by manual values. Using ReLU as activation function, Cross Entropy as cost function, and Adam as optimizer works well in general. The batch size hyperparameter lands somewhere in the middle. All of this shows there is a clear positive correlation between the size of the search space of a hyperparameter, and how critical it is.

Research question 3 can be answered based on the ablation tests which did not include diversity as part of the fitness function, presented in Chapter 5.2.2. As it turned out, the system actually performed slightly better when the fitness function only considered the accuracy of the networks, without diversity. This indicates that the other components of the genetic algorithm (e.g. mutation and stochastic selection of parents) are enough to ensure diversity, hence making it unnecessary to include a diversity measure explicitly in the fitness function. That being said, more tests could be conducted on other types of diversity measures, which could work better than what has been tried in this thesis.

In order to answer research question 4, two alternatives to incremental complexification were tested in Chapter 5.2.2; a destructive algorithm starting with a very complex topology, and a hybrid approach starting somewhat in the middle. The results showed that the three approaches (i.e. incremental complexification and the two alternatives) worked fairly equally, with a slight edge for the hybrid one. The hybrid also has the advantage of performing quite well from the first generations, while the other two use a few generations before starting to perform well. Stanley and Miikkulainen (2002) suggested incremental complexification, but it could be it worked well there primarily because the solutions evolved had a close to minimal topology, as their system was tested on fairly simply problems. Based on the results in this thesis, it seems like incremental complexification does not have any particular advantages, and that the hybrid approach is the best one overall, even though the choice is not the most critical one in order to evolve well-performing networks.

Even though all four research questions have been answered, it is important to emphasize that even more tests must be conducted before these answers can be given with absolute certainty. This both includes testing on more datasets, and running more tests for each dataset. The problem with this is that it requires a lot of computational resources. That being said, the conclusion drawn at the end of this thesis is that it works fairly well to evolve the hyperparameters of a neural network directly in the fashion used by the three implemented systems. As a last part of this chapter, some possible directions of future work based on this thesis will be discussed.

6.2 Future Work

The work of this thesis has been able to give some answers of the research questions, but there are many directions in which it can be continued. The most interesting ideas are now briefly presented.

Creation of One Universal System

In order to automate the process of creating neural networks even further, it would be interesting to create a system that can handle all types of problems. That means the system should be capable of choosing between dense, convolutional and other types of architectures by itself, be expanded to also handle regression problems, and support an accuracy measure which works well for unbalanced datasets.

As the results of Chapter 5.2.1 showed, the convolutional system performs much better for the image datasets, while much worse on the chess set, and it is not even compatible with the yeast set. By letting the system attempt to evolve nets with both approaches, the user would no longer have to predict what approach would work better for a new dataset. That being said, it does increase the search space of the algorithm and requires some additional functionality in order to determine if the dataset is compatible with convolution. Also, for most datasets it should be fairly easy for the user to determine if convolution will work well, meaning this is not necessarily the way to go. To try it out, however, seems like an interesting experiment for a future work. Related to this, it will also be interesting to try to evolve the convolutional architectures more freely, without the restrictions described in Chapter 4.3, and to let other types of architectures be evolved, like recurrent layers and skip connections. Skip connections were tried briefly early on during this thesis without too impressive results, but could be tested further.

The results could not give a clear answer to whether or not the use of merging subpopulations improve the results in general. For some datasets (i.e. EMNIST and chess) the main system performed better, while for others (i.e. MNIST, fashion-MNIST and yeast) the merging subpopulations system did better. There is no simple way of combining the two systems (like the main system and convolutional system easily could, as discussed above), as they differ in what way the algorithm itself works, and not just by how the nets are evolved. One could either keep both systems and let the user make the choice to use one of the systems or try both, or do more testing on even more datasets to either conclude what system works best on average, or find general tendencies to predict what system works best for a new dataset. It is also possible to create a convolutional merging subpopulations system, with the properties of both of these systems, as they are not mutually exclusive.

In order to handle unbalanced datasets and regression problems, all that is required is a change to the fitness function. As an example, a measure like F1 score instead of just the ratio of correctly classified validation examples can be used to calculate the accuracy for unbalanced classification datasets, and Mean Square Error can be used for a net in a regression problems. A modification to the PFC measure is also needed in order to support regression.

In Chapter 4.1.3, two different types of problems that neuroevolution systems might solve were discussed. One was to try finding the best hyperparameter values for a problem in general, while the other was to find the best trained net for a problem. During this thesis, the second one was chosen as it required less dataset partitions and less time to run each test. However, future work might try to focus on the first of these approaches as well.

