Matej Mnoucek

Evolving Transformer Architectures for Time Series Forecasting

Master thesis, Spring 2022

Artificial Intelligence Group
Department of Computer and Information Science
Faculty of Information Technology, Mathematics and Electrical Engineering

Abstract

Transformer-based neural network architectures have recently demonstrated state-of-the-art performance in many Natural Language Processing tasks. Furthermore, there have been successful attempts to apply the same principles to problems beyond the scope of NLP. A particularly interesting application area is time series forecasting as there is a great potential for improving the current techniques and the research to date has rarely considered Transformer-based architectures for forecasting. However, manual design and optimization of neural network architectures and their hyperparameters has proven to be difficult, time-consuming and mainly driven by trial and error.

This work explores the use of evolutionary computation to design Transformer-based architectures suitable for time series forecasting. The proposed neural architecture search system is capable of performing an automated evolution-driven search to determine the optimal architectural components as well as their parameterization and internal structure. The performance of the evolved architectures is assessed by experiments which compare the achieved forecasting accuracy with accuracies of common forecasting methods. A selection of time series benchmarks is used as a base for the comparison. The main contributions consist of the mentioned system and the final discovered architecture. The work also introduces a genetic representation for evolving Transformer-based architectures which can be seen as another contribution.

Preface

The following document presents a master thesis that was produced as the final work required to obtain Master of Science degree in Informatics at the Norwegian University of Science and Technology. First of all, I would like to thank Prof. Pauline Catriona Haddow for the time and effort dedicated to supervision. Her deep knowledge of the field, constructive feedback and endless curiosity vastly improved the quality of this work. In addition, for the entire duration of the project, I was honored to be part of NTNU CRAB Lab which is the place where most scientific debates and group brainstorming took place. Therefore, a huge thank you goes to all members of the lab, not only for their help and support but also for their everlasting optimism.

The thesis investigates the idea of using neural evolution for evolving Transformer-based architectures. More specifically, the focus is on time series and the aim is to provide a solution which achieves state-of-the-art forecasting performance.

Matej Mnoucek Trondheim, June 8, 2022

Contents

Acronyms

ANN Artificial Neural Network 32

ANOVA Analysis Of Variance 87, 88, 93, 100, 110, 115, 122

APE Added positional encoding xv, 51, 52, 76, 79

ARIMA Autoregressive Integrated Moving Average 24, 118, 121–126

ARMA Autoregressive Moving Average 24

BERT Bidirectional Encoder Representations from Transformers 46

CNN Convolutional Neural Network 18, 24, 34, 35, 56

CPPN Compositional Pattern Producing Networks 32

DL Deep Learning 33

DNN Deep Neural Network 33

DSR Design Science Research ix, 3–5

EA Evolutionary Algorithms 38, 61

GPT Generative Pre-trained Transformer 45, 51, 53

GPU Graphics Processing Unit 84

GRU Gated Recurrent Unit 21, 24

HyperNEAT Hypercube-Based Indirect Encoding for Neuroevolution of Augmenting Topologies 32

JSON JavaScript Object Notation 74

LN Layer Normalization 46, 76

LSTM Long Short-term Memory 20, 21, 24, 38, 56, 81, 118, 123–126

MAE Mean Absolute Error xvi–xviii, 24, 25, 87, 88, 93, 94, 97, 98, 101, 103, 121–126

MAM Modified attention matrix 51–53, 75, 76

MAPE Mean Absolute Percentage Error 25

MoE Mixture-of-Experts 51

MSE Mean Squared Error 25, 88

MTS Multivariate Time Series 55

NAO Neural Architecture Optimization 38

NAS Neural Architecture Search xi, 31, 33, 35, 37, 38, 42, 43, 59–61, 68, 130

NEAT Neuroevolution of Augmenting Topologies x, 31, 32

NLP Natural Language Processing 1, 2, 45, 46, 53, 61

RL Reinforcement Learning 37, 43, 61

RMSE Root Mean Squared Error xvi–xviii, 25, 87, 88, 93, 94, 97, 99, 102, 103, 121–126

RNN Recurrent Neural Network 19, 24, 43, 56, 81, 123–126

SLR Structured Literature Review 5

TCN Temporal Convolutional Network 56, 122–126

TDE Time Delay Embedding 54

List of Figures

List of Tables

Chapter 1

Introduction

This chapter describes the general background and motivation of this work (section 1.1) while also providing insight into the preliminary process that sparked interest in this area (section 1.2). Moreover, Structured Literature Review Protocol used to search for relevant literature is provided (section 1.6). In addition, the chapter presents the main hypothesis, goal and research questions (section 1.3), specifies the used research methods (section 1.4), summarizes contributions (section 1.5) and concludes with a description of the thesis structure (section 1.7).

1.1 Background and Motivation

Transformer-based neural network architectures have recently demonstrated state-of-the-art performance in many Natural Language Processing tasks. Furthermore, there have been successful attempts to apply the same principles to problems beyond the scope of NLP. A particularly interesting application area is time series forecasting as there is a great potential for improving the current techniques and the research to date has rarely considered Transformer-based architectures for forecasting. However, manual design and optimization of neural network architectures and their hyperparameters has proven to be difficult, time-consuming and mainly driven by trial and error.

1.2 Preliminary Process

As mentioned in the previous section (section 1.1), NLP has been the most prominent application area for Transformer-based architectures so far. However, these architectures have also shown promising results when applied to computer vi-

sion, audio and video processing or even biological sequence analysis problems. As Transformers represent sequence to sequence models, it appears that any data which can be encoded in a sequence can exploit the capabilities of these powerful architectures. Therefore, time series seem to be a logical candidate as they form sequences by definition and their accurate forecasting is a difficult problem to tackle.

In comparison to other areas, attempts to use Transformer-based models for time series forecasting seem quite rare. When there is such research, it usually struggles with choosing the appropriate input embedding method, positional encoding, attention model, setting hyperparameters etc. as most of these techniques were originally proposed for NLP and cannot be trivially tested and adapted.

For these reasons, it seems that there is some space for trying out automated ways of designing or adapting Transformer-based architectures to new application areas. Thus, in this thesis, it was decided to explore evolution-driven optimization of Transformer-based architectures for time series forecasting.

1.3 Goals and Research Questions

The aim of the work is to explore the potential of evolving Transformer-based architectures for time series forecasting. Therefore, this gives a rise to the main research driving hypothesis:

Hypothesis *Neural evolution is capable of designing Transformer-based neural network architectures achieving state-of-the-art performance in time series forecasting.*

The hypothesis is supported by the project goal which is accompanied by several related research questions:

Goal *Investigate evolution of Transformer-based architectures for time series forecasting.*

Research question 1 *What combination of Transformer encoder and decoder stages achieves better forecasting accuracy?*

Research question 2 *Which attention models capture long-term dependencies in time series the best?*

Research question 3 *What genotype encoding is suitable for representing Transformer-based architectures?*

Research question 4 *How does the accuracy of the evolved architectures compare to other time series forecasting methods?*

1.4 Research Method

The research captured in this document combines two research strategies, specifically, *Design Science Research (DSR)* methodology as described and summarized by Vaishnavi and Kuechler [158] and *Experiments* [115]. The role of the first strategy is to guide the creation of artifacts which constitute important contributions of this work (see section 1.5). The second strategy provides a matter for measurable quality assessment of the produced artifacts. The combination of both strategies is expected to bring concise and complete answers to the initially posed research questions. As Vaishnavi and Kuechler suggest, the research process was split into five phases:

1. **Awareness of Problem:** The first phase was focused on the definition of the research problem to study. In this case, there was no thesis topic proposal to start with, therefore, the activity was an integral part of the research effort. A detailed description of the process can be found in section 1.2. In addition, the main hypothesis, goal and research questions were also defined during this phase (section 1.3).

2. **Suggestion:** During the suggestion phase, a design framework and a solution proposal were made. The results are presented in chapter 4.

3. **Development:** The development phase was concerned with artifact creation i.e. the actual implementation of the proposed system.

4. **Evaluation:** In the evaluation phase, the second strategy i.e. Experiments starts to get involved. Experiments were defined and performed to obtain answers to the initially posed research questions (chapter 5).

5. **Conclusion:** This phase constitutes the grand finale of the research effort. The obtained results were consolidated, discussed and compared with the expected outcomes (chapter 6). Besides that, various realizations and findings occurring during the research process were assessed and addressed.

1.5 Contributions

Based on *Design Science Research Knowledge Contribution Framework* proposed by Gregor and Hevner [55], this research falls into the *Exaptation* category as it takes known solutions (Transformer-based neural network architectures) and applies them to a new problem domain (time series forecasting) in an innovative way (adaptation by evolutionary computation). See details in Figure 1.1.

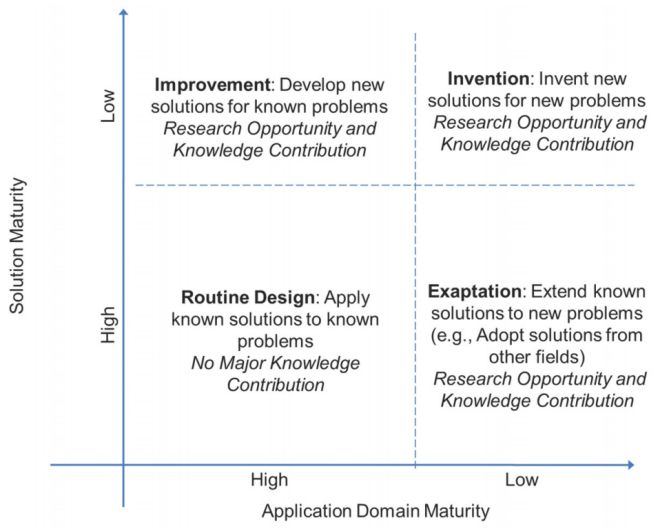

Figure 1.1: DSR Knowledge Contribution Framework [55]

The presented work makes several contributions to the field of research. For a better readability, they are organized in a list:

1. **Transformer-based architectures for time series forecasting:** As one of the main objectives of this work is to investigate automated creation of Transformer-based architectures suitable for time series forecasting, the obtained strong architectures form a contribution.

2. **Evolution-driven neural architecture search system for designing Transformer-based architectures:** To enable neural evolution to discover well-performing architectures, an evolutionary system which searches the vast space of possible solutions needs to be designed. The system presented in this work was designed as a standalone tool with reusability and modularity in mind, therefore, it constitutes a contribution.

3. **Genotype encoding for Transformer-based architectures:** To tackle the problem of evolving Transformer-based architectures beyond their known shapes and forms, a custom genotype encoding was designed to enable efficient search.

1.6 Structured Literature Review

The following section presents how relevant literature and research was discovered via *Structured Literature Review* (SLR). At first, the section shows identification of proper search queries, continues with search strategy description and finally explains the quality evaluation of the discovered research. Based on the review results, relevant and required knowledge was extracted and used during the design and evaluation of the created artifacts. The actual review was performed during Suggestion phase of Design Science Research strategy (see section 1.4).

1.6.1 Identification of Research

This subsection explains how suitable keywords and areas of relevant research were discovered. At first, an *exploratory literature search* was performed to identify search terms commonly used in connection with the research of interest. The terms which tend to retrieve the desired kind of literature were then organized into groups and used in the following *in-depth search*. Table 1.1 lists the final groups of keywords used for the construction of search queries. Short commentary and justification of each group can be found below Table 1.1.

Group 1	**Group 2**	**Group 3**
Transformer	Time series	Evolutionary algorithms
Attention	Forecasting	Neuroevolution
		Evolving
		Search

Group 4	**Group 5**
Sequence to sequence	Embedding
Neural network	Encoding
Deep learning	Positional
Architecture	

Table 1.1: Structured Literature Review – Grouping of relevant search terms used during the in-depth literature search

Group 1: The first group consists of two important keywords which frequently appear in literature concerning Transformer-based architectures. Hence, the presence of this group biases the search towards relevant papers.

Group 2: The second group reduces the vast search space further by limiting the search scope to time series and relevant forecasting methods.

Group 3: The next group brings the evolutionary computation aspect to the mix. It contains several keywords commonly found in papers presenting research from this area.

Group 4: The fourth group includes more general and potentially relevant terms from machine learning to ease the strong specificity and narrow focus enforced by the previous groups.

Group 5: The last group presents encoding and embedding keywords which represent important concepts closely tied to Transformer-based architectures.

During both literature searches, the online libraries listed below were used to retrieve publications for the provided search queries:

- *Google Scholar*
- *ACM Digital Library*
- *Science Direct*
- *Engineering Village*
- *Semantic Scholar*
- *IEEE Xplore Digital Library*

In the first iteration of *in-depth search*, the search query was constructed by combining keywords from all groups. Basically, all terms from each individual group were joined by OR operator and then the resulting strings were concatenated by AND operator. The query looked as follows:

(Transformer OR Attention) AND (Time-series OR Forecasting) AND (Evolutionary algorithms OR Neuroevolution OR Evolving OR Search) AND (Sequence to sequence OR Neural network OR Deep learning OR Architecture) AND (Embedding OR Encoding OR Positional).

However, the employed query did not perform so well. It turned out that it becomes too specific which hinders the discovery of many relevant papers. Therefore, it was necessary to execute several additional independent searches which did not involve all groups at once, but only a carefully crafted subset of them. The list below shows the combinations which yielded the best results:

- *(Group 1) AND (Group 2) AND (Group 3)*
- *(Group 1) AND (Group 3) AND (Group 4)*
- *(Group 1) AND (Group 4) AND (Group 5)*
- *(Group 1) AND (Group 2)*
- *(Group 1) AND (Group 5)*

Some of the retrieved high impact research papers referenced literature that was not discovered by the search yet still relevant. These sources were either influential papers themselves but for a broader area of research, or showcased interesting ideas potentially useful for this work. For that reason, the *forward snowballing* method [166] was employed to capture such literature.

1.6.2 Selection of Primary Studies

The group of primary studies constitutes a subset of all found literature which satisfies explicit criteria of relevance. The search queries presented in subsection 1.6.1 already managed to filter out most of the irrelevant research, however, there still are studies which do not satisfy the desired level of relevance.

For that reason, the studies obtained during all searches were also subjected to evaluation based on inclusion criteria. Each source must satisfy at least one of the provided criteria otherwise it is discarded. Satisfying more than one criteria is preferable but not strictly required. This selection process significantly reduced the number of considered studies down to a manageable subset. The used inclusion criteria are listed in Table 1.2.

ID	Inclusion criteria
IC 1	The study's main concern are Transformer-based architectures.
IC 2	The study's main concern is time series forecasting.
IC 3	The study concerns evolutionary computation.
IC 4	The study focuses on neuroevolution.
IC 5	The study focuses on neural architecture search.
IC 6	The study concerns embedding or encoding in relation to Transformer-based architectures.

Table 1.2: Structured Literature Review – Inclusion criteria

1.6.3 Quality Assessment

Finally, the remaining studies were subjected to quality assessment process to determine preferable and reliable research. Each study was evaluated based on the extent to which it satisfies each quality criteria listed in Table 1.3. The criteria are mostly of a general nature measuring the overall quality of each study from academic perspective. However, this time the evaluation was not used to exclude any studies but to obtain a quality score internally used to rank and order the retrieved studies.

ID	Quality criteria
QC 1	The study has a clear statement of the aim of the research.
QC 2	The study is put into context with other studies and research.
QC 3	The study conducts a set of documented experiments and presents their findings.
QC 4	The study contains a discussion of the results.

Table 1.3: Structured Literature Review – Quality criteria table

1.7 Thesis Structure

The document is divided into several chapters:

Chapter 2 – Background Theory: introduces a set of theoretical knowledge needed to reason about concepts presented in this thesis.

Chapter 3 – Related Work: presents and discusses the relevant research collected during Structured Literature Review.

Chapter 4 – Method and System Design: introduces a system design framework, explains the employed techniques and solutions and provides further details about the design and implementation process.

Chapter 5 – Experiments and Results: describes and conducts experiments to validate the developed system and evaluate the evolved architectures. The results are further analyzed by methods of statistical analysis.

Chapter 6 – Conclusion and Future Work: assesses the final outcomes of the work and discusses them in the light of the initially posed hypothesis, goal and research questions. Apart from that, limitations of this work and possible future work proposals are presented.

Chapter 2

Background Theory

In this chapter, the relevant background theory is presented in the amount needed to understand concepts introduced in later chapters. The first section presents the basics of *evolutionary algorithms* (section 2.1). The next section defines *artificial neural networks* and provides examples of relevant architectures (section 2.2). The following sections introduce *time series* (section 2.3) and basic *time series forecasting methods* (section 2.4). The last section describes *Transformer architecture* (section 2.5).

2.1 Evolutionary Algorithms

Evolutionary computing is a special area of computing inspired by the process of natural evolution i.e. survival of the fittest. In other words, an environment can host only a limited number of individuals, hence, individuals that manage to adapt to the environment better have a greater chance of survival [41]. *Evolutionary algorithm* constitutes a synthetic recreation of the natural evolution process. In contrast to natural evolution, artificial evolution is an optimization and/or design process that attempts to find solutions to predefined problems [47].

There are many variants of evolutionary algorithms but the core principles mostly stay the same. A population of candidate solutions is initialized to provide an environment for evolution. The population is then subjected to an iterative process of artificial evolution. At first, parents are selected for reproduction based on some predefined fitness measure. Next, the selected parents are recombined and mutated to produce offspring. The offspring is then evaluated based on the same fitness measure and forced to compete for survival with their parents. After survivors are determined, they are added back to the general population. The

whole cycle repeats until a termination condition is reached. For illustration, the cycle is shown in Figure 2.1. The optimization process is driven by two main factors: *recombination and mutation* which creates diversity in the population and constructs novel candidate solutions and *selection* which acts as a force improving the general quality of solutions within the population.

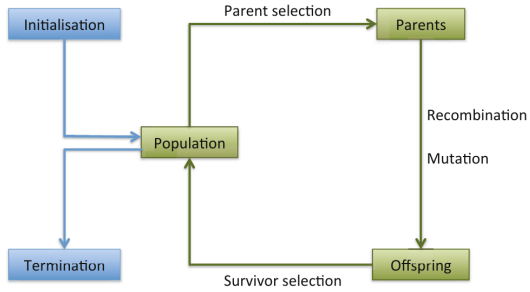

Figure 2.1: The general process of evolutionary algorithms [41].

2.1.1 Representation

One of the first tasks when building a genetic algorithm is choosing an appropriate *solution representation*. The representation is commonly referred to as *phenotype* while its encoding in a genetic algorithm is called a *genotype*. Genotype is sometimes also called *chromosome*. Mapping from genotype space to phenotype space needs to be explicitly defined to enable transformation from one into another. In the following paragraph, a few common representations are given as examples.

The *binary representation* is one of the simplest available. The genotype is represented as a list of zeros and ones i.e. a bit string. Another popular option is *integer representation* in which the binary numbers are replaced by integers. Similarly, *real-valued representation* uses floating-point values instead of bits or integers. There also exist more complex representations such as *tree-based representations*.

2.1.2 Population

The sole purpose of a population is to hold a set of candidate solutions. The main properties of populations are: the selected genotype representation and the

number of candidates it should hold i.e. the population size. There exist two common population models: *generational* and *steady-state* model. In a generational model, each genotype survives for exactly one generation. The entire population is replaced by genotypes selected from among the new offspring. In contrast, in a *steady-state* model the population is not replaced all at once, only several individuals are. The portion of the population which gets replaced is often called the *generational gap*.

2.1.3 Fitness Function and Selection

The *fitness function* describes the quality of candidate solutions. Frequently, it is merely a function which labels genotypes with a single numerical value representing their quality. Usually, the higher the number is, the better the solution. The purpose of fitness functions is to allow comparison of different solutions and to enable selection. Selection usually happens twice during one iteration of evolution loop. The first time, it is for *parent selection* and later for *survivor selection*.

The objective of the parent selection is to select high quality candidate solutions which will undergo further reproduction. A basic approach is *fitness proportional selection* in which solutions are selected based on their absolute fitness. However, this approach often causes premature convergence as good solutions quickly take over ruining diversity of the population. Therefore, different selection methods usually based on probabilities are preferred. One example is *rank-based selection*. Another popular method is *tournament selection*. The tournament selection allows selection of parents without calculating fitness for the entire population. It is based on sampling only a small portion of the population within which the selection takes place.

For the survivor selection, most of the parent selection strategies can be used as well. However, survivor selection often encapsulates not just selection but also a replacement operation as there is a need to identify older solution which will be replaced by the new survivor. An example strategy is *age-based replacement* which marks the oldest solutions in population for replacement or *fitness-based replacement* where the deciding factor is the fitness of the old and new solution.

2.1.4 Mutation and Crossover

The role of *mutation* and *crossover* is to create new candidate solutions based on already existing ones. Mutation is an unary operator which takes in a candidate solution and returns its mutated (modified) version. The mutated version is known as a *child* or *offspring*. The operator is always stochastic and introduces novelty to the solutions. In contrast, crossover is a binary operator which

consumes two so-called *parents* and combines their information into one or more offspring. Crossover is also a stochastic operator but it does not introduce any novelty. Only the information available in parents can be present in offspring. Both operators depend on the chosen genotype representation. In the following paragraphs, we will present a few example operators for some of the representations described in subsection 2.1.1.