Further Testing of the Systems

Another way of continuing this work is simply to test the systems further, on more datasets and with more configurations of the parameters of the systems. Also, alternative methods to those used can be tried, like alternative initialization and parent selection methods. In particular, the following two ideas seem interesting to investigate further:

- Letting the probabilities for the different mutations vary dynamically based on what has worked well in the past. A mutation probability is increased if an individual subject to it performs well, and decreased if it performs poorly.
- Evolving different hyperparameter values for different layers. Especially the learning rate and decay, activation function and dropout probability could be interesting to try holding independently for the different layers.

Both of these ideas were tried briefly in the early stages of testing during this thesis, as described in Chapter 4, but did not perform quite as well as the systems did without them. However, with more testing and tuning of these components, they could start improving the results.

Additionally, it could be interesting to further test the systems without diversity included in the fitness function, as the answer to research question 3 indicates the systems work slightly better that way. The same applies for further testing of alternatives to incremental complexification, like more thoroughly testing different hybrid approaches, as the answer to research question 4 slightly favored that approach.

Result Improving Components

In this thesis, the focus has been to create a system which could be used to answer the research questions, and not to achieve the best results possible. Probably, the results can be improved by including some data preprocessing, feature engineering, and more sophisticated ensembles. A future work could be to investigate what methods go well with the systems, and possibly even let which methods that are used for a particular network be evolved alongside the neural net hyperparameters.

When it comes to the ensembles, the choice of including every individual from the final population, like the systems currently do, is very simple. An idea that might work even better is to use only a subset of the individuals. If so, one needs a way of determining what individuals should be part of this ensemble. Should it be the ones with the highest accuracy, or should also a diversity measure be included in order to ensure a diverse ensemble? Also, as aging creates a temporal diversity in the population (see Chapter 4.1.2), the best individuals for an ensemble are not necessarily all in the final population. To store some of the best individuals from earlier generations for use in an ensemble, would therefore also be interesting to try.

The main reason why all of these future work ideas have not been tried during this thesis is the limitations on computational resources, meaning there was no time to properly test them, at least not with any statistical significance.

References

Ahire, J. B. (2018). *Artificial Neural Networks: The brain behind AI.* Independently Published.

Arellano-Verdejo, J., Godoy-Calderon, S., Alonso-Pecina, F., Arenas, A. G. & Cruz-Chavez, M. A. (2017). A New Efficient Entropy Population-Merging Parallel Model for Evolutionary Algorithms. *International Journal of Computational Intelligence Systems, 10*(1), 1186-1197. https://dx.doi.org/10.2991/ijcis.10.1.78

Bottou, L. (2019, January 23rd). Curiously Fast Convergence of some Stochastic Gradient Descent Algorithms. Retrieved from https://leon.bottou.org/publications/pdf/slds-2009.pdf

Brownlee, J. (2019a, February 4th). How to use Data Scaling, Improve Deep Learning Model Stability and Performance. Retrieved from https://machinelearningmastery.com/how-to-improve-neural-network-stability-and-modeling-performance-with-data-scaling/

Brownlee, J. (2019b, March 1st). Why Training a Neural Network Is Hard. Retrieved from https://machinelearningmastery.com/why-training-a-neural-network-is-hard

Børstad, E. (2019, December 10th). Utvikling av dype nevrale nett med genetiske algoritmer: Midtveisrapport [Evolving Deep Neural Networks using Genetic Algorithms: Midterm Report]. Unpublished.

Chalmers, D. J. (1991). The Evolution of Learning: An Experiment in Genetic Connectionism. *Connectionist Models: Proceedings of the 1990 Summer School*, 81-90. https://doi.org/10.1016/B978-1-4832-1448-1.50014-7

Chandra, A. & Yao, X. (2006). Ensemble Learning Using Multi-objective Evolutionary Algorithms. *Journal of Mathematical Modelling and Algorithms, 2006*(5), 417-445. https://doi.org/10.1007/s10852-005-9020-3

Chesscom. (2020, April 16th). How to Play Chess — Rules + 7 Steps to Begin. Retrieved from https://www.chess.com/learn-how-to-play-chess

Cohen, G., Afshar, S., Tapson, J. & van Schaik, A. (2017). EMNIST: an extension of MNIST to handwritten letters. Retrieved from http://arxiv.org/abs/1702.05373

Dasgupta, C., Papadimitriou, C. H. & Vazirani, U. V. (2006, July 18th). *Algorithms.* n.p.