For the binary representation, there exists a common and simple mutation operator. The operator treats every bit in the string separately and with a certain probability each individual bit gets flipped. When it comes to crossover, there is a few alternatives available. The first option is to use *n-point crossover*. At first, the operator randomly generates n split point indices from the $[1, l-1]$ range where l is the length of the chromosome. Then it alternates between the first and second parent and chooses genotype segments defined by the split points. The process is illustrated in Figure 2.2. For $n = 1$ special case, the operator is called *one-point crossover*. *Uniform crossover* is another popular crossover operator. This one resembles the presented mutation as it also treats each bit independently and randomly picks a parent from which the bit gets inherited.

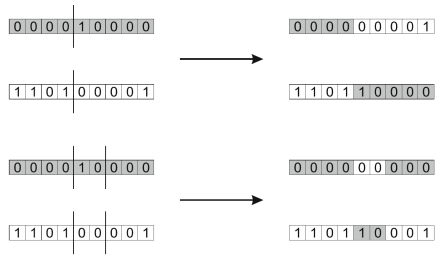

Figure 2.2: Two examples of n-point crossover operator for $n = 1$ and $n = 2$ [41].

In the case of integer-based representation, the appropriate mutation operator depends on the type of represented values. *Random resetting* is used when the values represent cardinal attributes. Basically, a set of possible values is defined beforehand and when the mutation occurs, a new value is drawn from the set. If the values represent ordinal attributes, *creep mutation* is more suitable. This operator with a certain probability adds or subtracts a small value from each number in the genotype. When it comes to crossover, most of the approaches used for binary representation are also applicable to integer-based representation.

2.2 Artificial Neural Networks

Artificial neural networks were inspired by neural networks which make up human brain and are to a large extent responsible for human cognitive abilities [51]. The fundamental building block of these networks is an *artificial neuron* which also draws inspiration from its biological counterpart. The real neurons are complex cells of similar structure and behaviour which can be specialized for various different tasks. They work as information processing units. Essentially, electrical signals travel to neuron cell body via several *dendrites*. Inside the body, the signals are added together and compared to a *threshold*. If the resulting signal is greater than the threshold, a new signal is sent through neuron's *axon*. As stated earlier, neurons build up whole networks and they do so by binding together via dendrites and axons. A simplified illustration of a biological neuron can be seen in Figure 2.3.

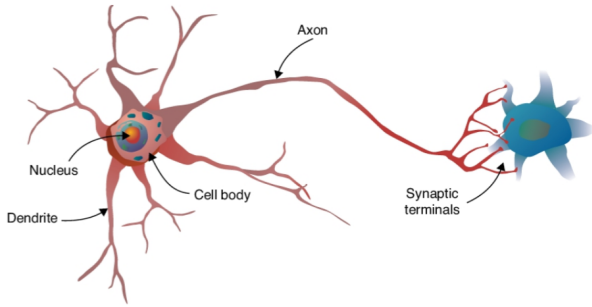

Figure 2.3: An illustration of a biological neuron describing its main parts [51].

The foundations for mathematical model of biological neurons were laid by *perceptron* [132]. A diagram is shown in Figure 2.4. Every perceptron input is a single numerical value multiplied by another value called *weight*. The scaled inputs are then added together and the result is passed through an *activation function*. There also exists a special input called *bias* which is in fact a scalable constant.

Like biological neurons, artificial neurons can also be linked together to form a network. However, in artificial neural networks we generally group neurons to *layers* first. One layer is enough to produce a valid and complete network. Such layer is usually called *output layer*. If a neural network has more than one layer, the additional layers are called *hidden layers* [51]. If a network has one or more

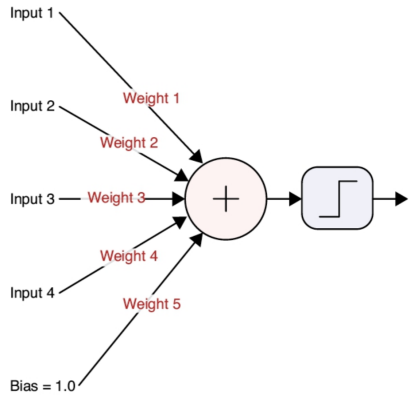

Figure 2.4: A single perceptron with four inputs, bias and activation function [51].

hidden layers, it is commonly referred to as *deep neural network*. Connections between neurons in different layers can be realized in multiple ways. A common configuration is that each neuron in one layer receives input from each neuron in the previous layer. Such configuration is called *fully-connected layer* or *fully-connected neural network*. An example of a deep neural network made from fully connected layers can be seen in Figure 2.5.

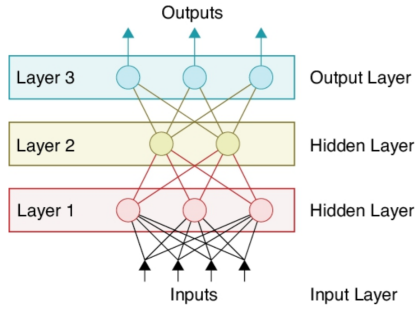

Figure 2.5: An example of deep neural network with one hidden layer [51].

Activation functions are non-linear functions which constitute an important part of artificial neural networks. Their role is to introduce non-linear operations to the chain of so far only linear operations (multiplication and addition within

neurons) to prevent networks from collapsing into a mathematical equivalent of a single neuron [51]. There are many different options available. A group of activation functions which are composed of several linear segments is called *piecewise linear activation functions*. The most prominent representative of this group is *Rectified Linear Unit* or *ReLU* and its derivatives such as *leaky ReLU* or *parametric ReLU*. Other commonly used functions are *Sigmoid*, *Tanh*, *Swish* or *ELU* which belong to the *smooth activation functions* category. All the presented functions and a few other alternatives are visualized in Figure 2.6.

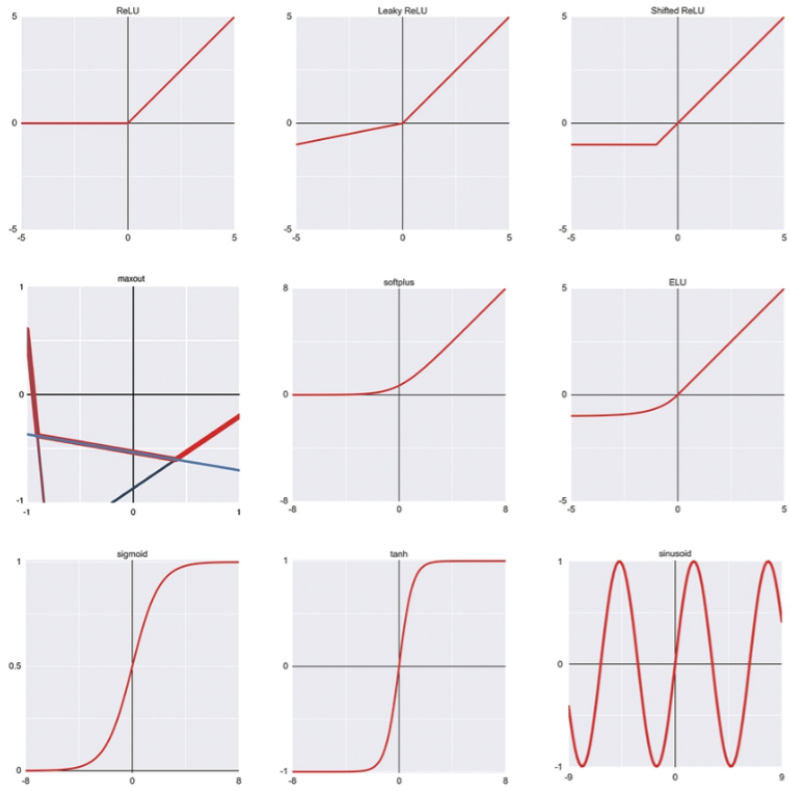

Figure 2.6: Examples of various activation function alternatives [51].

So far, artificial neural networks have been described as a simple collection of artificial neurons. However, in order to make them work, they also need to be *trained* i.e. their weights need to be tuned to minimize the so-called network *loss*

or prediction error. Arguably, the most common training method is *backgropagation*. This technique utilizes *gradient descent* to decide how to adjust network weights in order to improve the loss [54]. Additionally, *neuroevolution* has also been used as an alternative to backpropagation [47]. To train a neural network, we also need a labelled *dataset* which contains training examples. During the training process, the network is fed with training examples which provide basis for determining its loss. This kind of training is known as *supervised learning*.

A neural network can suffer from *overfitting*, if, simply put, is trained for too long. Essentially, the network learns the dataset used for training so well that it becomes tuned only to that data and performs poorly when applied to new data. The techniques used to delay the onset of overfitting are called *regularization methods*. A common regularization method is called *dropout*. With a certain probability, dropout randomly disables neurons within the network. This makes it harder for the network to learn the training dataset precisely. There also exist more sophisticated regularization methods such as *batch normalization* or *layer normalization*.

2.2.1 Convolutional Neural Network

Convolutional neural networks (CNN) have become a popular choice for processing grid-like data such as images as they are capable of learning hierarchies of spatial features ranging from low to high level patterns [174]. Their design was inspired by the model of visual cortex commonly used in neuroscience. In comparison to simple fully-connected networks, these networks simply prepend fully-connected layers with additional layers performing *convolution* and *pooling*.

If to use a fully-connected layer for image processing directly, then for an image with n pixels, there will be n neurons with n^2 weights needed. For larger images, this quickly becomes infeasible. In addition, fully-connected layers do not take local pixel adjacency into account. Therefore, a solution was proposed where each neuron in a layer receives input only from a small *local region* of the image [134]. Also, as we expect images to exhibit *spatial invariance*, we can keep the same region connection weights for each neuron [54]. This modification essentially transforms neurons into feature detectors which are capable of detecting the same features no matter where they appear in the input image.

In a nutshell, Convolutional neural network is a network that relies on spatially local connections and uses a pattern of weights replicated across neurons in each *convolutional layer* [134]. The replicated pattern of weights is commonly known as *kernel* and the process of applying this kernel to an image is called *convolution*.

Convolution is denoted by $*$ operator, so a simple 1D convolution $z = x * k$ is defined as follows:

$$z_i = \sum_{j=1}^{l} k_j x_{j+i-(l+1)/2} \tag{2.1}$$

where x is an input vector and k is a kernel vector of size l. An example of the convolution process is shown in Figure 2.7 where $[1, -1, 1]$ kernel is applied to $[5, 6, 6, 2, 5, 6, 5]$ input vector. The example also introduces a new concept called *stride*. If you look closely, the kernel is applied around every second number which corresponds to $stride = 2$. However, stride greater than 1 has a side effect which is reducing the size of the original input.

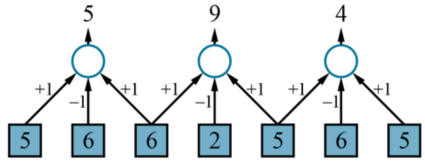

Figure 2.7: 1D convolution with $[1, -1, 1]$ kernel over $[5, 6, 6, 2, 5, 6, 5]$ vector [134].

Apart from convolutional layers, *pooling layers* are also used. Pooling is an operation used to summarize results of several adjacent neurons from the previous layer [134], in essence, downsample the input. There are two main types of pooling: *max-pooling* and *average-pooling*. The result of max-pooling is the maximum value present in the input whereas average-pooling returns the average value. Aside from reducing the input dimensions, the second objective of polling is to progressively decrease the number of subsequent learnable weights.

The rest of the network is composed of one or more fully-connected layers. The last layer often is a *softmax* layer which outputs probabilities for image classification. However, the possible configurations of convolutional neural networks are not limited to the softmax case only.

2.2.2 Recurrent Neural Network

Recurrent neural networks (RNN) are networks explicitly designed to handle processing of ordered sequences [51]. They introduced a concept of *recurrent cells* with *hidden states* where hidden states hold a compressed representation

of the inputs processed so far i.e. serve as memory. These cells are commonly classified as *recurrent layers*. Similarly to Convolutional neural networks, weight sharing is also utilized. However, this time it is across different parts of the network which allows generalization to inputs of various lengths [54]. Thanks to the recurrent dependencies in the network, predictions need to be done iteratively in *time steps*. After one input element is processed, the hidden state needs to be updated before the next input processing can take place. For each input, there is an output produced. However, in some use cases the intermediate outputs are discarded as only the final output is needed. An illustration of the network structure is shown in Figure 2.8.

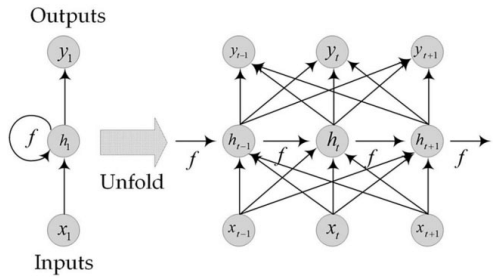

Figure 2.8: The folded (left) and unfolded (right) illustration of recurrent neural network structure. x_t represents input sequence, h_t hidden states and y_t elements of the potential output sequence [152].

For training of recurrent neural networks, a special version of backpropagation called *backpropagation through time* is used. Unfortunately, the training process often suffers from *vanishing* or *exploding gradients* problem which occurs when gradients become extremely small or large during backpropagation. These conditions significantly hinder further learning capabilities of the network. To combat this problem, improved versions of recurrent cells were proposed.

A popular alternative to recurrent cells is *Long short-term memory* (LSTM) cell which in addition to hidden state (short-term memory) incorporates additional memory for long-term information storage (long-term memory). In comparison to recurrent cell, LSTM differs in two fundamental ways: 1.) the same long-term memory is cloned from time step to time step which reduces accumulation of gradient multiplications and leads to fewer vanishing or exploding gradients, 2.) cell structure includes three *gating units* (or gates) which regulate the flow of information within the cell [51]. The *input gate* determines which elements in

the long-term memory get updated by input data from the current time step. The *output gate* selects long-term memory elements which should be moved to short-term memory. Finally, the *forget gate* decides which long-term memory elements are erased i.e. set to zero. There also exist a slightly modified variant of LSTM called *Gated recurrent unit* (GRU).

2.3 Time Series

A *time series* is defined as a set of observations x_t, where each observation was taken at a specific time t. *Discrete time series* is time series for which the set of times when observations were made, T_0, is a discrete set. In contrast, *continuous time series* are produced when observations are continuously recorded over a time interval e.g. when $T_0 = [0, 1]$ [15]. An example of discrete time series can be seen in Figure 2.9. Literature also differentiates between time series that is *univariate* i.e. consist of only one observed variable varying over time and *multivariate* which follows many variables [22].

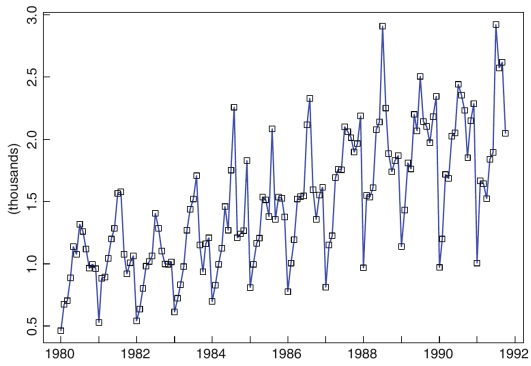

Figure 2.9: An example discrete time series showing sales of The Australian red wine between years 1980 and 1991 [15].

2.3.1 Time Series Components

Time series are commonly affected by four main *components* (sometimes also called *variations*): *trend*, *seasonality*, *cyclicity* and *irregularities* [101]. *Trend* is a long-term increase or decrease in values of observations. Increase in values is

referred to as *upward trend* and decrease as *downward trend*. *Seasonal component* captures when a certain pattern periodically repeats after a known and fixed period. Seasonality usually refers to variations within one year i.e. patterns that repeat monthly, weekly, daily etc. The concept is illustrated in Figure 2.10. *Cyclic component* also describes a recurring pattern but not necessarily after a fixed period of time. Also, the period might be longer than one year. Finally, the remaining series of residual value changes are known as *irregular component* [112].

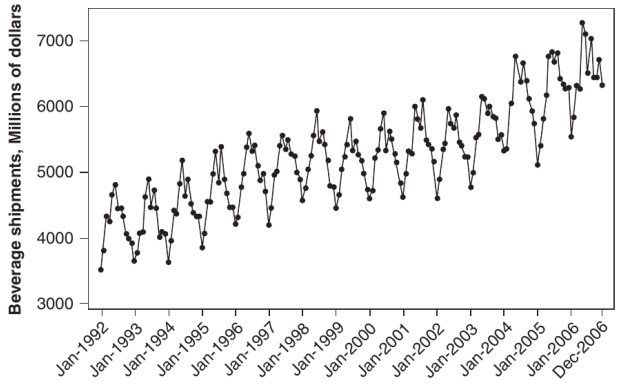

Figure 2.10: Monthly beverage shipments time series showing clear signs of seasonality [112].

2.3.2 Decomposition and Smoothing

To represent time series as a composition of components, three types of *decomposition* models are commonly used. The first model is called *Multiplicative model* and is defined as follows:

$$Y_t = T_t \times S_t \times C_t \times I_t \tag{2.2}$$

where Y_t represents the original observation value at t, T_t the trend component value at t, S_t the seasonal fluctuation, C_t the cyclic fluctuation and I_t the irregular variation at t. Similarly, *Additive model* is defined as:

$$X_t = T_t + S_t + C_t + I_t \tag{2.3}$$

The third model is known as *Mixed model* and is a combination of both previous models [32]. To identify trend of a time series, various *smoothing methods* are

available. Smoothing attempts to remove short-term fluctuations and leave out the long-term trend component [15]. One popular method is *Moving average*:

$$SM_{ma}(x_t) = \frac{1}{2k+1} \sum_{i=-k}^{k} x_{t-i} \tag{2.4}$$

where k is the order of moving average i.e. the amount of applied smoothing. Simply put, trend estimate at time t is determined by the average of observation values within the window defined by k. Another common method is *Exponential smoothing* as defined below:

$$SM_e(x_t) = \sum_{i=0}^{\infty} \alpha(1-\alpha)^i x_{t-i} \tag{2.5}$$

where α is a smoothing constant such that $0 < \alpha < 1$.

2.4 Time Series Forecasting

The simplest form of forecasting is *point forecast* which is an attempt to estimate a future value of some variable of interest. These estimates are rarely perfect, hence, include some amount of *forecast error*. Therefore, we sometimes use *prediction interval* instead of a single value. A prediction interval has upper and lower bound between which the estimated value is expected to lie with some defined probability [22]. Other important aspects of forecasting are *forecast horizon* and *forecast interval* [112]. The forecast horizon is the number of future periods which we intend to produce estimates for and forecast interval is the period of time between individual forecasts. The next few sections describe the most common models for time series forecasting.

2.4.1 Classical Models

Classical models are mostly statistical methods used to forecast an observation at $t + 1$. However, these methods usually require time series to be *stationary*. *Stationary time series* is time series which shows no systematic change in mean, no systematic change in variance and that had all strictly periodic variations removed [112].

Firstly, *Moving average model* and *Autoregressive model* are defined [66] as they constitute important building blocks for the later presented models. Moving average model of order p (MA(p)) is defined as:

$$x_t = c + \varepsilon_t + \phi_1\epsilon_{t-1} + \phi_2\epsilon_{t-2} + \cdots + \phi_p\epsilon_{t-p} \tag{2.6}$$

where ϵ_t is white noise, ϵ_{t-i} are *residual errors* and ϕ_i are weights. Residual error is the difference between actual value and predicted value calculated as $\epsilon_t = x_t - \hat{x}_t$. Similarly, Autoregressive model of order q (AR(q)):

$$x_t = c + \varepsilon_t + \theta_1 x_{t-1} + \theta_2 x_{t-2} + \cdots + \theta_q x_{t-q} \tag{2.7}$$

where ϵ_t is white noise, x_{t-i} are past values of the observed variable and θ_i are weights. The first combined model is *ARMA(p,q)* of order p, q which is simply a combination of MA(p) and AR(q). The equation looks as follows:

$$x_t = c + \varepsilon_t - \theta_1 \epsilon_{t-1} - \theta_2 \epsilon_{t-2} - \cdots - \theta_p \epsilon_{t-p} + \theta_1 x_{t-1} + \theta_2 x_{t-2} + \cdots + \theta_q x_{t-q} \tag{2.8}$$

A variant of ARMA which has the benefit of being able to handle non-stationary time series is called ARIMA. The model adds differencing to the mix and is usually defined as ARIMA(p,d,q) where d is the degree of first differencing. Shown as an equation:

$$x'_t = c + \varepsilon_t - \theta_1 \epsilon_{t-1} - \theta_2 \epsilon_{t-2} - \cdots - \theta_p \epsilon_{t-p} + \theta_1 x'_{t-1} + \theta_2 x'_{t-2} + \cdots + \theta_q x'_{t-q} \tag{2.9}$$

2.4.2 Neural Network Models

Neural network models have also been used to tackle forecasting problems. In order to do so, the employed network needs to be able to consume and process sequential data. RNNs together with LSTMs and GRUs are some of the possible candidates. Thanks to their recurrent connections they are capable of capturing temporary dependencies in time series data and thus provide reasonable estimates. Another option is to use CNNs which have also shown potential in time series forecasting.

2.4.3 Model Evaluation

For model evaluation and comparison, it is important to consider an appropriate performance metric. A very common and essential metric is the accuracy of future forecasts. The set of standard measures of accuracy [66] includes *mean absolute error* defined as:

$$MAE = \frac{1}{n} \sum_{i=0}^{n} |x_i - \hat{x_i}| \tag{2.10}$$

where n is the number of observations, x_i an actual observation value and $\hat{x_i}$ the value predicted by a model. Mean absolute error is simple but a *scale-dependent* metric. Scale-dependent means that the error is expressed in the units of the underlying series which makes it hard to compare a model across different

series [67]. Other popular scale-dependent metrics are *mean squared error* and *root mean squared error* which are in fact quite similar:

$$MSE = \frac{1}{n}\sum_{i=0}^{n}(x_i - \hat{x_i})^2 \quad RMSE = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(x_i - \hat{x_i})^2} \tag{2.11}$$

In comparison to MAE, MSE and RMSE are more sensitive to outliers i.e. weight large errors more than the smaller ones. The fourth presented method is *mean absolute percentage error* which belongs among *percentage error metrics*. These metrics are scale independent, therefore, safe to use for comparisons across different series. MAPE is calculated as follows:

$$MAPE = \frac{1}{n}\sum_{i=0}^{n}|\frac{x_i - \hat{x_i}}{x_i}| \tag{2.12}$$

The resulting value is in percentage.

2.5 Attention and Transformer Architecture

Attention mechanism was initially designed to mitigate vanishing gradients problem of Recurrent neural networks in *sequence to sequence models*. However, Vaswani et al. [159] discovered that *attention* can also be used in non-recurrent models. Their novel architecture called *Transformer* has surpassed several recurrent models in terms of predictive performance.

2.5.1 Sequence to Sequence Models

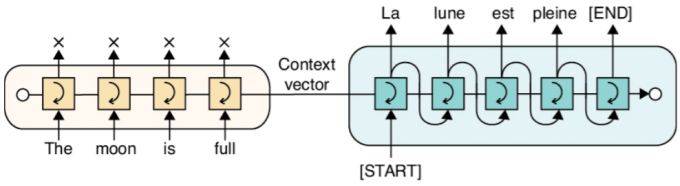

Figure 2.11: An example of sentence translation by sequence to sequence model. The encoder on the left takes in a sentence and encodes it into a context vector (hidden state). The decoder on the right uses the hidden state to produce the translated sentence [51].

Sequence to sequence model was first proposed by Sutskever et al. [151] as a generic solution for sequence to sequence mapping. The main motivation was the fact that one sequence does not necessarily map to another sequence on element to element basis. An example is e.g. language translation where relevant words in the input sentence might occur at different places in the translated sentence. The lengths of the sentences might differ too. For these reasons, sequence to sequence models consist of two stages: *encoder* and *decoder*. Encoder processes input and encodes its representation into a fixed length hidden state (sometimes also called *context vector*). In contrast, decoder takes the hidden state as input and produces the desired output sequence. A simple illustration is provided in Figure 2.11. These models are often *autoregressive* which means that in addition to the provided input sequence, their decoder also consumes its previous outputs to further generate new elements.

2.5.2 Attention Mechanism

Attention mechanism or simply *attention* is an extension originally proposed for sequence to sequence models. In their work, Bahdanau et al. [5] and Luong et al. [100] argued that the propagation of hidden state constitutes a bottleneck which hinders performance of recurrent models. They suggested allowing the decoder to bind directly to different parts of the encoder to facilitate easier information flow and allow for bypassing the hidden state entirely if needed. As demonstrated in their experiments, this technique has improved performance of the tested models. An illustration of this attention binding is shown in Figure 2.12.

2.5.3 Transformer Architecture

Inspired by the improvements brought by attention mechanism, Vaswani et al. [159] proposed an architecture without recurrent connections relying solely on attention. Their experiments showed that the model can perform better than recurrent models while being more parallelizable and requiring less time to train. The architecture is called *Transformer* and is built as autoregressive sequence to sequence model i.e. consists of separate encoder and decoder stages. The complete architecture diagram is shown in Figure 2.15.

The job of the encoder stage is to process and encode the input sequence (*source sequence*). Transformer's encoder is made up of 6 identical stacked layers. Each layer consists of two sub-layers placed in the following order: *multi-head attention* and fully-connected *feed forward* neural network. In addition, there are residual connections allowing bypass of each sub-layer followed by layer normalization.

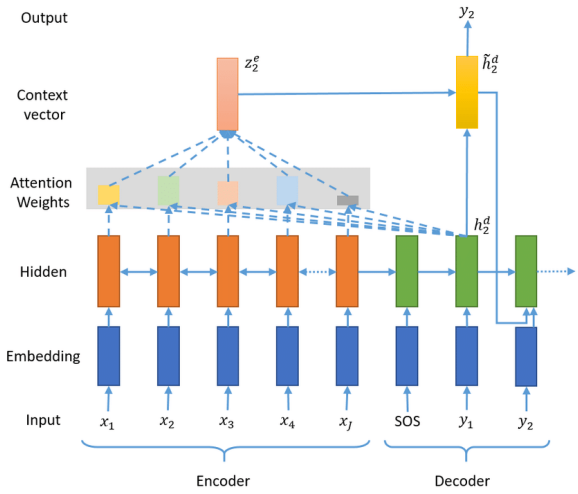

Figure 2.12: Bindings from decoder (green) to encoder (orange) via attention in sequence to sequence models. Each attention link has its own set of trainable weights [141].

The decoder stage uses its previously generated outputs (*target sequence*) as input sequence in order to generate a new target sequence element. It is formed as a stack of 6 layers too. However, apart from the same sub-layers as the encoder stage has, there is a third *multi-head attention* sub-layer placed in between the two existing sub-layers. This sub-layer allows attention to bind to the outputs of the encoder stack. Residual connections with layer normalization are present around sub-layers in the same way as they are in the encoder stage. Importantly, the first multi-head attention sub-layer is *masked* to prevent it from attending to subsequent target sequence tokens. This ensures that prediction of a new target sequence element at position i can only attend to known outputs at positions less than i.

In the context of Vaswani et al. [159] work, *attention* is used to determine relations between elements in both the source sequence and the target sequence. *Self-attention* is a special case of attention which considers only a single sequence and the relations between elements in it. Figure 2.13 illustrates the concept on a simple example.

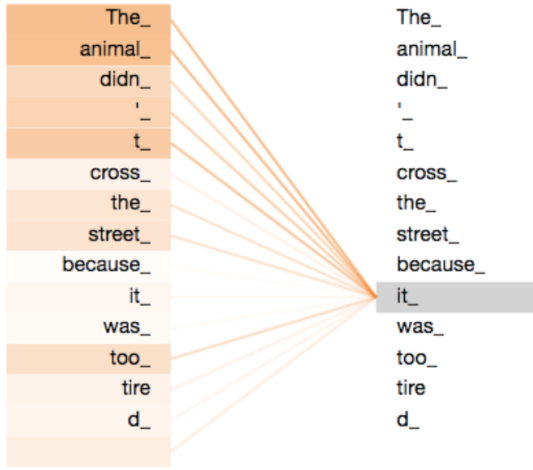

Figure 2.13: A simple example of self-attention in a sequence of words: *What does it refer to in this sentence?* The darker the color on the left is, the higher the attention score i.e. the more focus the corresponding word gets [3].

To calculate self-attention, *query (Q)*, *key (K)* and *value (V)* vectors are created by linearly transforming the input representation of each element. Then, attention scores are determined by taking a dot product between query of the considered element and keys of all other elements. Next, the obtained scores are passed though softmax to obtain attention weights. Finally, the weights are multiplied by value vectors to obtain the actual attention values. In a similar way, attention can be calculated for cases when the query comes from a different sequence than keys and values e.g. when attention between source and target sequence is considered. In other words, attention is calculated as a weighted sum of values where each weight is determined by a compatibility function describing the relationship between the corresponding query and key. This particular attention calculation is called *scaled dot-product attention* as shown in Figure 2.14 and defined in Equation 2.13 where d_k represents the dimension of the query vector.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.13}$$

In addition, Vaswani et al. [159] found beneficial to use multiple ($h = 8$) *attention heads* to linearly project queries, keys and values multiple times, each time with a different learned projection. This allows a Transformer model to express several

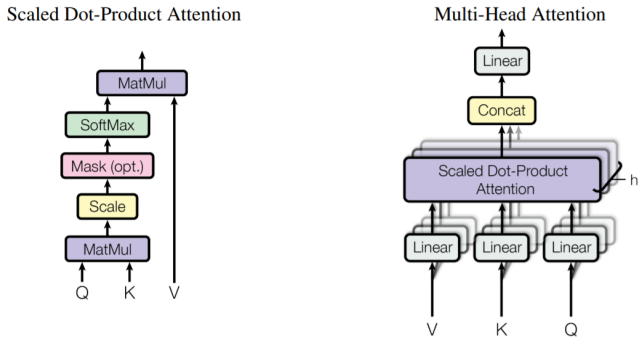

Figure 2.14: Left: *Scaled dot-product attention.* Right: *Multi-head attention* which allows parallel execution of multiple independent attention calculations [159].

learned attention functions in parallel as demonstrated in Equation 2.14. The projections are represented by parameter matrices: QW_i^q, KW_i^k and VW_i^v. An illustration of *multi-head attention* concept is provided in Figure 2.14.

$$\begin{aligned} MultiHead(Q,K,V) &= concat(head_1, head_2, \dots head_h) \\ where\, head_i &= Attention(QW_i^q, KW_i^k, VW_i^v) \end{aligned} \tag{2.14}$$

Before a sequence can enter a Transformer model, it needs to be preprocessed by *input embedding*. Input embedding converts each sequence element into a vector representation of dimension d_e which is then used by the model. d_e is often referred to as *embedding dimension*. As the model contains no recurrence, information about positions of elements in the input sequence is lost. For that reason, the model adds *absolute positional encoding* to the input embedding to be able to reason about element distances. More specifically, sine and cosine functions of different frequencies were used. The actual functions are shown in Equation 2.15.

$$\begin{aligned} PE_{(pos,2i)} &= sin(pos/10000^{2i/d_{model}}) \\ PE_{(pos,2i+1)} &= cos(pos/10000^{2i/d_{model}}) \end{aligned} \tag{2.15}$$

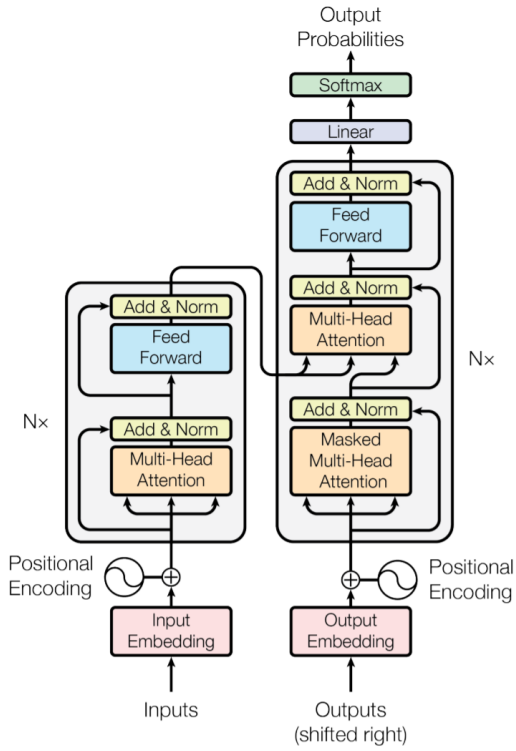

Figure 2.15: *Transformer* architecture diagram. The model consists of two stages: *encoder* and *decoder*. Encoder is made by stacking several *encoder layers*. Each layer contains two sub-layers: *Multi-head attention* and *Feed forward* sub-layer. Similarly, decoder is made up of *decoder layers* and contains the same two sub-layers. In addition, decoder includes another multi-head attention sub-layer which performs attention over the outputs of the encoder stack. The three arrows pointing to the bottom of each multi-head attention sub-layer represent *key*, *value* and *query* vectors (in this order). Finally, the input sequence is provided in a form of *input embedding* with *positional encoding* added to it [159].

Chapter 3

Related Work

In the following chapter, research relevant for the scope of this work is presented. The chapter starts with an introduction to evolution of artificial neural networks (section 3.1) and puts emphasis on *Neural Architecture Search* (section 3.2). Next, section 3.3 discusses the application of NAS to Transformer-based architectures. In section 3.4, improvements and modifications proposed for Transformer architectures are reviewed. Finally, as this work focuses on time series, section 3.5 reviews research concerning the use of Transformers for forecasting, section 3.6 reviews state-of-the-art forecasting methods and section 3.7 presents time series forecasting benchmarks.

3.1 Neuroevolution

The artificial evolution of neural networks, also termed *Neuroevolution*, has been widely studied in literature. The early works focused primarily on evolving neural network weights, but the focus quickly shifted towards more ambitious goals such as evolving network topology alongside the weights [179, 57]. The technique has proven successful in a variety of applications [113, 53].

Noticeable advancements came to the field with the introduction of *Neuroevolution of Augmenting Topologies* or NEAT for short [147]. NEAT is a method for genetically encoding and evolving both the topology and weights of a neural network. It does so by starting with a minimal topology and incrementally adding more genes to evolve more complex topological structures. This approach biases the search towards minimal solutions which in turn reduces the number of parameters to optimize. An important contribution of the work was solving the problem of *Competing Conventions* i.e. that the same solutions might get

represented in different ways [111]. Competing Conventions pose a problem especially during crossover operations when they greatly increase the likelihood of producing a damaged offspring. NEAT employs *innovation numbers* which are assigned to specific genes to enable correct alignment during crossover. As shown in Figure 3.1, *direct encoding* differentiating between *node genes* and *connection genes* is used. Another crucial concept of NEAT is *speciation* which groups individuals based on the similarity of their topologies. The speciation guarantees that the newly introduced individuals have time to evolve their structures before competing with more complex individuals from the rest of the population. These enhancements made NEAT performance-wise superior to other commonly used neuroevolution methods at the time.

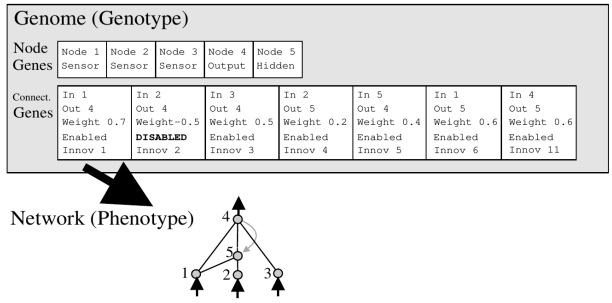

Figure 3.1: Mapping from genotype to phenotype in NEAT. Genotype is divided into two types of genes: *node genes* and *connection genes*. *Node genes* represent neurons and their types while *connection genes* describe connections between neurons including their weights and *innovation numbers* [147].

After NEAT was published [147], several papers suggesting improvements or extensions of the work appeared. HyperNEAT [146] suggests replacing direct encoding of NEAT with *indirect encoding* based on *Compositional Pattern Producing Networks* (CPPN). The new encoding expresses connectivity patterns in hyperspace which are then mapped to lower-dimensional space to produce the actual network connectivity. Because of that, HyperNEAT is able to efficiently encode much larger ANNs by utilizing regularities and repeating patterns. Apart from the efficient representation, the nature of CPPN enables HyperNEAT to further scale the produced networks with no additional evolution needed.

A different approach to evolving larger networks was taken by CoDeepNEAT [110]. In CoDeepNEAT, nodes of the genotype no longer represent individual nodes in

the produced network but whole layers. Therefore, each node needs to hold several so-called *hyperparameters* which determine layer properties (e.g. type of the layer, number of neurons or activation function). Also, the encoded edges do not possess weights anymore. They simply express how nodes (layers) are connected together. To convert the genotype into a DNN, the genotype graph needs to get traversed while replacing its nodes with layers of the corresponding configuration. In addition, genotypes also contain *global hyperparameters* which control the training of the produced DNNs (e.g. learning rate or the chosen optimizer). The assembled networks are trained via backpropagation for a fixed number of epochs. On top of that, CoDeepNEAT employs hierarchical decomposition into *modules* and *blueprints*. Modules are smaller parts of networks which are put together via blueprints. Both concepts are evolved in two separate populations where blueprints hold pointers to modules. The motivation is that repetitive modular structures can be evolved more efficiently. The approach taken by CoDeepNEAT is also commonly known as *neural architecture search*.

3.2 Neural Architecture Search

The goal of Neural Architecture Search (NAS) is to search for optimal neural network topology given the problem of interest [65]. Although NAS can be seen as a subfield of Neuroevolution (section 3.1), the motivation differs slightly. There are two main motivating factors [130]:

- The design of novel neural architectures is a manual, time-consuming and error-prone process.
- The process heavily depends on researcher's prior experience and knowledge which introduces biases.

In addition, NAS is often connected to *Deep Learning* (DL) which has achieved great success in many fields such as natural language processing, computer vision or machine translation [54], but has proven to be difficult to design good Neuroevolution-based solutions for [49].

As the research field concerning NAS is very broad, it was decided to split this section based on different stages of NAS. The first part (subsection 3.2.1) shows various approaches to *search space design* which determines what architectures can be represented. The second part (subsection 3.2.2) details the possible ways of *exploring the search space* in order to find the optimal solution. Finally, the last part (subsection 3.2.3) concerns *evaluation of performance* of the candidate architectures.

3.2.1 Search Space

A well-designed search space should reduce the effort needed to find a good architecture while staying flexible and reasonably constrained to enable efficient search. On the other hand, introducing too many constrains might bring a great amount of human bias which often prevents discoveries of novel architectures.

Chain-structured search space

One of the simplest solutions is *chain-structured search space* [44]. This kind of search space models the architecture as a sequence of n layers where each layer receives input from its preceding layer. An illustration is shown in Figure 3.2. The search space parameter set usually consists of:

1. The maximum number of layers
2. The type of the operations of each layer
3. The hyperparameters of each layer operation

This design was applied by Baker et al. [7]. They used reinforcement learning agent to iteratively choose CNN layers to generate high-performing architectures for image classification. Similarly, work by [149] utilizes Cartesian genetic programming for the same task.

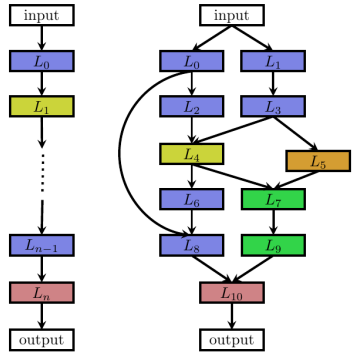

Figure 3.2: Left: *Chain-structured architecture* example. Each layer consumes the result of the previous layer. The only exception is the input layer. Right: *Multi-branch architecture* example which also includes branches and skip connections. Each colored rectangle corresponds to a network layer [43].

Multi-branch search space

Modern human-designed architectures often introduce new concepts which help to improve their performance. A particularly interesting enhancement was proposed by He et al. [61]. Their work suggests using skip connections to allow better propagation of gradients in deep neural networks. This technique was proven effective and also found its way to NAS.

Search space design inspired by skip connections is often called *multi-branch search space* [44]. An example is shown in Figure 3.2. This approach was employed by Elsken et al. [42] in their work about searching for well-performing CNN architectures via hill climbing, and their paper utilizing multi-objective architecture search which apart from predictive performance also considers network parameter count [43]. An influential work of Zoph and Le [194] also fits in this category. In contrast to the previous work, they utilized multi-branch search space together with a recurrent neural network-based controller which produces the candidate architectures.

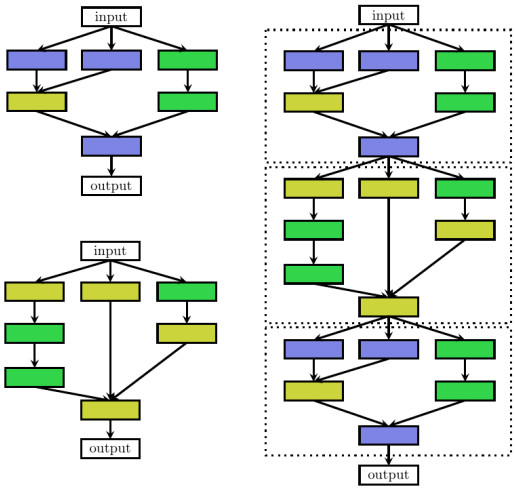

Figure 3.3: Left: Two different kinds of blocks/cells which encapsulate found or hand-crafted architectural patterns. Right: Stacking blocks/cells to produce a complete network architecture [43].