Erb, R. J. (1993, February). Introduction to Backpropagation Neural Network Computation. *Pharmaceutical Research, 1993*(10), 165-170. https://doi.org/10.1023/A:1018966222807

Floreano, D., Dürr, P. & Mattiussi, C. (2008,, January 10th). Neuroevolution: from architectures to learning. *Evolutionary Intelligence, 2008*(1), 47-62. https://doi.org/10.1007/s12065-007-0002-4

Ghiasi, G., Lin, T. & Le, Q. V. (2018). DropBlock: A regularization method for convolutional networks. *Advances in Neural Information Processing Systems*(31). Retrieved from http://papers.nips.cc/paper/8271-dropblock-a-regularization-method-for-convolutional-networks

Glorot, X. & Bengio, Y. (2010, March). Understanding the difficulty of training deep feedforward neural networks. *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 249-256. Retrieved from http://www.jmlr.org/proceedings/papers/v9/glorot10a/glorot10a.pdf

Gomez, F. & Miikkuulainen, R. (1997, January 1st). Incremental Evolution of Complex General Behavior. *Adaptive Behavior - Animals, Animats, Software Agents, Robots, Adaptive Systems, 1997.* https://doi.org/10.1177/F105971239700500305

Google Brain. (n.d.). TensorFlow. Retrieved from https://www.tensorflow.org

Gürbüzbalaban, M., Ozdaglar, A. & Parrilo, P. (2019, January 9th). Why Random Reshuffling Beats Stochastic Gradient Descent. Retrieved from https://arxiv.org/abs/1510.08560

He, X., Zhao, K. & Chu, X. (2019, August). AutoML: A Survey of the State-of-the-Art. Retrieved from https://arxiv.org/abs/1908.00709

Horton, P. & Nakai, K. (1996, June). A Probabilistic Classification System for Predicting the Cellular Localization Sites of Proteins. *Proceedings of the Fourth International Conference on Intelligent Systems for Molecular Biology*, 109-115. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.7487

Hwang, M. W, Choi, J. Y. & Park, J. (1997, April). Evolutionary Projection Neural Networks. *Proceedings of 1997 IEEE International Conference on Evolutionary Computation*, 667-671. https://doi.org/10.1109/5.784219

Ioffe, S. & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Retrieved from https://arxiv.org/abs/1502.03167

Jayasundara, V., Jayasekara, S., Jayasekara, H., Rajasegaran, J., Seneviratne, S. & Rodrigo, R. (2019, April 17th). TextCaps: Handwritten Character Recognition with Very Small Datasets. *2019 IEEE Winter Conference on Applications of Computer Vision, 2019*, 254-262. https://doi.org/10.1109/WACV.2019.00033

Kim, H. B., Jung, S. H., Kim, T. G. & Park, K. H. (1996, May). Fast learning method for back-propagation neural network by evolutionary adaptation of learning rates. *Neurocomputing, 11*(1), 101-106. https://doi.org/10.1016/0925-2312(96)00009-4

Lau, S. (2017, July 29th). Learning Rate Schedules and Adaptive Learning Rate Methods for Deep Learning. Retrieved from https://towardsdatascience.com/learning-rate-schedules-and-adaptive-learning-rate-methods-for-deep-learning-2c8f433990d1

LeCun, Y., Cortes, C. & Burges, C. J. C. (n.d.). The MNIST Database of handwritten digits. Retrieved from http://yann.lecun.com/exdb/mnist

Lehman, J. & Stanley, K. O. (2008). Exploiting Open-Endedness to Solve Problems Through the Search for Novelty. *Proceedings of the Eleventh International Conference on Artificial Life*, 329-337. Retrieved from https://pdfs.semanticscholar.org/8cb8/3e8ad10fed42f24932cead992b4ae9ba1625.pdf

Lehman, J. & Stanley, K. O. (2011). Novelty Search and the Problem with Objectives. In R. Riolo, E. Vladislavleva and J. Moore, *Genetic Programming Theory and Practice IX* (pp. 37-56). New York: Springer. https://doi.org/10.1007/978-1-4614-1770-5_3

Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., ... Hodjat, B. (2019). Evolving Deep Neural Networks. Retrieved from https://arxiv.org/abs/1703.00548