Cell-based search space

Another search space category proposed by some recent works is *cell* or *block-based search space*. In this case, the final architecture is composed of repeated patterns commonly referred to as *cells* or *blocks*. The core idea is to search only for the cell or block architecture and compose the final architecture by stacking them together. For illustration, see Figure 3.3.

The papers of Zoph et al. [195] and Zhong et al. [189] follow exactly this strategy. The former designed its own search space called *NASNet search space* (see Figure 3.4) which was used to evolve a novel *NASNet* architecture for image classification. The latter utilizes *block-wise generation* which essentially means stacking evolved blocks to produce the final architecture. In addition, the work of Liu et al. [88], Cai et al. [18], Pham et al. [119], Luo et al. [98], Liu et al. [89] and Zhang et al. [187] also benefit from a similar search space design.

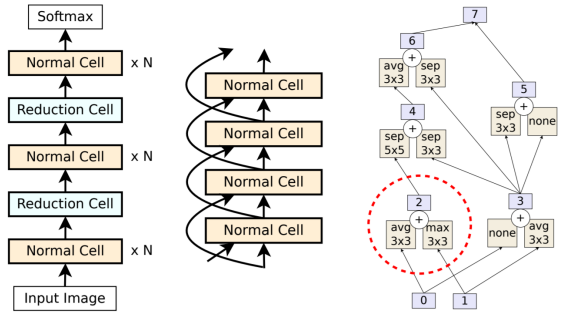

Figure 3.4: Left: The complete NASNet structure made of alternating *Normal cells* and *Reduction cells*. Middle: An illustration showing skip connections (omitted in the first diagram). Right: Detailed example of an evolved cell [195].

Unfortunately, *cell-based search space* introduces a new problem to solve. Namely, how to decide the *macro architecture* i.e. how many cells to stack, which kinds of cells and how to connect them to produce the actual architecture. For completeness, the inner structure of cells is sometimes referred to as *micro architecture*. To provide an example, each *NASNet cell* of Zoph et al. [195] consumes outputs of two previous cells and is later stacked into a sequential model. In contrast, Cai et al. [18] uses macro structures of known manually designed architectures and uses its custom cells within them.

Another issue is how to optimize both the macro architecture as well as the micro architecture together and ideally at the same time. Joint optimization can prevent creation of sub-optimal architectures in which a well-performing cell gets trapped in a badly designed macro architecture and vice versa [44].

Hierarchical search space

Liu et al. [87] proposes using a *hierarchical search space* which defines multiple levels for optimization. The first level defines atomic primitive operations, the second level defines connections between operations from the first level, the third level defines connections between patterns from the second level etc. In fact, *cell-based search space* can be seen as a special case of *hierarchical search space* with just two levels.

3.2.2 Search Strategy

After a suitable *search space representation* has been selected, the next step is to pick an appropriate *search strategy*. The main role of a *search strategy* is to sample promising candidate architectures from the search space. There have been many attempts using various methods, hence it was decided to review only the strategies that are potentially relevant to the goal of this work.

Reinforcement learning

The first strategy utilizes *Reinforcement learning* (RL), so it is referred to as *Reinforcement learning-based search strategy*. To formulate NAS as a RL problem, the agent is repeatedly asked to pick an action i.e. choose a network layer or cell until the whole network is built. The reward is determined by the estimated performance of the produced architecture. The presented works usually differ in agent's policy representation and in the way they optimize it.

Zoph and Le [194] opted for a recurrent neural network-based controller which implements the architecture generation policy. The controller iteratively generates a sequence of tokens which encode hyperparameters of the produced network. *REINFORCE* policy gradient method [165] was used to train the controller network. In contrast, their later work [195] used *Proximal Policy Optimization*. A similar controller design was chosen by Baker et al. [7] but they employed *Q-learning* algorithm for training. Similarly, Zhong et al. [189] also chose Q-learning.

Cai et al. [17] suggest to model the search as *sequential decision process*. In their paper, they propose to use a partially assembled architecture as the current

state which is then mutated by actions applying *network morphisms* [26]. Additionally, LSTM was used to allow compression of variable length architectures into a fixed-length encoding.

Gradient-based strategy

Another popular approach utilizes gradients, therefore, it was named *Gradient-based search strategy*. If the search space can be expressed as differentiable and continuous, gradients can be used to guide the search [65]. For the search space transformation, Liu et al. [88] employed *softmax relaxation* technique. The same approach was later used by Dong and Yang [39], Shin et al. [142] and Wu et al. [167]. Alternatively, Ha et al. [60] demonstrated the use of a *hypernetwork* i.e. a network which generates weights of another network in order to encode candidate networks into the continuous space. In the work of Luo et al. [99], *Neural architecture optimization* (NAO) method makes use of LSTM-based decoder and encoder to perform the search space transformation.

Evolutionary algorithms

A great number of strategies fall into *Evolutionary algorithms-based search strategy* category. In contrast to Neuroevolution (section 3.1), NAS-related approaches usually employ backpropagation for weight optimization and use evolution only for the optimization of the architecture itself. *Evolutionary algorithms* (EA) hold a population of candidate solutions (i.e. architectures) which undergo several iterations of evolution. During each evolution round, one or more solutions are sampled and subjected to crossover and/or mutation operations to produce offspring. In the context of NAS, crossover and mutations usually provide modifications to the architecture topology, for instance, adjusting hyperparameters or introducing new skip connections. The fitness of the newly created offspring is evaluated based on performance metrics. Model perplexity or validation accuracy are common examples of such metrics.

The typical point of divergence lies in how different works select parents for reproduction, create offspring and make space for them in the population. In both cases [128, 129], Real et al. adopted *tournament selection* [52] to sample parents, whereas Elsken et al. [43] used multi-objective genetic algorithm, therefore, could sample parents directly from the pareto front. Also, [128, 129] differs in the approach to individual replacement. The former replaces the oldest individual while the latter substitutes the worst-performing one. Zhu et al. [192] employed a special variant of crossover and mutation operators which are guided by information obtained incrementally during the evolution process.

To conclude, let's look at how different search strategies perform. Real et al. [129] focused on comparison of *Reinforcement learning-based strategies*, *Evolutionary algorithms-based strategies* and *random search* (which is considered a baseline). They reached the conclusion that both strategies produce equally good results while being consistently better than random search. In addition, evolution turned out to be better at finding more compact models. Liu et al. [87] focused only on *Evolutionary algorithms-based strategies* and *random search* achieving results similar to Real et al. [129].

3.2.3 Performance Estimation Strategy

Search spaces (subsection 3.2.1) are used to produce candidate architectures that are sampled by *Search strategies* (subsection 3.2.2). To inform and guide the search process, predictive performance of the produced models needs to be evaluated. The default approach is to fully train a given model for a fixed number of epochs to obtain its performance estimate. However, this technique may impose a high computational burden on the search process. Therefore, more efficient *Performance estimation strategies* were developed.

Reduced training fidelity

One of the popular strategies is to *reduce training fidelity* and use the approximate results as a proxy performance metric. To speed up training, Klein et al. [76] opted for using only a subset of the available training data. In contrast, Zoph et al. [195] and Zela et al. [183] trained their models for a small number of epochs and additionally retrained the final models. Yang et al. [177] proposed to train all models for a fixed number of epochs and then assign additional training time to well-performing models. However, Zela et al. [183] also observed that the greater the difference between in-search training setup and the final training setup is, the more skewed and less representative the proxy metric becomes. Hence, they advise to reduce training fidelity with care to prevent inclusion of biases.

Learning curve extrapolation

Another similar approach relies on *learning curve extrapolation*. The idea is that the initial learning curve accurately describes the trend a model would follow if trained further. That's why training of unpromising models can be stopped early on. The work of Klein et al. [76] takes advantage of this technique. A subtly different kind of extrapolation appears in the paper by Liu et al. [86]. They trained a surrogate model capable of predicting performance based on architectural properties of simple cell-based architectures. The surrogate model was then

used to predict performance of substantially larger architectures during an actual training process. Similarly, Baker et al. [8] used several regression models for performance prediction and compared the obtained accuracies. Domhan et al. [38] employed a weighted probabilistic model for the same task.

Network morphism

Network morphism technique proposes weight initialization of candidate architectures based on weights of similar already trained architectures. This idea was investigated by Wei et al. [164] and further utilized by Cai et al. [17] and Li et al. [82]. The goal is to transform one network into another while completely preserving its function i.e. make the morphed network inherit knowledge from its parents. Because of that, time-consuming training from scratch can be avoided. Usually, there is still some fine tuning required but the training duration is greatly reduced.

One-shot architecture search

One-shot architecture search constitutes another important group of methods. In this case, all candidate architectures are inherited from a super network called *one-shot model*. The one-shot model is trained only once and weight inheritance is used while sampling candidate architectures from it (these are simply sub networks of the one-shot model). No further training of sampled architectures is needed. Details of the process are explained in Figure 3.5. Brock et al. [14], Bender et al. [12] and Cai et al. [19] utilized this technique. On the other hand, weight sharing often introduces bias. One-shot models tend to underestimate performance of the best sampled architectures [12]. Another limitation is that the fixed and a priori defined super network noticeably constraints the search space. Also, during the sampling operation, the whole one-shot model needs to reside in memory which might cause issues with handling large networks.

Estimation without training

There have also been attempts to estimate neural network performance *without any training*. Mellor et al. [109] used overlaps of activations between data points in mini-batches to determine performance score of untrained networks. They claim that networks can be uniquely identified by a binary code which corresponds to the activation pattern of rectified linear units. Then, Hamming distance between binary codes can be used to obtain a matrix which takes distinctive forms for well-performing networks. Lopes et al. [95] and Wu et al. [169] employed a similar solution. Lopes et al. [96] aimed to eliminate unpromising

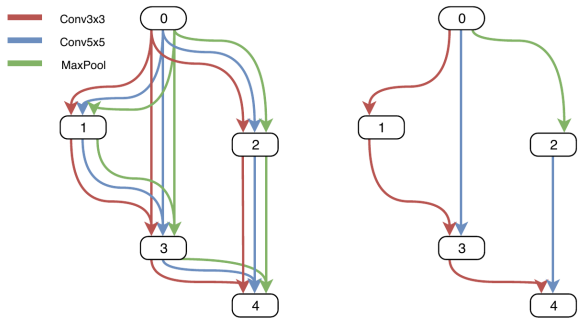

Figure 3.5: An illustration of *One-shot architecture search* process. Left: An example of One-shot model with one input node (0), one output node (4) and three hidden nodes (1,2,3). The edges represent operations that are applied to nodes. The model is trained as a whole including all nodes and operations. Right: To obtain a candidate architecture, the one-shot model is sampled to produce a sub network. Relevant weights of the one-shot model are transferred to the candidate architecture to avoid the need for training [44].

networks right after initialization. For that purpose, they used a zero proxy estimator based on Jacobian covariance of input data points. In the end, their method can also be seen as a variant of Mellor et al. [109] approach.

3.3 Transformer Architecture Search

Several works attempted applying Neural Architecture Search (section 3.2) to Transformers and the derived Transformer-based architectures. Apart from searching for complete architectures, a common approach is to limit the focus to a subset of Transformer components. A popular candidate is the attention mechanism i.e. searching for more efficient or better performing variants of it. In addition, there is an active area of research focused on evolving efficient transformers for resource-constrained environments.

3.3.1 Search Space

Similarly to subsection 3.2.1, this section is also divided into categories based on different search space types. For the definition of the categories, consult the mentioned subsection. The topic is the same as in subsection 3.2.1, however, this time, the section focuses on Transformer-based architectures.

Multi-branch search space

Chen et al. [25] used a variant of multi-branch search space while searching for Vision Transformer architectures specialized on image processing. Interestingly, the search space used in their work was made dynamic and evolved alongside the architectures. In another paper of Chen et al. [24], a similar search space type was also used. They split the Transformer architecture into stackable building blocks with variable hyperparameters. The parameters are then adjusted during the search process.

Cell-based search space

An influential paper by So et al. [143] describes the use of NAS to search for a better alternative to the vanilla Transformer [159]. They proposed a cell-based search space based on NASNet [195] including modifications allowing representation of Transformer architectures. The search space cell structure is shown in Figure 3.6. Kim et al. [73] employed the same search space type while evolving architectures for automatic speech recognition. Furthermore, Jing et al. [69] used it as a foundation for Transformer-based architecture search framework.

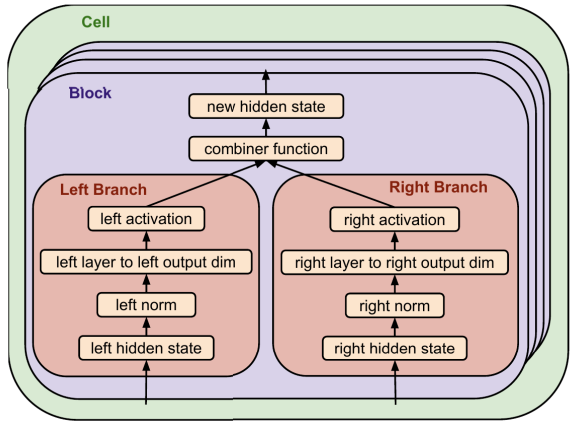

Figure 3.6: Search space cell of *The Evolved Transformer*. Each of the blocks (violet) ordered and stacked within the cell (green) outputs a hidden state which is added to a hidden state pool. Other blocks can then select two hidden states produced by the preceeding blocks as input. These inputs are then fed into the left and right branches (red) of the corresponding block [143].

3.3.2 Search Strategy

The majority of search strategies used in connection with Transformers seem to be based on evolutionary algorithms. However, a few papers also employed reinforcement learning or one-shot search.

Reinforcement learning

Liu et al. [90] investigated the use of Transformers together with convolution and simple multi-layer perceptrons. The intention was to search for their optimal combination to obtain an architecture suitable for vision tasks. To perform the search, they employed reinforcement learning and used a simple RNN-based controller. Zhu et al. [193] also used RL to navigate through the search space. Their goal was to automatically design Transformer architectures optimized for natural language processing.

Evolutionary algorithms

Tsai et al. [157] focused on finding fast and efficient Transformers with a help of evolutionary algorithms. They suggested decomposing Transformer into series of blocks with hyperparameters which can be adjusted by the evolution process. So et al. [144] also tackled the problem of searching for efficient Transformer alternatives, however, in contrast to the previously mentioned work, they performed evolutionary search on a lower level. Their search space consisted of primitives that define a TensorFlow program. The architecture discovered during the performed search is called *Primer*. In comparison to vanilla Transformer, the main differences are: squaring of ReLU activations and adding depthwise convolution after each query, key and value projection in the attention sub-layers.

Guan et al. [58] proposed a framework for automatic design of self-attention models called *AutoAttend*. In their work, NAS was used to search for optimal attention models as well as to optimize the macro architecture of Transformer. The attention model search was designed in the same way as the macro architecture search, therefore, the search space and search strategy were reused directly. Additionally, The Evolved Transformer [143], Global and Local Image Transformer [23] and Transformer for neural machine translation by Feng et al. [46] also utilized evolutionary search.

Gradient-based search

Zhao et al. [188] explored the possible ways of applying differentiable architecture search to Transformers. However, their efforts were quickly hindered by too high memory requirements. To deal with this problem, they decided to split the

network into n reversible networks which allow their inputs to be reconstructed from their outputs. Thanks to this technique, the memory needed for storing the network decreased significantly and differentiable architecture search became possible. Mandava et al. [107] investigated trade-offs between different orderings of multi-head attention and feed-forward modules in Transformers. To search for an optimal solution, they also used differentiable architecture search method inspired by the work of Wu et al. [167].

3.3.3 Performance Estimation Strategy

For Transformer-based architectures, sophisticated performance estimation strategies seem to be utilized to a much lesser degree. To obtain an evaluation score, most of the works simply default to training of candidate models from scratch and using their validation accuracy as a performance metric. Nevertheless, there is a few noticeable exceptions.

Reduced training fidelity

Because the search for complete Transformer architectures was proved to be time-consuming, a speedup method called *Progressive Dynamic Hurdles* was proposed by So et al. [143]. During training, the method dynamically allocates more training steps to models with promising evaluation metric values (such as perplexity or accuracy). The same technique was also utilized by Kim et al. [73].

One-shot search

Several papers used one-shot search to avoid training of individual architectures by sampling them from an already trained super network. The set includes papers from Tsai et al. [157], Chen et al. [24] or the work of Guan et al. [58].

3.4 Transformer Modifications

As Transformer architectures started to get more widely used, their weaknesses were revealed. In some cases, the weaknesses even hindered applications to new problems. The main issue is that attention calculation has $O(n^2)$ complexity in both memory and time, so it becomes difficult to process long sequences. Other difficulties include keeping track of very long dependencies between sequence elements, making the model smaller or designing efficient positional encoding. For that reason, numerous improvements and modifications were proposed. The most influential ones are described in the following subsections.

3.4.1 Architecture Variations

After the first Transformer was introduced (subsection 2.5.3), new lines of research focused on improving the architecture appeared. Surprisingly, the most successful works either proposed relatively minor architectural changes or utilized a scaled up variant of the Transformer *encoder* or *decoder* stage only. It is worth mentioning that apart from the architectural modifications, the performance improvements of many of the presented architectures are also due to *masked pretraining* [35] which is explained in a separate subsection 3.4.6.

Encoder-decoder

The original *encoder-decoder* Transformer by Vaswani et al. [159] is an auto-regressive model. In auto-regressive models, the element being produced has access to all previously produced elements via attention i.e. *the context to the left* from the current element. An encoder-decoder configuration similar to the original Transformer was used by Raffel et al. [125] who explored the possibilities of transfer learning. They managed to reach state-of-the-art results on several NLP benchmarks even though their model was not trained from scratch on all measured tasks. The work of Song et al. [145] had a similar focus and also utilized the full encoder-decoder variant. Fedus et al. [45] focused on scaling up the Transformer architecture and improving its performance on larger datasets by routing data within the model. Borgeaud et al. [13] optimized the encoder-decoder variant for retrieval of knowledge from textual documents. Their approach performed especially well on knowledge intensive tasks such as question answering.

Decoder-only

Decoder-only Transformer variants are also auto-regressive i.e. have the *left context* available. Some of the most prominent representatives are the three versions of *Generative Pre-trained Transformer* i.e. GPT [122], GPT2 [123] and GPT3 [16] which perform text generation. At the time, they managed to advance state-of-the-art in several tasks such as machine translation, reading comprehension or document summarization. Building on the learnings from GPT3, *PaLM* architecture [29] scales up the decoder even further reaching hundreds of millions of learnable parameters and improved performance.

Encoder-only

The main feature of *encoder-only* variants is that the models are no longer auto-regressive i.e. produce the whole output sequence at once. The input sequence is also processed in one go which allows all elements to access both *left* and *right contexts* enhancing model learning capabilities. The first proposed encoder-only

model is called BERT [35] which stands for Bidirectional Encoder Representations from Transformers. The model was trained and evaluated on popular NLP benchmarks reaching state-of-the-art or beyond performance on them. The solution was further improved by Liu et al. [93] who found that BERT was significantly undertrained and can perform even better. However, the model became quite large which made its proper training even harder. Hence, Lan et al. [78] came up with measures for lowering model memory consumption and increasing its training speed.

3.4.2 Layer Normalization

Layer normalization (LN) and residual connections are mechanisms which help neural networks to stabilize training and avoid degeneration of gradients. In the original Transformer, both mechanism are used extensively (section 2.5). Additionally, there exists research investigating the optimal use of such mechanisms in Transformers which already proposed alternative solutions. The alternatives usually challenge the placement of layer normalization within the network or suggest different ways of performing normalization.

Layer Normalization Placement

There are two common variants of Transformer layer each using a different placement of layer normalization: *pre-norm* and *post-norm*. The former places LN right before attention or feed forward sub-layer, so the residual connection goes around it while the latter places LN in between the blocks formed by residual connections. For clarity, both variants are illustrated in Figure 3.7. The post-norm option was used in the original Transformer while pre-norm was adopted by some later works [27, 162]. Xiong et al. [171] discovered that gradients near the last layer in post-norm Transformers become quite large which might explain the difficulties with Transfomer training and the need for learning rate warmup. The pre-norm Transformers do not suffer from the same problem. Unfortunately, Wang et al. [162] showed that post-norm Transformers are more performant.

Alternative Normalizations

Raffel et al. [125] found beneficial to use a simplified version of layer normalization which rescales gradients but applies no additive bias. Xu et al. [173] noticed that the learnable parameters of LN do not bring any measurable improvement and can even increase the risk of overfitting. For that reason, they proposed *AdaNorm*: a normalization technique with no learnable parameters. Nguyen and Salazar [114] suggested using *scaled l2 normalization* instead of LN as it is more

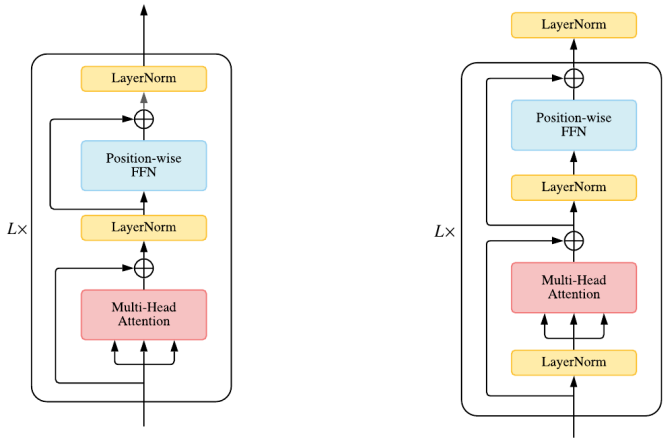

Figure 3.7: Left: *post-norm* variant of the Transformer layer, Right: *pre-norm* variant of the Transformer layer [85].

parameter efficient and was shown to perform equally well. Shen et al. [140] investigated why batch normalization performs poorly in Transformers. Based on their learnings, they proposed *PowerNorm*: a batch normalization technique optimized for Transformers.

3.4.3 Attention Modelling

To mitigate the $O(n^2)$ attention complexity, alternative attention modelling approaches were proposed. A common technique is to restrict the number of elements to attend to by imposing a specific access pattern. Other models try to approximate the calculated attention scores. Additionally, attempts to replace the attention model with other constructs were also proposed.

Fixed Attention Patterns

Some of the early modifications suggest limiting the elements available for attention calculation by imposing a *fixed mask pattern* on them. Qiu et al. [120] split the input sequence into fixed blocks which reduced the attention complexity to $O(b^2)$ where $b << n$ and b stands for the block size. Another approach is to attend only at predefined intervals which was employed by Beltagy et al. [11]. *Longformer* (as they call the architecture) combines sliding attention window with global attention which results in *sparse attention* pattern. This allows

the lower levels of the model to capture local patterns while higher levels can focus on modelling of the global ones. The resulting attention pattern is shown in Figure 3.8.

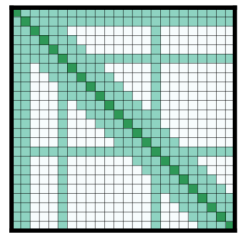

Figure 3.8: The fixed attention pattern of Longformer. The picture illustrates how elements attend to other elements. Both vertical and horizontal axis represent elements of the same sequence, therefore, the diagonal line shows attention to the element itself [11].

Similarly, Zaheer et al. [182] combined three different types of fixed attention patterns in their *Big Bird Transformer*: *global attention* which defines a subset of elements that can attend to the whole sequence, *window attention* which allows attention to nearby tokens and *random attention* that attends to randomly chosen elements. An illustration of the final pattern is given in Figure 3.9. In addition, Child et al. [27] employed a similar technique in their *Sparse transformer*.

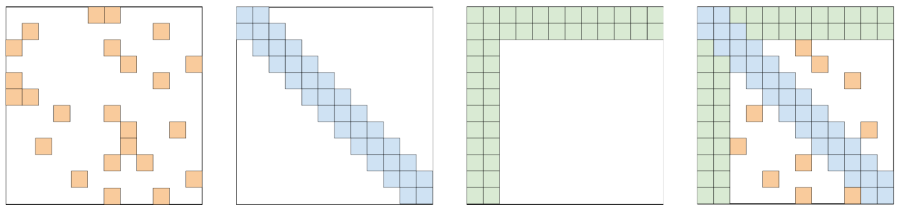

Figure 3.9: The attention pattern of Big Bird Transformer. Three fixed attention patterns were combined into one [182].

Learnable Attention Patterns

In addition to fixed patterns, *learnable patterns* were also used. *Reformer* by Kitaev et al. [74] relies on *locality-sensitive hashing* to reduce the attention complexity from $O(n^2)$ to $O(n\ log(n))$. The hashing works as a similarity measure which clusters tokens into chunks efficiently. *Routing Transformer* [133] utilizes *k-means clustering* algorithm to obtain a similar grouping of tokens. A similar clustering approach was also used by Vyas et al. [160]. Zhu et al. [191] suggested using two separate attention models whose outputs get combined. The first model employs a learnable projection to capture long-term dependencies while the second short-term model focuses on local relations between elements. Sukhbaatar et al. [150] proposed an attention mechanism which is capable of learning the optimal attention span. Furthermore, Tay et al. [154] achieved attention sparsity by learning how to sort blocks of elements extracted from the input sequence.

Approximation of Attention

A typical representative of *attention approximation* is *Linformer* [163] which approximates attention values by a low-rank matrix. The original n^2 attention matrix is decomposed into $n \times k$ where $k << n$. Based on these learnings, the authors propose a new attention mechanism which reduces $O(n^2)$ complexity down to $O(n)$ in both time and space. Choromanski et al. [28] also employed a similar kind of attention approximation. *Perceiver* proposed by Jaegle et al. [68] uses cross-attention module to project input into a fixed-dimensional latent bottleneck array. This trick enables efficient handling of large inputs as it effectively decouples the model from the input size. Attention is then calculated iteratively by stacking several cross-attention modules interleaving with latent self-attention blocks. In fact, Perceiver can be seen as a variant of recurrent neural network. Figure 3.10 shows the architecture in more detail. Finally, Katharopoulos et al. [70] and Peng et al. [118] suggested using kernel-based dot product attention instead of softmax attention which allowed reduction of complexity to $O(n)$.

Alternatives to Attention

Among the vast amount of research which aims to improve the attention mechanism, there exist a few papers that question the actual importance of it. Zhai et al. [185] introduced *Attention Free Transformer* which eliminates the need for scaled dot product attention. Instead, they proposed combining keys and values with a set of learned position biases and multiplying them by queries. In their experiments, they managed to obtain performance competitive to vanilla Transformer. Lee-Thorp et al. [80] replaced self-attention with Fourier Transforms

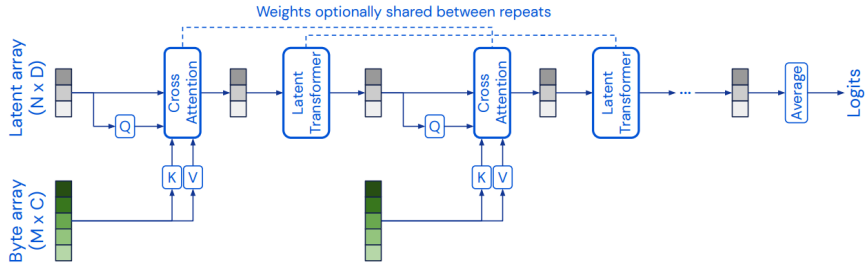

Figure 3.10: A stack of alternating cross-attention modules and transformer-style self-attention blocks. The byte array represents the original input while latent array is its low-dimensional projection (at least in the beginning) [68].

while retaining up to 97% of the original Transformer classification accuracy. *Synthesizer* by Tay et al. [153] successfully substitutes attention with random alignment matrices. As well as the previous papers, they reach competitive performance. Bello [10] replaces attention with a set of linear functions termed *lambdas* and also obtains promising results. Gulati et al. [59] proposed using convolution in addition to the standard multi-head attention to improve capturing of the local element context. The convolution module was placed in between attention and feed forward sub-layers. Finally, Wu et al. [168] experimented with using lightweight convolutions instead of attention.

Additional Memory

Leveraging a global memory module that can access several elements at once is another verified approach. *Set Transformer* introduced by Lee et al. [79] uses *inducing points method* which forms a temporary memory and holds context for future processing. Dai et al. [33] proposed *Transformer-XL* which in order to preserve history maintains a memory of past activations at each layer. When these activations become too old, they are discarded. *Compressive Transformer* by Rae et al. [124] builds on similar principles, however, instead of discarding the memory, it compresses it. *Extended Transformer Construction* [2] and *Longformer* [11] also used some form of a global memory.

3.4.4 Feed Forward Sub-layer

Aside from attention and layer normalization, feed forward sub-layer constitutes another important building block of Transformer layers. Modifications to this

sub-layer usually attempt to expand the capacity of its fully-connected layers (to increase the amount of information they are able to absorb) or to pick a more suitable activation function.

Fully-connected layers

Shazeer et al. [138] demonstrated the use of *Mixture-of-Experts* (MoE) technique in neural networks. MoE applies conditional computation, where parts of a network are activated based on the supplied input sequence i.e. different sequences might get routed through a different part of the network. Yang et al. [176] and Fedus et al. [45] proposed MoE-based feed forward sub-layer optimized for the use in Transformers. Yang et al. [178] discovered that feed forward sub-layers are not being used efficiently in Transformer decoders. They also showed that the sub-layer can be removed with little to no loss of performance.

Activation function

The default activation function used in the vanilla Transformer is ReLU. Ramachandran et al. [126] tried to replace ReLU with SiLU [127] and experienced consistent improvements in performance. GPT [122] replaced ReLU with GELU [63] which then became a popular choice even for more recent language models [34, 62]. So et al. [144] conducted Transformer architecture search and discovered that squaring ReLU activations leads to a better performance. Other common choices include LeakyReLU [172] and ELU [30] activation functions.

3.4.5 Positional Encoding and Embedding

Positional encoding is a way of encoding positional relations of elements in a sequence while *positional embedding* refers to a n-dimensional vector representing positional encoding. Positional encoding techniques have been studied even before the introduction of Transformer architecture, so there is a wide variety of solutions to draw inspiration from. Moreover, alternatives and improvements tailored specifically to Transformers have also been proposed.

The positional encoding methods presented below can be divided into two groups based on how they supply the encoding/embedding to Transformer models:

1. **Added positional encoding (APE)**: Add positional embedding to the input of the model.

2. **Modified attention matrix (MAM)**: Inject positional encoding directly to *attention matrices* inside Transformer layers.

Attention matrix refers to n^2 attention scores matrix used in vanilla multi-head attention (and some other attention models) which holds scores representing attention between all sequence elements. The scores they are then passed through softmax to obtain the final attention weights (see subsection 2.5.3).

Absolute positional encoding

Absolute positional encoding encodes the absolute position of an element within a sequence. The original Transformer paper [159] utilizes *fixed sinusoidal positional encoding* which is an example of absolute encoding (subsection 2.5.3). The encoding is added to the model input embedding (APE). The effectivity of sinusoidal encodings was also verified by Liu et al. [92]. Yan et al. [175] noticed that dot product of two sinusoidal positional embeddings depends only on their relative distance and also that the embedding is unaware of direction. To alleviate these issues, they proposed *direction and distance-aware positional encoding*. Li et al. [83] claimed that variance of sinusoidal positional embeddings depends on the actual position i.e. tends to be small for lower positions and tends to be large for later positions. To mitigate the problem, *maximum variances positional embedding* was introduced.

An alternative to fixed sinusoidal encoding is *learnable absolute positional encoding*. Devlin et al. [34] and Liu et al. [93] used a generic learnable embedding which learned element positions from scratch without any kind of positional encoding provided beforehand. Kitaev et al. [74] proposed a parameter-efficient learned embedding called *axial position embedding*. In fact, their embedding encapsulates two different kinds of embeddings. The first embedding is used to encode larger segments while the second embedding determines positions within each segment. Liu et al. [92] modelled positional information with a *continuous dynamic model* and injected it into attention matrices in each encoder layer (MAM). Based on their experiments, this approach outperforms sinusoidal absolute encodings.

Relative positional encoding

Relative positional encoding is a positional encoding method which encodes element positions relative to other elements in a sequence ignoring their absolute position. He et al. [62] added relative positional biases to attention matrices in all Transformer layers (MAM). Raffel et al. [125] used a combination of learnable positional encoding and relative positional biases. The resulting encoding was also injected directly to attention matrices (MAM). However, the learned component was shared. Chang et al. [21] investigated a combined use of convolutions

with relative positional encoding. Their efforts were successful as they detected improved performance in natural language understanding tasks.

Combined positional encoding

Combined positional encoding represents techniques which combine certain forms of absolute positional encoding (section 3.4.5) and relative positional encoding (section 3.4.5). Su et al. [148] introduced MAM positional encoding which uses a rotation matrix to encode element positions. The matrix encodes absolute positions directly but also holds implicit relative positional biases. Ke et al. [72] discovered that adding positional encoding to the input embedding introduces mixed correlations between the two information sources. For that reason, they proposed *untied position embeddings* injected directly into attention matrices (MAM). The untied approach opted for modelling both absolute and relative encoding separately and adding them together just before they are provided to the model. Shaw et al. [137] concluded that relative encoding improves performance in machine translation tasks whereas the combination of relative and absolute encoding does not bring any benefits.

3.4.6 Masked Pretraining

Transformer pretraining is an approach suggesting training of Transformer models in two consecutive steps. In the first step, the models are pretrained on large datasets (e.g. large corpora in NLP) which makes them learn universal representations beneficial for downstream tasks [121]. After pretraining, the models are fine-tuned for a given task on a smaller dataset of interest. The popular *masked pretraining* technique was pioneered by Devlin et al. [34] who proposed pretraining of Transformer encoder-only models by randomly masking out certain portions of sentences and asking the model to predict the masked sections. After that, the model was fine-tuned for a number of benchmark tasks in which it surpassed performance of non-pretrained models. The same method was further refined by He et al. [62] and Liu et al. [93]. A similar technique was applied to GPT decoder-only models [122, 123, 16] improving their performance significantly. Additionally, Zerveas et al. [184] successfully performed pretraining on time series datasets.

3.5 Transformers for Time Series Forecasting

Despite the fact that Transformers were originally proposed for solving natural language processing problems, their span has recently expanded to new domains.

They are now being used to tackle problems in different fields such as image processing [40] or biological sequence analysis [131]. As the goal of this work is to apply Transformers to time series forecasting, this section presents some of the early attempts and highlights concepts potentially applicable to this work.

Wu et al. [170] used Transformer to forecast influenza-like illness epidemics. The employed architecture was taken from the work of Vaswani et al. [159] but included a few minor modifications. The forecasted time series were encoded via *time delay embedding* (TDE) which embeds their values into d-dimensional vectors that Transformers can process. The embedding definition is shown in Equation 3.1.

$$TDE_{d,\tau}(x_t) = (x_t, x_{t-\tau}, ..., x_{t-(d-1)\tau}) \tag{3.1}$$

where x_t is time series data at the timestamp t, d is the dimension of the embedding vector and τ is the time lag. To obtain training examples for the model, *fixed-length windowing* was used to extract pairs of consecutive time series patches created by sliding a window over the full time series. The technique is illustrated in Figure 3.11.

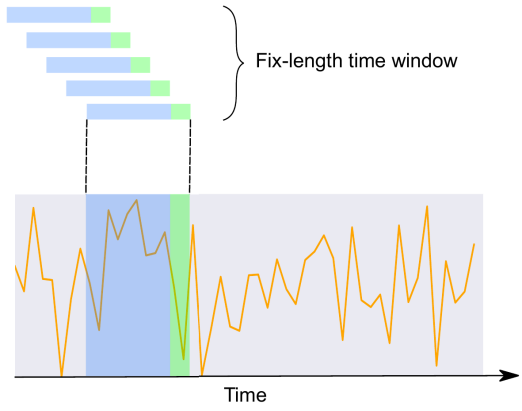

Figure 3.11: Use of a sliding window for training examples generation [170].

Zhou et al. [190] introduced *Informer*, a custom designed Transformer-based architecture optimized for long time series forecasting. The first proposed enhancement is the use of a fixed attention pattern-based *ProbSparse* attention model

which lowers the attention complexity to $O(n\ log(n))$ in both time and memory usage. Secondly, Informer relies on *attention distilling* method which uses convolution to incrementally filter out elements that obtain low attention scores. This approach helps with handling of long sequences as the number of considered elements decreases with every Transformer layer. Finally, Informer is not used in autoregressive manner. The whole forecast is produced at once which avoids accumulation of forecast errors.

Grigsby et al. [56] explored the use of Transformers for multivariate time series (MTS) forecasting. They proposed converting MTS into a flat *spatiotemporal sequence* in which each token represents a value of a single time series at a given timestamp. This makes the model capable of learning interactions between space, time, and the actual series values properly. The authors chose positional embedding based on *Time2Vec* [71] which uses value timestamps to capture long-term seasonal patterns or periodicity. Their model architecture was heavily inspired by Performer [28] and Informer [190] Transformers.

Zerveas et al. [184] also focused on multivariate time series but in contrast to Grigsby et al. [56] they used only the encoder part of Transformer. The work investigated the potential benefits of masked pretraining with time series datasets. Evaluation on multiple benchmarks showed that the pretrained encoder performs significantly better than the other currently available methods used for time series regression and classification. The difference was most noticeable on small datasets containing only several hundreds of training examples.

Cai et al. [20] examined the use of Transformers for traffic forecasting. For that purpose, they customized the vanilla Transformer [159] and proposed *Traffic Transformer*. One of their main contributions was a detailed analysis of positional encodings for time series forecasting. In the end, they merged several kinds of encodings together (each capturing different time series features such as continuity or periodicity). In addition, they enhanced the model by including *graph convolutional neural networks-based layer* for capturing spatial dependencies.

Another attempt was made by Yi et al. [180] who used Transformers for tourism demand forecasting. Their model is nearly identical to Vaswani et al. [159] and used input encoding similar to Wu et al. [170]. They also performed in-depth comparison of the model with other time series forecasting methods.

Shen and Wang [139] suggested combining attention with convolution to achieve better efficiency and enhance the locality of the model. For that reason, they designed *CSPAttention block* which splits signal into two paths. One path applies convolution and the second one performs self-attention. The architecture

is in fact based on Informer [190] with one noticeable difference. In the distilling operation of Informer, regular convolutions are replaced with *dilated causal convolutions*. This enables exponential widening of the receptive field. The final proposed enhancement is *passthrough mechanism* which is in fact a simple residual connection around multiple encoder layers.

Some general improvements to Transformer architecture for time series forecasting were proposed by Li et al. [84]. They developed self-attention mechanism based entirely on causal convolutions which excels at capturing local context. In addition, to mitigate the Transformer memory bottleneck, they introduced *LogSparse Transformer* which was showed to capture long-term dependencies well while keeping the memory usage at $O(n\,log(n)^2)$.

3.6 Time Series Forecasting Methods

In addition to classical methods (subsection 2.4.1) and simple neural network-based models (subsection 2.4.2), more sophisticated time series forecasting methods are being introduced. Flunkert et al. [48] proposed *DeepAR* methodology which uses RNN and LSTM networks trained on a large number of related time series to improve forecasting accuracy on the time series of interest. Bai et al. [6] investigated application of convolutional neural networks (subsection 2.2.1) to sequence modelling and discovered that a relatively simple CNN outperforms RNNs on several benchmark tasks such as language modelling. The prosposed architecture is referred to as *Temporal Convolutional Network* or simply TCN. An important part of the architecture are *causal convolutions* which ensure that at time t convolution can only access sequence elements from time $t-1$ or earlier.

Oreshkin et al. [116] introduced *N-BEATS* neural architecture specialized for forecasting built as a deep stack of fully-connected layers. Different groups of layers focus on different time series components (subsection 2.3.2) which makes explanation and interpretation of the produced forecasts easier. Taylor and Letham [155] came up with a modular regression model with interpretable parameters which can be tuned by experts who have the appropriate domain knowledge of the forecasted time series. The model is referred to as *Prophet*. The same model was further improved by Triebe et al. [156] who incorporated neural networks into it producing a hybrid *NeuralProphet* model connecting classical methods with deep learning while also being superior in accuracy.

3.7 Forecasting Benchmarks

Most of the available *time series forecasting benchmarks* were initially released as forecasting competitions aiming to compare accuracy of various forecasting methods. A well-known series of competitions are Makridakis competitions also known as M*x*. The first *M1* competition [135] contained 1001 real-world time series collected primarily from firms and industries. The second *M2* competition [104] considered 26 time series only but put more emphasis on the evaluation of forecasting methods. The competition lasted for almost 4 years during which the organizers collected additional data that were then used for the methods evaluation. Later, a similar *M3* [103] and recently also *M4* [105] and *M5* [102] competitions took place. Crone et al. [31] took 111 time series from the M3 competition and set up a new *NN3* competition targeting neural network-based forecasting methods. To raise awareness about the importance of energy forecasting, *Global Energy Forecasting* competition was held in 2012 [64]. Also, Athanasopoulos et al. [4] proposed another competition focused on forecasting of tourism data.

Bauer et al. [9] argued that most of the forecasting competitions use very homogeneous datasets which hinders objective evaluation of the compared forecasting methods. To illustrate the issue, they showed that e.g. majority of M3 time series are less than 100 samples long or that M4 series have mostly short frequencies of seasonality. For that reason, they proposed *Libra*, a forecasting benchmark which evaluates forecasting methods on a diverse set of time series coming from various domains and sources. The benchmark consists of four categories of time series (*economics*, *finance*, *human access* and *nature and demographics*) each containing 100 different series. Bauer et al. also conducted statistical analysis of the benchmark to support their claims about the diversity and heterogeneity of the time series included in it.