Miller, G. F., Todd, P. M. & Hegde, S. U. (1989). Designing Neural Networks using Genetic Algorithms. *Proceedings of the Third International Conference on Genetic Algorithms*, 379-384. Retrieved from https://www.researchgate.net/publication/220885651 _Designing_Neural_Networks_using_Genetic_Algorithms

Montana, D. J. & Davis, L. (1989). Training Feedforward Neural Networks Using Genetic Algorithms. *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, 762-767. Retrieved from https://www.ijcai.org/Proceedings/89-1/Papers/122.pdf

Ngatchou, P., Zarei, A. & El-Sharkawi, M. A. (2006). Pareto Multi Objective Optimization. *Proceedings of the 13th International Conference on Intelligent Systems Applications to Power Systems*, 84-91. https://doi.org/10.1109/ISAP.2005.1599245

Nielsen, M. (2015). *Neural Networks and Deep Learning.* n.p.

NIST: National Institute of Standards and Technology. (2017, April 4th). The EMNIST Dataset. Retrieved from https://www.nist.gov/itl/products-and-services/emnist-dataset

NTNU: High Performance Computing Group. (n.d.). Hardware. Retrieved from https://www.hpc.ntnu.no/idun/hardware

Orr, G., B. & Müller, K.-R. (1998). *Neural Networks: Tricks of the Trade.* Berlin: Springer.

Patel, D. (1996, March 4th). Using genetic algorithms to construct a network for financial prediction. *Proceedings Volume 2664: Applications of Artificial Neural Networks in Image Processing*, (2664), 204-213. https://doi.org/10.1117/12.234258

Razali, N. M. & Geraghty, J. (2011). Genetic Algorithm Performance with Different Selection Strategies in Solving TSP. *Proceedings of the World Congress on Engineering 2011, 2*, 1134-1139. Retrieved from http://www.iaeng.org/publication/WCE2011/WCE2011_pp1134-1139.pdf

Real, E., Aggarwal, A., Huang, Y. & Le, Q. V. (2019, July 17th). Regularized Evolution for Image Classifier Architecture Search. *Proceedings of the AAAI Conference on Artificial Intelligence, 33*(1), 4780-4789. https://doi.org/10.1609/aaai.v33i01.33014780

Real, E., Moore, S., Selle, A., Saxena, S., Suematsu, Y., Tan, J. ... Kurakin, A. L. (2017, August). Large-Scale Evolution of Image Classifiers. *Proceedings of the 34th International Conference on Machine Learning,* (70), 2902-2911. Retrieved from https://dl.acm.org/doi/10.5555/3305890.3305981

Reddy, D. (2019, May 21st). Factorization Machines and Follow the Regularized Leader for dummies. Retrieved from https://medium.com/@dhirajreddy13/factorization-machines-and-follow-the-regression-leader-for-dummies-7657652dce69

Risi, S. & Stanley, K. O. (2010). Indirectly Encoding Neural Plasticity as a Pattern of Local Rules. In S. Doncieux, B. Girard, A. Guillot, J. Hallam, J. A. Meyer and J. B. Mouret, *From Animals to Animats 11* (pp. 533-543). Berlin: Springer.

Ruder, S. (2017). An overview of gradient descent optimization algorithms. Retrieved from https://arxiv.org/abs/1609.04747

Sahoo, S. (2018, August, 19th). Deciding optimal kernel size for CNN. Retrieved from https://towardsdatascience.com/deciding-optimal-filter-size-for-cnns-d6f7b56f9363

Shah, T. (2017, December 6th). About Train, Validation and Test Sets in Machine Learning. Retrieved from https://towardsdatascience.com/train-validation-and-test-sets-72cb40cba9e7

Sharma, S., Sharma, S. & Athaiya A. (2020). Activation Functions in Neural Networks. *International Journal of Engineering Applied Sciences and Technology, 4*(12), 310-316. Retrieved from https://www.ijeast.com/papers/310-316,Tesma412,IJEAST.pdf

Srivastava, N. (2013). *Improving Neural Networks with Dropout* (Master's Thesis, University of Toronto, Canada). Retrieved from http://www.cs.toronto.edu/∼nitish/msc_thesis.pdf

Stanley, K. O., Bryant, B. D. & Miikkulainen, R. (2005, December). Real-Time Neuroevolution in the NERO Video Game. *IEEE Transactions on Evolutionary Computation, 9*(6), 653-668. Retrieved from https://ieeexplore.ieee.org/abstract/document/1545941

Stanley, K. O., Clune, J., Lehman, J. & Miikkulainen R. (2019, January 7th). Designing neural networks through neuroevolution. *Nature Machine Intelligence, 2019*(1), 24-35. https://doi.org/10.1038/s42256-018-0006-z