Chapter 4

Method and System Design

This chapter presents a framework used to design the evolution-driven system for neural architecture search and subsequently obtain competitive Transformer-based architectures for time series forecasting. The chapter is split into several sections describing individual phases of the framework as explained in section 4.1. Apart from these, there is one extra section discussing the explicit design decisions made (section 4.2). The work presented in this chapter maps to Suggestion and Development phases of Design Science Research methodology (section 1.4).

4.1 Design Framework

The proposed framework defines several phases each representing a step towards the design of evolution-driven NAS system and creation of Transformer-based architectures for time series forecasting. All phases are illustrated in Figure 4.1. The first phase focuses on selecting a well-performing Transformer variant to act as a base for genotype encoding design (section 4.3). The second phase compares attention models and selects a subset suitable for use with time series (section 4.4). The third phase deals with genotype encoding design (section 4.5). The fourth phase covers design of the neural architecture search system later used to evolve Transformer-based architectures (section 4.6). The fifth phase gathers learnings from all previous phases in order to evolve a strong architecture for forecasting (section 4.7). Notice that the diagram shows a recurrent connection on this phase as the evolution might require several attempts before it produces a well-performing architecture. Finally, the last phase compares the evolved architectures with other common forecasting methods (section 4.8). If the architectures reach the desired accuracy, the framework terminates, otherwise, it starts over as some decisions made along the way may need to be reconsidered.

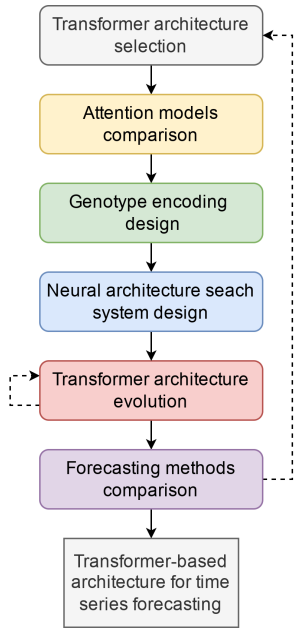

Figure 4.1: The framework used to design evolution-driven NAS system and obtain a Transformer-based architecture for time series forecasting.

4.2 Design Decisions

Design and implementation of the described NAS system were guided by several design decisions made beforehand. These decisions are either based on findings made during literature review (chapter 3) or simply represent attempts to lower the complexity of the system.

Univariate Time Series without Timestamps

The system supports only univariate time series without timestamps. If a series has timestamps assigned, they are ignored, however, the data can still be used. This keeps the system more flexible as time series with timestamps are not available in all datasets.

Time Series Embedding

Transformers were designed to process sequences of elements encoded as vectors. Unfortunately, elements in univariate time series are represented as scalars i.e. their vector representation needs to be built explicitly. To address this issue, an approach used by Zhou et al. [190] and Grigsby et al. [56] was utilized. Univariate time series are split into equally long patches which creates vectors that can be consumed by the model. In addition, each vector is passed through a fully-connected layer to adapt the patch sizes to the embedding dimension of a Transformer model.

Search Strategy

The conducted literature review (chapter 3) revealed that several search strategies (subsection 3.2.2) are applicable in the context of NAS. Some of the strategies, primarily Reinforcement learning (section 3.3.2) and Evolutionary algorithms (section 3.3.2), have also been successfully applied to Transformer architectures. In the proposed system, it was decided to employ strategy based on Evolutionary algorithms (section 3.2.2). There are several works supporting this decision. Firstly, Maziarz et al. [108] concluded that EA are capable of beating RL-based strategies consistently. Additionally, multiple papers demonstrated evolution of Transformer architectures for a wide-variety of tasks such as neural machine translation [46], speech recognition [73] or NLP [143].

4.3 Transformer Architecture Selection

The first phase of the framework focuses on selecting a Transformer architecture suitable for time series forecasting. The subsection 3.4.1 has presented three possible alternatives: *encoder-decoder*, *encoder-only* and *decoder-only*. Additionally, there are two other variations which seem to be worth comparing. Zhou et al. [190] proposed a modification of the encoder-decoder variant to make it more efficient when used for regression. This variant was dubbed *Informer encoder-decoder*. Also, Zerveas et al. [184] suggested to concatenate all hidden states of encoder-only models before passing them through the final fully-connected regression layer. This variant is referred to as *merged encoder-only*. More details about both variants can be found in section 3.5. To sum up, this phase aims to compare these five alternatives and determine which variant should be used as a base architecture for genotype encoding design. The comparison is done in two consecutive steps: 1.) hyperparameter search to determine optimal setup for all variants and 2.) training and evaluation of each variant on benchmark datasets.

Parameter	Type	Description
patch size	*int*	Size of patches that a time series is divided into before entering the model
embedding dim	*int*	Embedding dimension of the Transformer model
head count	*int*	Number of heads in multi-head attention sub-layer
layer count	*int*	Number of stacked encoder and/or decoder layers inside the model
forward expansion	*int*	Expansion ratio of fully-connected layers inside the feed forward sub-layer
dropout probability	*float*	The probability of dropout in the Transformer model

Table 4.1: A list of Transformer hyperparameters.

In the first step, evolutionary algorithm-based approach is used as there are several works which confirmed its suitability for Transformer hyperparameter search [24, 46]. Hyperparameters are encoded as a simple list of values which makes it possible to use basic and well-established crossover and mutation operators (see section 2.1 and subsection 2.1.4). The list of considered hyperparameters is provided in Table 4.1. To obtain fitness of a candidate, the candidate is first trained on a training dataset and then evaluated on a test dataset. The accuracy obtained on the test dataset serves as fitness. Unfortunately, this kind of evaluation is quite time-consuming, so tournament selection was used to avoid evaluation of many candidates (subsection 2.1.3).

In the second step, the discovered hyperparameters are used to parametrize all five variants which are then fully-trained and evaluated for forecasting accuracy on several benchmark datasets. The best performing architecture is selected as a base for later phases.

4.4 Attention Models Comparison

The goal of the second phase is to determine a subset of attention models which perform well when used in time series forecasting Transformers. The candidate models were selected from the set of models described in subsection 3.4.3.

The chosen models are listed in Table 4.2 and Table 4.3. The approach is simple, a Transformer architecture with fixed hyperparameters is used as a host for all models to compare. Then, several Transformer instances differing only in the used attention model are trained on a train dataset and evaluated on a test dataset. The accuracy obtained on the test dataset serves as a measure of attention model performance. The resulting subset of models is utilized in the following phases.

Attention model	**Description**
Passthrough attention	Dummy model which performs no attention modelling. It is included to form a baseline.
Multi-head attention	Original attention model proposed in the first Transformer [159] (subsection 2.5.3).
Longformer attention	A model using a fixed attention pattern (section 3.4.3) which combines global attention with local sliding window attention [11].
Big Bird attention	Also employs a fixed pattern (section 3.4.3). In comparison to Longformer, it works with blocks of elements and adds attention at random locations in the pattern [182].
Long-short attention	A model aggregating two independent attentions to better capture both long and short-term dependencies [191] (section 3.4.3).
Reformer attention	An efficient attention model based on locality-sensitive hashing [74] (section 3.4.3).
Routing attention	A model capable of learning sparse patterns dynamically (section 3.4.3) by routing attention via k-means clustering [133].
Adaptive span attention	Multi-head attention extension capable of learning an optimal attention span [150] (section 3.4.3).

Table 4.2: A list of Transformer attention models (part 1).

4.5 Genotype Encoding Design

The third phase concentrates on genotype (and phenotype) encoding design. The main requirement of the encoding is that it needs to be capable of representing all concepts found in the Transformer architectures of interest. In addition to the

Attention model	Description
Linformer attention	A model which approximates attention by a low-rank matrix [163] (section 3.4.3).
Performer attention	Employs a custom kernel method called *Fast Attention Via Orthogonal Random features* to approximate attention [28] (section 3.4.3).
TransformerXL attention	Attention model that injects a form of relative positional encoding to its keys and queries [33] (section 3.4.3).
Attention free full attention	A model which eliminates the need for dot product-based attention. Instead, it uses a simpler multiplication of keys, values and queries [185] (section 3.4.3).
FNet attention	Replaces attention with Fourier Transforms [80] (section 3.4.3).
Synthesizer attention	Substitutes attention with random alignment matrices [153] (section 3.4.3).
Conformer attention	A model which combines attention with convolution [59] (section 3.4.3).

Table 4.3: A list of Transformer attention models (part 2).

actual design, this phase also aims to determine the optimal configuration of all aspects of the encoding.

4.5.1 Search Space

From the search spaces presented in subsection 3.2.1 and applied to Transformers in subsection 3.3.1, only multi-branch search spaces (section 3.2.1 and section 3.3.1), cell-based search spaces (section 3.2.1 and section 3.3.1) and hierarchical search spaces (section 3.2.1) constitute feasible options for the neural architecture search system.

The core structure of the search space being designed was inspired by the work of Zoph et al. [195], Real et al. [128], and Real et al. [129] who utilized variants of NASNet search space (see section 3.2.1 for more details). However, similarly to So et al. [143], the search space was enhanced to allow representation of the original Transformer stages (subsection 2.5.3) and to express new concepts found in the recent Transformer-based architectures (described in section 3.4).

The search space consists of 4 levels of concepts organized in a hierarchy. All concepts are illustrated in Figure 4.2. The root concept is referred to as *Genotype* and represents the whole Transformer decoder or encoder stage. Each Genotype has several evolvable hyperparameters which define various aspects of the represented architecture. A list and description of them is provided in Table 4.4. More details about how evolvable parameters work is provided in the following subsection 4.5.2. In addition to hyperparameters, Genotype holds a layer representation called *Genotype layer*. Genotype layer is conceptually identical to Transformer layer. Each Genotype holds one Genotype layer which is stacked x amount of times as defined by hyperparameters i.e. output of one layer constitutes input to the following layer. The purpose of the layer is to hold a number of *Genotype cells* which represent its internal structure and to keep a list of hidden states. Each element in the hidden state list stores output of one Genotype cell. Zeroth state is an exception to this rule as it is used to represent input of the layer. Each Genotype cell has two inputs which come from the hidden state list. Every cell can use any of the preceding cells' outputs or the layer input (i.e. first hidden state in the list) as its own inputs.

Parameter	**Type**	**Description**
patch size	*int*	Size of patches that a time series is divided into before entering the model
embedding dim	*int*	Embedding dimension of the represented Transformer model
layer count	*int*	Number of stacked layers which make up the encoder or decoder
use memory	*bool*	If to use inter-layer memory as proposed by Dai et al. [33] (details provided in section 3.4.3)
merge before regression	*bool*	The number of sequence elements which are passed to the final fully-connected layer for regression. *True* stands for all elements [184] while *false* means just the first one [34].
dropout probability	*float*	Global dropout probability

Table 4.4: A list of genotype hyperparameters.

The internal structure of each cell is made up of two independent branches. Each branch keeps a stack of *Genotype modules* which represent concepts commonly found in Transformer architectures. More detailed description of the module concept is provided in subsection 4.5.3. Several ways of stacking modules in each branch were proposed. Zhang et al. [187] suggested to always use two modules, one per each branch. So et al. [143] proposed using at 3 modules per each branch but their presence is optional. However, the modules have defined categories which must be preserved e.g. the first module must perform normalization and the third module must always be an activation function. This work introduces another alternative which does not restrict the number of modules per branch nor their types. After both input states are passed through their respective branch, the results are merged together via addition and sent out as the cell output.

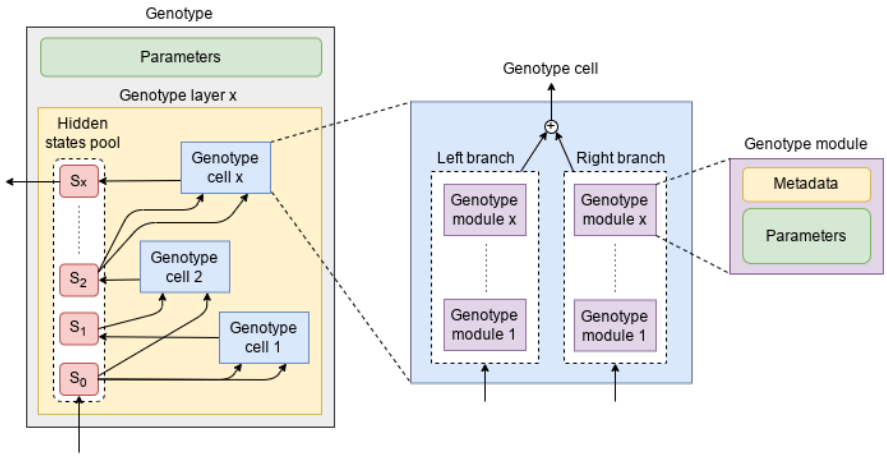

Figure 4.2: A diagram illustrating *Genotype*, *Genotype layer*, *Genotype cell* and *Genotype module* concepts and the relations between them.

To build and host an actual Transformer model which can be trained and used, the system introduces *Phenotype* concept. Phenotype is structured in the same way as Genotype is i.e. has *Phenotype layer*, *Phenotype cell* and *Module instances*, however, it is built based on primitives of the utilized machine learning library. The library is called *PyTorch* [117]. The role of the phenotype is two-fold: 1.) to provide abstraction from the machine learning library, 2.) to host additional logic which is not a subject to evolution. In this case, the additional logic consists of modules dealing with time series patch embedding (section 4.2)

and regression. Still, these fixed modules can be controlled indirectly via Genotype hyperparameters. Additionally, Phenotype has two user defined parameters which specify *input* and *output sizes* of the model. These sizes are usually set based on the dimensions of training examples (section 4.6.2). Thanks to the similarity of Phenotype and Genotype concepts, genotype to phenotype conversion is in fact pure mapping.

The described representation is capable or representing Transformer encoder or decoder while keeping flexibility to evolve architectures that go beyond the commonly used Transformer configurations which aligns with the goals of this work.

4.5.2 Parameters

As mentioned in the previous section (subsection 4.5.1), the system has a concept of *evolvable parameters*. These parameters themselves hold information about how they can be mutated. They are used to evolve hyperparameters of Genotypes or to parametrize Genotype modules. Four parameter types are available:

1. **Constant parameter:** The simplest of all parameters, represents a constant value that cannot be changed.
2. **Set parameter:** A parameter that can take any value from a provided set of values. Mutation samples a new value from the set.
3. **Range parameter:** A parameter which can sample any value from a given range. It needs to be numerical.
4. **Neighbourhood parameter:** This parameter uses a concept of *limits* and *steps*. When the parameter is mutated, its value can be changed by $+step$ or $-step$. If the value reaches one of the limits, it's clamped to the range defined by them. The parameter also needs to be numerical.

4.5.3 Modules

This section presents the concept of *modules*. For clarification, the section makes no difference between Genotype and Phenotype modules, as the latter is in fact an instance of the former. The main purpose of modules is to encapsulate concepts found in Transformer-based and other architectures potentially suitable for time series forecasting. The system provides a wide variety of modules (see section 4.7) which leaves the evolution with a great freedom to select modules that are the most suitable. In contrast to the definition of Transformer structure by layers and cells, modules represent model behaviour.

Each module has one input and one output, both having the same shape. The actual dimensions are: *(batch size, input size, embedding dimension)* where input size is identical to Phenotype input size and embedding dimension represents Transformer embedding size (subsection 2.5.3). Every module has the same input and output dimensions which allows every module to be used in place of every other module. Modules can define a list of evolvable parameters (subsection 4.5.2) which alter their behavior. Examples of such parameters are: kernel size, activation function or dropout probability. The hyperparameters defined on genotype/phenotype level are passed to each module as constants, however, their values can be overridden by module parameters if needed. Also, modules have metadata attached such as their name or category.

4.6 Neural Architecture Search System Design

The fourth phase describes the design of the evolution-driven NAS system which builds on the genotype encoding introduced in the previous phase (section 4.5).

4.6.1 Search Strategy

As explained and justified in section 4.2, the employed search strategy utilizes *genetic algorithms*. The following sections present details about *selection* (section 4.6.1) and the proposed *mutation* (section 4.6.1) and *crossover* (section 4.6.1) genetic operators.

Selection

Similarly to Zhu et al. [192], *tournament selection* is used to avoid costly evaluation of many genotypes to obtain their fitness for comparison. In this case, four tournament selection criteria were considered as listed and described in Table 4.5. The simple *best* and *worst fitness* criteria were inspired by the work of So et al. [143] and Kim et al. [73] who successfully applied them to evolution of Transformer-based architectures. *Oldest genotype* criterion was introduced by Real et al. [129] who argued that it makes the evolutionary search more resilient to noise.

Mutation

Mutation operators heavily depend on the used genotype representation (subsection 4.5.1). The used hierarchical representation allows and potentially requires different types of mutations for each of its levels. For that reason, the employed

Selection criteria	Applicability	Description
best fitness	parent selection survivor selection replacement selection	Genotype with the highest fitness is selected.
worst fitness	replacement selection	Genotype with the lowest fitness is selected.
oldest genotype	replacement selection	Genotype with the lowest creation timestamp is selected.
random genotype	replacement selection	Random genotype is selected.

Table 4.5: Selection criteria for tournament selection.

operators were designed for each level separately. There are three basic mutation levels: *layer*, *cell*, and *module*. The operators available for each level are presented in Table 4.6, Table 4.7 and Table 4.8 respectively.

Layer mutation	Description
add empty layer	Empty genotype layer is added to the genotype.
duplicate layer	Random genotype layer is duplicated and inserted back at a random index.
swap two layers	Two genotype layers swap their placement.
remove layer	Random genotype layer is deleted.

Table 4.6: Available *layer level* mutations.

In addition to mutations on different genotype levels, modules or genotypes themselves can hold several evolvable parameters which affect their behavior (subsection 4.5.2). Hence, two mutation operators dedicated to modifying the parameters were also included. The first operator alters genotype-level parameters while the second one performs changes of parameters on randomly selected modules. A brief summary is given in Table 4.9.

Cell mutation	Description
add empty cell	Empty genotype cell is added at a random place within a genotype layer. Its inputs are randomly selected from the pool of hidden states.
duplicate cell	Randomly selected cell is duplicated and inserted at a random place within a genotype layer. The inputs of the new cell are randomly selected from the pool of hidden states.
swap two cells	The placement of two randomly selected cells is swapped. Their inputs are swapped too.
change cell inputs	The inputs of a randomly selected genotype cell are reselected from the pool of hidden states.
remove cell	Randomly selected cell is removed.

Table 4.7: Available *cell mutations*. The layer within which cell mutation takes place is selected randomly.

Crossover

Crossover operators were derived from the designed mutations (section 4.6.1). There are three provided operators which are basically extensions of mutation operators to two genotypes. A summary is given in Table 4.10.

4.6.2 Performance Estimation Strategy

To obtain the fitness of candidate architectures, each candidate is trained from scratch on a training dataset and then evaluated on a test dataset. The final test loss value serves as a measure of architecture quality i.e. fitness (the lower the loss, the better the candidate is). Unfortunately, this method is very time and resource consuming. Several alternatives and modifications of the performance estimation process were proposed attempting to alleviate the issue (subsection 3.2.3). A common choice for Transformer-based architectures seems to be One-shot search (section 3.3.3) i.e. sharing weights from a pretrained super network (section 3.2.3). However, Yu et al. [181] argues that weight sharing decreases performance and leads to inaccurate evaluation of candidates. For that reason, it was decided to stick to training from scratch while employing methods which can make the approach more efficient (section 4.6.2).

Module mutation	Description
add module	Module selected from a list of available modules is added at a random place within a genotype cell.
duplicate module	Randomly selected module is duplicated and inserted at a random place within a genotype cell.
swap two modules	The placement of two randomly selected modules is swapped.
swap module for new	Randomly selected module is swapped with a new module selected from a list of available modules. The old module is discarded.
remove module	Randomly selected module is removed.

Table 4.8: Available *module mutations*. The layer and cell within which module mutation takes place are selected randomly.

Training Examples

Time series datasets need to be turned into training examples before they can be used for model training. An approach utilized by Wu et al. [170], Zhou et al. [190] and Yi et al. [180] uses a sliding window to create patches of consecutive series values followed by patches which act as labels (see section 3.5). The same technique was implemented in the designed system.

Initialization of Population

Given the complex genotype representation (subsection 4.5.1), there needs to be some focus on a proper population initialization. While using a similar representation, So et al. [143][144] experienced problems with random initialization. The search struggled to find strong architectures and often failed to converge. The issues were mitigated by using a form of smart initialization. The search was warm started by seeding the initial population with the original Transformer architecture. In a similar way, Tsai et al. [157] seeded the population with the encoder stage of Transformer. The search system described in this work also has the option to include known Transformer models in the initial population.

Parameter mutation	Description
modify genotype parameter	Randomly selected genotype parameter is assigned a new value randomly selected from its list of available values.
modify module parameter	Randomly selected module parameter is assigned a new value randomly selected from its list of available values.

Table 4.9: Genotype and module *parameter mutations*.

Crossover	Level	Description
swap random layers	layer	Swap randomly selected layers between two genotypes.
swap random cells	cell	Swap randomly selected cells between two genotypes.
swap random branches	cell	Swap a randomly selected cell branch between two genotypes.