Stanley, K. O., D'Ambrosio, D. & Gauci, J. (2009). A Hypercube-Based Encoding for Evolving Large-Scale Neural Networks. *Artificial Life, 15*(2), 185-212. https://doi.org/10.1162/artl.2009.15.2.15202

Stanley, K. O. & Miikkulainen, R. (2002). Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation, 10*(2), 99-127. https://doi.org/10.1162/106365602320169811

Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O. & Clune, J. (2018). Deep Neuroevolution: Genetic Algorithms are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning. Retrieved from https://arxiv.org/abs/1712.06567

Tanese, R., Holland, J. H. & Stout, Q. F. (1989). *Distributed genetic algorithms for function optimization* (Ph.D. Dissertation, University of Michigan, USA). Retrieved from https://dl.acm.org/doi/book/10.5555/915973

Tinós, R & Yang, S. (2007, May 15th). A self-organizing random immigrants genetic algorithm for dynamic optimization problems. *Genetic Programming and Evolvable Machines, 2007*(8), 255-286. https://doi.org/10.1007/s10710-007-9024-z

Tyantov, E. (2017, December 21st). Deep Learning Achievements Over the Past Year: Great developments in text, voice, and computer vision technologies. Retrieved from https://blog.statsbot.co/deep-learning-achievements-4c563e034257

Umbarkar, A. J. & Sheth, P. D. (2015, October). Crossover Operators in Genetic Algorithms: A Review. *Journal on Soft Computing, 6*(1), 1083-1092. http://doi.org/10.21917/ijsc.2015.0150

University of California, Irvine. (n.d.-a). Chess (King-Rook vs. King) Data Set. Retrieved from https://archive.ics.uci.edu/ml/datasets/Chess+%28King-Rook+vs.+King%29

University of California, Irvine. (n.d.-b). Yeast Data Set. Retrieved from https://archive.ics.uci.edu/ml/datasets/Yeast

Venkatachalam, M. (2019, March 1st). Recurrent Neural Networks. Retrieved from https://towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce

Wan, L., Zeiler, M., Zhang, S., LeCun, Y. & Fergus, R. (2013). Regularization of Neural Networks using DropConnect. *Proceedings of the 30th International Conference on Machine Learning, 28*(3), 1058-1066. Retrieved from http://proceedings.mlr.press/v28/wan13.html

Whitley, D. (1994, June). A genetic algorithm tutorial. *Statistics and Computing, 1994*(4), 65-85. https://doi.org/10.1007/BF00175354

Xiao, H., Rasul, K. & Vollgraf, R. (2017, August 28th). Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. Retrieved from https://arxiv.org/abs/1708.07747

Xie, L. & Yuille, A. (2017). Genetic CNN. *The IEEE International Conference on Computer Vision, 2017*, 1379-1388. Retrieved from https://arxiv.org/pdf/1703.01513.pdf

Yao, X. (1999, September). Evolving Artificial Neural Networks. *Proceedings of the IEEE, 87*(9), 1423-1447. Retrieved from https://ieeexplore.ieee.org/abstract/document/784219

Yin, L. (2017, March 9th). A Walk-through of Cost Functions. Retrieved from https://medium.com/machine-learning-for-li/a-walk-through-of-cost-functions-4767dff78f7

Zalando Research. (n.d.). Fashion-MNIST. Retrieved from https://github.com/zalandoresearch/fashion-mnist

Zeiler, M. D. (2012, December 22nd). Adadelta: An Adaptive Learning Rate Method. Retrieved from https://arxiv.org/abs/1212.5701

Zhong, Z., Zheng, L., Kang, G., Li, S. & Yang, Y. (2017, November, 16th). Random Erasing Data Augmentation. Retrieved from https://arxiv.org/abs/1708.04896

Appendices

Appendix 1 - System Manual

Document with technical descriptions of how the systems are used.

Appendix 2 - Main System Source Code

Python code for the implemented main system.

Appendix 3 - Complexity of Multi-Level Systems

Derivation of the complexity of a multi-level system, like in neuroevolution.

Appendix 4 - Merging Subpopulations System Source Code

Python code for the implemented merging subpopulations system.

Appendix 5 - Convolutional System Source Code

Python code for the implemented convolutional system.

Eivind Børstad

Evolving Deep Neural Networks using Genetic Algorithms

NTNU

Norwegian University of
Science and Technology