Table 4.10: Crossover operators for all genotype levels.

Progressive Dynamic Hurdles

The *Progressive Dynamic Hurdles* method was first proposed by So et al. [143] and later used by Kim et al. [73] and So et al. [144]. The core idea is to stop training of badly performing candidates early, so they do not waste additional resources nor take more time to evaluate. The method is based on allocation of additional training steps to promising candidates and constitutes the default option in the designed system.

Each candidate is allocated an initial number of training steps s_0. After k_1 iterations of the evolutionary loop, mean fitness of the whole population is calculated and used to establish a hurdle h_1. Then, only the candidates which have fitness higher or equal to the established hurdle get an additional amount of training steps s_1. The rest of the candidates is not trained beyond s_0 training steps. The situation repeats until $k_1 + k_2$ iterations is reached. At that point, mean fitness of the population is calculated again and a new hurdle h_2 is put in place. Then, only the candidates with fitness higher than h_2 can train for $s_0 + s_1 + s_2$ training

steps. The whole process repeats until a max number of training steps is reached or the evolutionary loop terminates. In the designed system, $k_1, k_2, k_3...$ are usually set to the same value and the number of training steps represents epochs that a candidate model can train for.

Difference Factor

Difference factor is a custom metric created to track diversity in the population. As the evaluation of candidates is expensive, the metric is able to provide assessment of diversity without the need to train all candidates in the population and evaluate them for fitness. Under the assumption that the same genotype always reaches the same fitness when trained, the metric pair-wise compares all genotypes in the population and calculates the mean amount of difference between them. The result is a value between 0 and 1, where 1 means max theoretical diversity and 0 means that all genotypes are identical. Source code for the metric calculation is provided in Appendix (section 6.3).

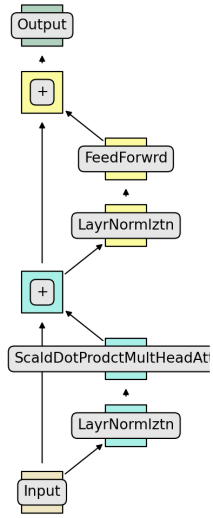

Figure 4.3: Visualization of a Genotype layer representing one layer of pre-norm (section 3.4.2) Transformer encoder (section 3.4.1). Squares with labels represent Genotype modules. Modules of the same color belong to the same Genotype cell.

4.6.3 System Configuration

The neural architecture search system can be configured via JSON configuration files. There are four types of configuration files available: *dataset config* which determines what dataset is used and how it is transformed into training examples, *evaluation config* specifying metrics used for evaluation and configuring Progressive Dynamic Hurdles (section 4.6.2), *evolution config* which configures the main aspects of evolution including initialization, tournament selection, crossover and mutations, *genotype config* which defines a template Genotype (subsection 4.5.1) with evolvable parameters (subsection 4.5.2) for population initialization and finally *module registry* providing a list of modules (section 4.7) that can be utilized by evolution.

4.6.4 Result Presentation

To save system configurations, track metrics and store the discovered architectures, TensorBoard [1] is used. During each evolution run, survivor fitness, difference factor (section 4.6.2) and the established hurdles (section 4.6.2) are recorded per iteration and can be viewed live as the search progresses. All data is also stored to disk, so it can be inspected even after the search finishes. In addition, the system includes a custom tool for Genotype visualisation. The tool was designed to render Genotype as a graph showing its internal structure and placement of modules. An example of a Genotype representing Transformer encoder is shown in Figure 4.3.

4.7 Transformer Architecture Evolution

The goal of the fifth phase is to evolve a strong Transformer-based architecture by using the system designed in the previous phase (section 4.6). So far, the system is still missing an important ingredient which in fact makes it work: a diverse set of modules which encapsulate concepts found in successful Transformer-based and other architectures used for time series forecasting.

The following sections present the available modules category by category while providing a brief description of them and justification for their inclusion in the system. In general, modules vary in complexity. Some modules represent only a simple concept such as an activation function while other modules contain complex architectural blocks.

4.7.1 Transformer Building Blocks

In order to represent a complete Transformer, several core modules are needed. All these modules were extracted from the original Transformer architecture proposed by Vaswani et al. [159] (subsection 2.5.3). The list of modules is provided in Table 4.11.

Module	**Description**
Sinusoidal absolute positional encoding	This module encapsulates the absolute positional encoding method from vanilla Transformer.
Multi-head attention	Implementation of *scaled dot product multi-head attention* i.e. the core concept of Transformer.
Feed forward	Feed forward stack in Transformer consists of two fully connected layers and ReLU activation function in between. Embedding dimension of the module input is first expanded by the first fully-connected layer, activation function is applied and the second layer squeezes the dimension back to its original size. The expansion ratio used by the layers is commonly called *forward expansion*.
Layer normalization	A module performing layer normalization.

Table 4.11: A list of modules encapsulating core building blocks of Transformer.

4.7.2 Attention Modules

The attention models selected in the second phase (section 4.4) are also included in this phase but encapsulated in modules. As a reminder, the list of models is provided in Table 4.2 and Table 4.3. Additionally, several *augmentations* for either attention scores or key, query and value matrices were proposed. Simply put, augmentations are optional components which perform data preprocessing. Most of the provided augmentations are various forms of MAM positional encodings which were presented in subsection 3.4.5. To keep the attention modules simple, these augmentation options were built only into *multi-head attention* module (the attention model used in the original Transformer, see Table 4.11). The list of available augmentations is given in Table 4.12.

4.7.3 Convolution Modules

The concept of convolution (subsection 2.2.1) played an important role in many works related to Transformers (section 3.4.3). Convolution is also used in neural architectures for time series forecasting (section 3.6). For that reason, several modules encapsulating common uses of convolution are included. The modules are listed and described in Table 4.13.

4.7.4 Feed Forward Modules

In addition to *Feed forward* module based on the original Transformer, a single fully-connected layer is included as a separate module called *Single feed forward*. This allows the evolutionary search to manipulate even such simple concepts and potentially combine them into new architectural patterns.

4.7.5 Activation Modules

Several activation functions were included as independent modules. The subset was chosen based on the findings from section 3.4.4. Along with a brief description, the function modules are enumerated in Table 4.14.

4.7.6 Positional Encoding Modules

The positional encoding methods presented in subsection 3.4.5 are of two kinds: MAM and APE. MAM encoding methods have already been included as augmentations proposed for multi-head attention module (subsection 4.7.2). The promising APE candidates are provided as modules. The list of included APE encodings is given in Table 4.15.

4.7.7 Normalization Modules

The original Transformer architecture (subsection 2.5.3) relies on a normalization technique called Layer Normalization. However, more recent works proposed alternative normalizations for Transformers (subsection 3.4.2). To capture their potential suitability for time series forecasting architectures, several promising candidates were built into modules. The modules are described in Table 4.16.

4.7.8 Dropout Modules

The last group of modules consists of dropout modules. These modules encapsulate three dropout variants which might be helpful in Transformers. As usual, the list of modules is provided in a table, Table 4.17.

Augmentation	Placement	Description
Rotary positional encoding	key, query or value matrix	Positional encoding represented by a rotation matrix [148]
TUPE absolute positional encoding	key, query or value matrix	*Transformer with Untied Positional Encoding* [72] uses two different ways of adding positional encoding. The first way proposes adding absolute positional encoding directly to key and query matrices.
Spatial depth-wise convolution	key, query or value matrix	The work of So et al. [144] found beneficial to postprocess key, query and value matrices with spatial depth-wise convolution.
DeBERTa relative positional encoding	attention scores	Fixed relative positional encoding added directly to attention scores as used in DeBERTa [62]
T5 relative positional encoding	attention scores	Learnable relative positional encoding added to attention scores as used in T5 Transformer [125]
TUPE relative positional encoding	attention scores	The second way of adding positional encoding in *TUPE* [72] i.e. add relative positional bias to attention scores.

Table 4.12: A list of all possible augmentations in multi-head attention module. *Placement* column specifies where in the attention module can the particular augmentation be applied.

Module	Description
Convolution 1D	Convolution performed over embedding dimension for each sequence element separately.
Convolution 2D	Convolution performed over all sequence elements and their embeddings at the same time.
Max pooling 1D	Max pooling performed over embedding dimension for each sequence element separately.
Max pooling 2D	Max pooling performed over all sequence elements and their embeddings at the same time.
Depth-wise separable convolution	Convolution which splits channel-wise and spatial-wise computation into two independent steps. Sequence length is interpreted as the spatial dimension while embeddings are represented as channels.
Inverted bottleneck	A module similar to Transformer *Feed forward* module (Table 4.11) which uses convolutions instead of fully-connected layers. This variant was introduced by Liu et al. [94].
CoordConv	The module provides functionality similar to *Convolution 2D*. In addition, before applying convolution, it adds two channels representing element position in the 2D input. One channel represents vertical position while the second one horizontal position. This modification was originally proposed by Liu et al. [91]

Table 4.13: A list of modules encapsulating the common uses of convolution.

Module	Description
ELU	*Exponential Linear Unit* originally proposed by Clevert et al. [30]
GeLU	*Gaussian Error Linear Unit* introduced by Hendrycks and Gimpel [63]
ReLU	*Rectified Linear Unit* which was used in the original Transformer [159].
SiLU	*Sigmoid Linear Unit* also known as *Swish* [127].
Leaky ReLU	An activation function based on ReLU which allows a small slope for negative values.
Squared ReLU	An enhancement of ReLU activation discovered by neural architecture search [144].

Table 4.14: A list of modules that provide activation functions.

Module	Description
Axial positional encoding	Positional encoding capable of efficiently storing long sequences [74].
BERT learned positional encoding	Positional encoding by Devlin et al. [35] which is learned completely from training data.
Simple absolute positional encoding	A simple positional encoding which adds bias ranging $0 - 1$ to all embedding dimensions based on element positions.

Table 4.15: A list of modules which implement APE positional encodings.

Module	Description
AdaNorm	Layer normalization which contains no learnable parameters [173].
PowerNorm	Batch normalization adapted for use in Transformers [140].
Scaled L2 normalization	L2 normalization scaled by a single parameter leads to a better performance of Transformers [114].
Root mean square layer normalization	A more lightweight version of layer normalization [186].

Table 4.16: A list of modules providing different kinds of normalization.

Module	Description
Alpha Dropout	Dropout method which maintains mean and standard deviation [75].
Dropout	Randomly zeros out values in the embedding dimension.
Dropout 2D	Randomly zeros out whole elements in the sequence.

Table 4.17: A list of dropout modules.

4.8 Forecasting Methods Comparison

The last phase focuses on the evaluation of the architectures produced during the previous phase (section 4.7). For this purpose, several commonly used forecasting methods are trained on the same dataset together with the evolved architectures, and their forecasting accuracy on the corresponding test dataset is evaluated and compared. The set of considered methods was decided based on the reviews conducted in subsection 2.4.1, subsection 2.4.2 and section 3.6 and includes: *ARIMA*, *Temporal Convolutional Network*. *Recurrent Neural Network*, *LSTM* and *N-BEATS*. Additionally, the original *Transformer* (subsection 2.5.3) is also included to see if the evolved architectures outperform it. Unfortunately, Prophet and NeuralProphet methods from section 3.6 cannot be utilized as they require time series annotated with timestamps (section 4.2).

Chapter 5

Experiments and Results

This chapter presents experiments conducted to gain knowledge about using Transformers for time series forecasting and to verify the effectivity and efficiency of the designed neural architecture search system. In section 5.1, experimental plan describing the conducted experiments is introduced. In the next section (section 5.2), experimental setup provides details about the used time series datasets and evaluation metrics. The result of each experiment is described in section 5.3. The experiments were performed during *Evaluation* phase of *Design Science Research* methodology (section 1.4).

5.1 Experimental Plan

The experiments were conducted in phases which map to certain phases of the design framework introduced in Method and System Design (chapter 4). The purpose of each phase is to gradually build understanding of the behaviour of Transformer-based architectures when used for forecasting time series. One phase can be split into multiple experiments to impose logical grouping or emphasize iterative buildup of knowledge. Each phase and the corresponding experiments are described below. More detailed description of each experiment is provided in section 5.3 which also presents experimental results.

- **Phase 1:** The goal of the first experimental phase is to select a Transformer variant suitable for time series forecasting in order to obtain a base architecture for genotype encoding design. As described in section 4.3, there are several alternatives to choose from.
  - **Transformer hyperparameter search:** The first experiment aims to find the optimal values of hyperparameters for all candidate variants

to allow their fair comparison. The values are obtained by performing evolutionary hyperparameter search.

  - **Transformer architecture evaluation:** The second experiment conducts forecasting accuracy comparison by evaluating the optimally parametrized candidate variants on several datasets.

- **Phase 2:** The second phase focuses on alternative attention models and the assessment of their performance in time series forecasting Transformers. The best-performing variant from the previous phase is used as a host for comparing the attention models preselected in section 4.4. The discovered strong models are then included in the later evolutionary search as modules.

  - **Attention model evaluation:** The attention models are one by one injected in the architecture selected during Phase 1 parametrized by the same hyperparameters. Then, the forecasting performance of all instances is measured on several datasets and compared.

- **Phase 3:** The third phase tries to determine the optimal genotype encoding configuration capable of evolving the Transformer variant from Phase 1 beyond its commonly used variations while producing architectures achieving better time series forecasting accuracy.

  - **Genotype representation comparison:** The genotype encoding designed in section 4.5 has several aspects which can be tweaked. This experiment seeks to find the optimal configuration of these aspects to produce encoding which evolves the most performant architectures.

- **Phase 4:** The last phase compares the time series forecasting methods presented in section 4.8 with Transformer-based architectures evolved by the neural architecture search system described in section 4.6 and section 4.7.

  - **Time series forecasting methods comparison:** Several time series forecasting methods are used to compare their performance against the evolved Transformer-based architectures. All methods and architectures are evaluated on the same benchmark dataset.

5.2 Experimental Setup

This section presents datasets, evaluation metrics and other configuration used by all experiments unless specified otherwise. The experiments were run on Amazon EC2 [106] cloud service using *g4dn.xlarge* virtual instances equipped with Nvidia T4 GPUs featuring 16 GB of memory and specialized Tensor cores to accelerate training of neural network-based models.

5.2.1 Datasets

Two types of datasets were used: 1.) a custom designed *Synthetic dataset* that provides simple time series which are easy to explain and reason about and 2.) datasets sampled from *Libra benchmark* (section 3.7) providing a reasonable proxy for real-world time series.

Synthetic dataset consists of three synthetic time series: 1.) *single sine wave*, 2.) *composed sine waves* and 3.) *composed sine waves with noise*. For illustration, the series are shown in Figure 5.1. The series were generated based on the equations and parameter values provided in Table 5.1. In contrast to the fixed parameters, the length of each series can be specified as needed. Throughout the experiments, the length of all series was set to 5000 elements. The series were also designed and verified (by *Augmented Dickey-Fuller test* [37]) to be stationary (subsection 2.4.1) in order to keep the forecasting difficulty of the dataset low.

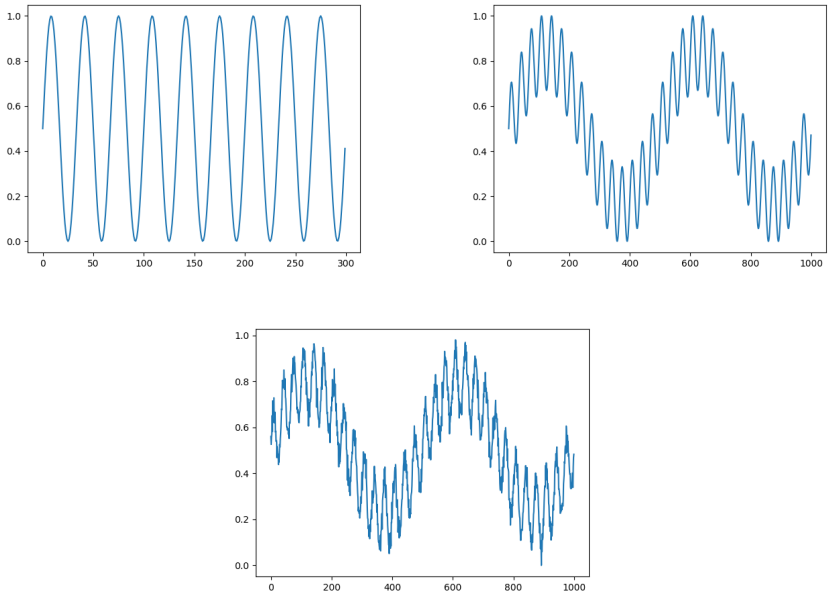

Figure 5.1: Visualization of the three types of time series included in *Synthetic dataset*. Upper left: *single sine wave*, Upper right: *composed sine waves*, Bottom center: *composed sine waves with noise*.

Series	Definition	Parameter values
Single sine wave	$y(x) = sin(2\pi fx) \cdot a$	$f = 0.002, a = 1.0$
Composed sine waves	$y(x) = sin(2\pi f_1 x) \cdot a_1 + sin(2\pi f_2 x) \cdot a_2$	$f_1 = 0.002, a_1 = 1.0,$ $f_2 = 0.03, a_2 = 0.5$
Composed sine waves with noise	$y(x) = sin(2\pi f_1 x) \cdot a_1 + sin(2\pi f_2 x) \cdot a_2 + \mathcal{N}(\mu, \sigma^2)$	$f_1 = 0.002, a_1 = 1.0,$ $f_2 = 0.03, a_2 = 0.5$ $\mu = 0, \sigma = 1$

Table 5.1: The equations which produced the series included in *Synthetic dataset*. Each equation was parametrized by the corresponding set of parameter values.

By default, Libra benchmark is split into 4 categories (*Economics*, *Finance*, *Human access*, *Nature and demographics*) with 100 time series per category. As training a model on the whole benchmark takes a long time, the full benchmark did not serve well as a proxy for candidate architecture performance during neural architecture search. In addition, preliminary testing has shown that using the whole benchmark and forcing a model to learn all time series at once results in poor performance for all tested forecasting methods. For that reason, *two sub-sampled variants of the benchmark* containing only *10 time series per category* were introduced. The first variant was used to estimate the performance of candidate architectures during evolutionary searches whereas the second variant was employed for comparing accuracy of forecasting methods in the last experimental phase. The series were randomly sampled under one assumption: the selected series need to be able to produce at least n training examples i.e. have a certain minimal length. Optionally, each series can be cropped to contain only the first k samples to further reduce the size of the sub-sampled benchmarks. For both benchmarks, n and k were kept constant during all experiments to ensure fair comparisons. More specifically, n was set to 100 and k to 5000. The indices of the series selected for the first benchmark are provided in Table 5.2 and the indices of the series used in the second benchmark are listed in Table 5.3.

Stationarity of any of the time series in Libra benchmark is not guaranteed and in reality, it does not hold for many of them. Based on *Augmented Dickey-Fuller test* [37], only 20% of the series in *Economics*, 12% in *Finance*, 72% in *Human access* and 75% in *Nature and demographics* category is stationary.

The series of all used datasets/benchmarks were deliberately subjected to only a minimal amount of data preprocessing. The intention is to force the evolved models to learn representations from raw time series in order to make them easily

Libra category	Randomly sampled time series indices
Economics	[71, 86, 69, 75, 70, 83, 80, 74, 81, 77]
Finance	[63, 96, 49, 39, 71, 18, 99, 66, 82, 17]
Human	[86, 56, 46, 21, 67, 5, 2, 92, 68, 97]
Nature	[0, 91, 39, 70, 23, 42, 73, 1, 53, 24]

Table 5.2: The first sub-sampled Libra benchmark variant presented as indices of the randomly selected time series in each category.

Libra category	Randomly sampled time series indices
Economics	[68, 84, 73, 88, 87, 79, 72, 76, 26, 85]
Finance	[61, 81, 72, 41, 57, 94, 83, 53, 19, 88]
Human	[99, 27, 65, 20, 17, 69, 47, 61, 13, 38]
Nature	[33, 62, 66, 55, 46, 43, 26, 92, 18, 20]

Table 5.3: The second sub-sampled Libra benchmark variant presented as indices of the randomly selected time series in each category.

applicable and transferable. Preliminary testing showed that the sole normalization of values to $0.0 - 1.0$ range is sufficient. To generate training examples, the windowing method selected in section 4.6.2 was used. The length of the first patch maps to the *model input size* while the length of the second patch represents the *model output size* i.e. the forecasting horizon. Before training, the produced examples are shuffled but in a deterministic way. This is done to mitigate a problem dubbed *floating fitness* observed on models getting trained on the same examples but in a different order. In some cases, this caused models to suddenly perform better or worse even though no changes were made to them yet. To obtain a separate *training* and *test set*, the set of all training examples was split by *80:20* ratio in favor of the training set.

5.2.2 Evaluation Metrics

To compare the performance of different candidate architectures or forecasting methods, two evaluation metrics were used throughout all experiments: *MAE* and *RMSE*. The second metric was included to allow better assessment of outliers in predictions. For explanation of the metrics, see subsection 2.4.3. If there was a need to compare metric values, *One-way* or *Two-way ANOVA* [36] methods were used to ensure that only statistically significant differences are being

considered. The former method was applied in cases which dealt with one independent variable whereas the latter in cases that included two independent variables. Importantly, ANOVA assumes *homogeneity of variances* i.e. that the variances between the considered groups are equal and that the *residuals* (also called *experimental error*) are *approximately normally distributed*. Therefore, before each application of ANOVA, *Shapiro-Wilk test* [136] was used to confirm the normal distribution of residuals and *Levene's test* [81] to check the homogeneity of variances. The *p-value* defining statistical significance was set to 0.05.

5.2.3 Training of Models

For models training, *AdamW* [97] optimizer with $\beta_1 = 0.9, \beta_2 = 0.98, \lambda = 0.01$ and $\epsilon =$ 1e-9 was used. To avoid potential training instabilities (section 3.4.2), *learning rate* was scheduled based on the same algorithm as proposed by Vaswani et al. [159]. The so-called *Noam* optimizer modifies learning rate based on the number of warmup steps w and the embedding size of the model. In the experiments, w was set to 1000. Figure 5.2 shows a few example learning rate schedules based on three combinations of warmup steps and embedding sizes. The loss function used during training was *Smooth L1* [50] loss which represents a compromise between MAE, and MSE/RMSE while also being able to deal with exploding gradients problem better. For all performed training runs, the *batch size* was set to 16.

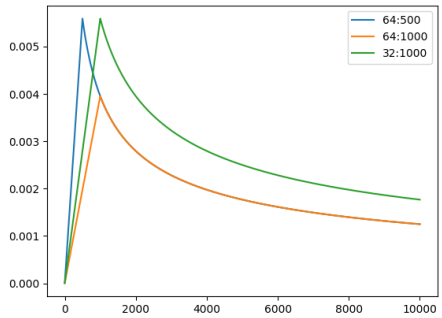

Figure 5.2: Examples of *Noam* optimizer learning rate scheduling. The curves show schedules of three combinations of embedding size and warmup steps. The vertical axis represents learning rate while the horizontal axis shows the current training step index.

5.3 Experimental Results

The following sections walk through all experimental phases described in section 5.1 while presenting results of the experiments conducted during each of them.

5.3.1 Phase 1: Transformer Architectures

The first phase presents results of the experiments aiming to select a suitable Transformer architecture variant as described in section 4.1. The first section shows and assesses the obtained Transformer hyperparameters whereas the second section compares and discusses the accuracy achieved by each of the candidate architectures.

Ex1: Transformer hyperparameter search

To conduct the hyperparameter search, the list of parameters shown in Table 5.4 was used as a genotype. Parameter types and their available values were set based on preliminary testing to make sure the configuration is not too restrictive. Other aspects of the evolutionary search were configured as specified in Table 5.5.

Parameter	**Parameter type**	**Available values**
patch size	Set	[1, 5, 10, 25, 50]
embedding dim	Set	[8, 16, 32, 64, 128]
head count	Set	[1, 2, 4, 8]
layer count	Range	(1, 16), step: 1
forward expansion	Range	(1, 8), step: 1
dropout probability	Range	(0.0, 0.5), step: 0.05

Table 5.4: The list of parameters used as a genotype during Transformer hyperparameter search. The type column specifies the used evolvable parameter types as defined in subsection 4.5.2

The search was executed two times. The first run used *Synthetic* dataset while the second run utilized the first *sub-sampled Libra* benchmark. The results of the former search are shown in Table 5.6 and of the latter in Table 5.7. Let's first focus on the results of the first search (Synthetic dataset). *Embedding dim* is mostly trending between 32 and 64 with *merged encoder-only* variant being the only exception to this rule with its value of 8. The situation is similar with

Concept	**Configuration (value)**
datasets for evaluation	[Synthetic, sub-sampled Libra 1]
model input size	200
model output size	50
iterations	150
population size	80
initialization	genotypes with randomized values
parent selection	best fitness tournament (size: 2)
survivor selection	best fitness tournament (size: 2)
replacement selection	worst fitness tournament (size: 2)
crossover	uniform (probability: 0.3)
mutation	randomly change value (probability: 0.1)
evaluation epochs	5
search repeated	10x per dataset/architecture combination

Table 5.5: The setup of the evolutionary hyperparameter search.

patch size, where *merged encoder-only* performed the best with the patch size of 50 while the rest of architectures were mostly between 1 and 10. *Decoder-only* is also worth mentioning as the patch size of 1 was selected for it during every run. *Head count* is mostly between 2 and 4 apart from *encoder-decoder* which was trending between 4 and 8. *Layer count* of *encoder-only* and *merged encoder-only* was the lowest, only between 1 and 2 layers. The remaining architectures were around 5 (*encoder-decoder*), 7 (*encoder-decoder*) and almost 12 (*Informer encoder-decoder*). The last number is unusually large given the fact that in both *encoder-decoder* variants, this number sets the number of layers for both encoder and decoder i.e. results in double the number of layers in total. *Forward expansion* was mostly between 2 and 3 but around 5 for *encoder-decoder* which is close to its default value. The lowest value i.e. 1 was selected for *Informer encoder-decoder*. This seems to somewhat counteract the high number of layers in *Informer encoder-decoder* as mentioned earlier. *Dropout probability* was very low in general, all architectures were between 0.01 and 0.02 while *decoder-only* being the only exception with the value of 0.08.

The results on the *sub-sampled Libra* dataset differed slightly. When it comes to *embedding dim*, apart from *encoder-decoder*, there is a noticeable increase suggesting that more complex series need larger embedding sizes. The most prominent

Params	Encoder decoder	Informer encoder decoder	Decoder only	Encoder only	Merged encoder only
embedding dim	64±0	57.6±12.8	57.6±12.8	38.4±12.8	8±0
patch size	3.4±2	1.8±1.6	1±0	7±2.45	50±0
head count	6.4±2	2.4±0.8	2±1.1	2.6±1.2	2.8±0.1
layer count	4.8±0.75	11.8±4.12	7.4±2.73	1.6±0.8	1.6±0.5
forward expansion	4.6±1.36	1±0	3±1.26	2.6±0.8	2.4±0.5
dropout probability	0.03±0.02	0±0	0.15±0.08	0.01±0.01	0.01±0.01

Table 5.6: Results of Transformer hyperparameter search on Synthetic dataset. The notation stands for: *mean ± std*. Note: Some of the values come from a predefined set, so *mean* and *std* need to be interpreted accordingly.

increases are in the case of *merged encoder-only* and *decoder-only*. *Patch size* also changed. For *encoder-decoder*, *Informer encoder-decoder* and *decoder-only*, all runs selected 1. Also, the patch size of *merged encoder-only* was lowered to values between 5 and 10. For all architectures, *head count* became very similar and was trending between 4 and 8. This also seems to reflect a need to capture more complex patterns in the data. *Layer count* values mostly increased, especially for *encoder-only* and *merged encoder-only* architectures for which they increased around 4x. In contrast, *Informer encoder-decoder* went down to roughly half of the layer count selected for *Synthetic* dataset. In general, the situation of *forward expansion* is very similar to head counts. The values increased and became more uniform, mostly within the range of 4-5. Finally, *dropout probability* stayed nearly unchanged compared to *Synthetic* dataset. Based on these findings, it appears that the architectures started to slightly overfit the *Synthetic* dataset, therefore, the results obtained on *sub-sampled Libra* dataset are likely closer to truly optimal hyperparameters.

Params	**Encoder decoder**	**Informer encoder decoder**	**Decoder only**	**Encoder only**	**Merged encoder only**
embedding dim	57.6±12.8	22.4±7.84	89.6±31.4	51.2±15.7	38.4±12.8
patch size	1±0	1±0	1±0	7±2.45	6.67±2.36
head count	5.8±2.86	6±2.53	6.4±1.96	5.6±1.96	5.83±3.08
layer count	7.8±3.12	6.4±0.8	12.2±2.23	8.8±1.72	7.4±1.96
forward expansion	6.6±2.33	4.6±0.5	5±1.9	6.6±0.8	6.2±2.23
dropout probability	0.03±0.02	0±0	0.03±0.02	0.02±0.01	0.01±0.01

Table 5.7: Results of Transformer hyperparameter search on sub-sampled Libra dataset. The notation stands for: *mean ± std*. Note: Some of the values come from a predefined set, so *mean* and *std* need to be interpreted accordingly.

Ex2: Transformer architecture evaluation

The second experiment utilizes hyperparameters from the first experiment to establish a fair comparison of the candidate architecture variants. To perform the comparison, two sets of hyperparameters were created, one for *Synthetic* dataset and one for the *sub-sampled Libra* dataset. The values closest to *mean* values presented in Table 5.6 and Table 5.7 were selected producing two sets of hyperparameters per architecture. If the selection was a tie between two values, the lower one was selected. The sets are listed in Table 5.8.

Dataset	**Encoder decoder**	**Informer encoder decoder**	**Decoder only**	**Encoder only**	**Merged encoder only**
Synthetic	[64, 1, 4, 8, 7, 0.03]	[64, 1, 2, 12, 1, 0]	[64, 1, 2, 8, 3, 0.15]	[32, 5, 2, 2, 3, 0.01]	[8, 50, 2, 2, 2, 0.01]
Libra	[64, 1, 4, 8, 7, 0.03]	[16, 1, 4, 6, 5, 0]	[64, 1, 8, 12, 5, 0.03]	[64, 5, 4, 9, 7, 0.02]	[32, 5, 4, 7, 6, 0.01]

Table 5.8: The sets of hyperparameters for Transformer variants evaluation chosen based on the results of hyperparameter search described in section 5.3.1.

Concept	**Configuration**
datasets for evaluation	[Synthetic, sub-sampled Libra 1]
evaluation metrics	[MAE, RMSE]
model input size	200
model output sizes (forecasting horizons)	[1, 20, 50]
trained for epochs	10
evaluation repeated	10x per dataset, horizon and variant combination

Table 5.9: The setup used for the Transformer variants evaluation experiment.

The evaluation setup is shown in Table 5.9. The comparison was performed using three different forecasting horizons to assess the suitability of each architecture for a wider variety of forecasting problems. Also, both evaluation metrics from subsection 5.2.2 were used. The results are presented in Table 5.10 and Table 5.11. As the tables show, *merged encoder-only* achieved the best performance on all tested datasets, forecasting horizons and for both evaluation metrics. One-way ANOVA confirmed that the differences in both metrics are statistically significant. For that reason, *merged encoder-only* was selected as the base architecture for the following experiments and genotype design. To provide more insight into how forecasts produced by the different variants looked like, Figure 5.3 displays forecasts of one training example from Synthetic dataset and Figure 5.4 shows forecasts of one example from the sub-sampled Libra dataset.

Transformer variant (MAE)	**Synthetic 1**	**Synthetic 20**	**Synthetic 50**	**Libra 1**	**Libra 20**	**Libra 50**
encoder decoder	0.0203 ± 0.0063	0.0432 ± 0.0135	0.0796 ± 0.0268	0.0964 ± 0.0054	0.1481 ± 0.0234	0.1731 ± 0.0222
informer encoder decoder	0.0288 ± 0.0092	0.0187 ± 0.004	0.0176 ± 0.0021	0.0757 ± 0.0067	0.0782 ± 0.0021	0.0961 ± 0.0069
decoder only	0.0171 ± 0.0011	0.0233 ± 0.006	0.0249 ± 0.0068	0.0836 ± 0.0057	0.1097 ± 0.0068	0.154 ± 0.0131
encoder only	0.0174 ± 0.0061	0.017 ± 0.0022	0.019 ± 0.0031	0.0657 ± 0.0032	0.0734 ± 0.0006	0.0842 ± 0.0012
merged encoder only	**0.0126 ± 0.0013**	**0.0157 ± 0.001**	**0.0151 ± 0.0012**	**0.0645 ± 0.0024**	**0.068 ± 0.0015**	**0.0759 ± 0.0018**

Table 5.10: Results of Transformer architecture comparison using *MAE* metric on combinations of datasets and forecasting horizons specified Table 5.9. The notation stands for: *mean ± std*. The best (lowest) values are shown in **bold**.

Transformer variant (RMSE)	**Synthetic 1**	**Synthetic 20**	**Synthetic 50**	**Libra 1**	**Libra 20**	**Libra 50**
encoder decoder	0.0203 ± 0.0063	0.0511 ± 0.016	0.0965 ± 0.0333	0.0964 ± 0.0054	0.1819 ± 0.0257	0.2099 ± 0.0267
informer encoder decoder	0.0288 ± 0.0092	0.0223 ± 0.0044	0.0211 ± 0.0023	0.0757 ± 0.0067	0.1038 ± 0.0028	0.1249 ± 0.0074
decoder only	0.0171 ± 0.0011	0.0273 ± 0.0061	0.0306 ± 0.0078	0.0836 ± 0.0057	0.1347 ± 0.0057	0.1859 ± 0.0165
encoder only	0.0174 ± 0.0061	0.0206 ± 0.0024	0.023 ± 0.0035	0.0657 ± 0.0032	0.0966 ± 0.0008	0.1104 ± 0.0018
merged encoder only	**0.0126 ± 0.0013**	**0.0191 ± 0.001**	**0.0187 ± 0.0014**	**0.0645 ± 0.0024**	**0.0891 ± 0.0016**	**0.0998 ± 0.002**

Table 5.11: Results of Transformer architecture comparison using *RMSE* metric on combinations of datasets and forecasting horizons specified Table 5.9. The notation stands for: *mean ± std*. The best (lowest) values are shown in **bold**.

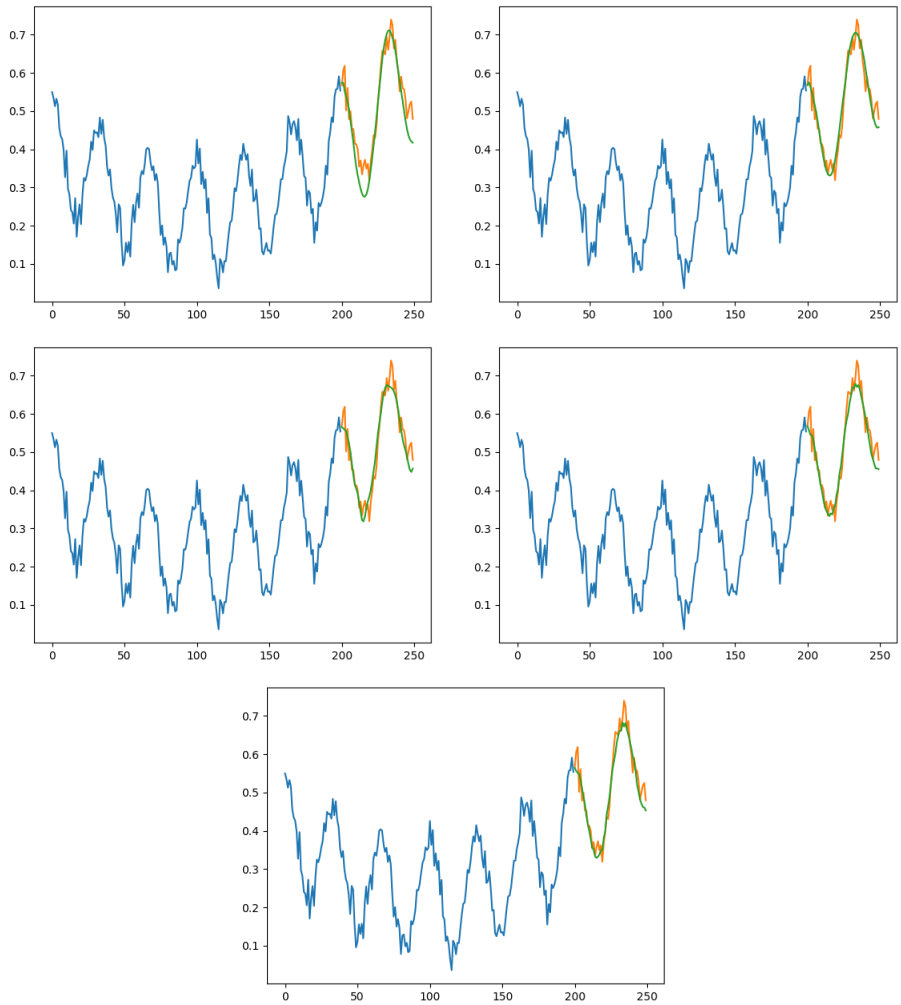

Figure 5.3: Example forecasts of one training example from Synthetic dataset produced by all architecture variants. Blue color represents the model input, orange color the true values and green color the forecasted values. Top left: *encoder-decoder*, top right: *Informer encoder-decoder*, center left: *decoder-only*, center right: *encoder-only*, bottom: *merged encoder-only*.

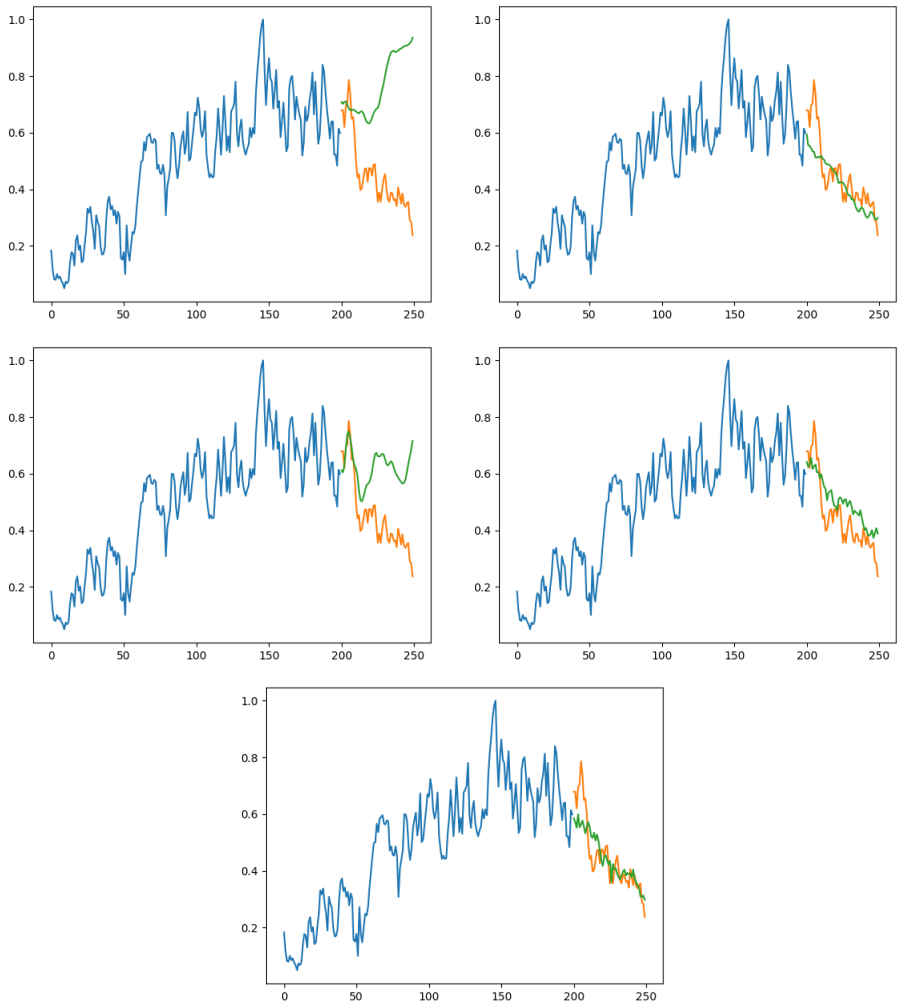

Figure 5.4: Example forecasts of one training example from the sub-sampled Libra dataset produced by all architecture variants. Blue color represents the model input, orange color the true values and green color the forecasted values. Top left: *encoder-decoder*, top right: *Informer encoder-decoder*, center left: *decoder-only*, center right: *encoder-only*, bottom: *merged encoder-only*.

5.3.2 Phase 2: Attention Models

The goal of the second phase is to perform a comparison of alternative attention models presented in section 4.4 to assess their suitability for time series forecasting. The phase consists of only one experiment which carries out the comparison.

Ex1: Attention model evaluation

To evaluate and compare the alternative attention models, a setup similar to the second experiment (section 5.3.1) from phase 1 (subsection 5.3.1) was used. However, as the objective is to measure the ability of each model to extract optimal relations between elements, the size of training examples and forecasting horizons were increased significantly to construct more challenging forecasting situations and put more strain on the attention models inside Transformers.

The *model input size* was increased to 400 and and the tested *horizons* were expanded to 100, 200 and 400. Preliminary testing also considered larger examples but encountered problems with fitting all training data into Nvidia T4's memory (section 5.2). The models were tested while injected into the best-performing Transformer architecture variant from the previous phase (subsection 5.3.1) i.e. *merged encoder-only* replacing the original multi-head attention. The attention models listed in section 4.4 were swapped in one by one creating several merged encoder-only instances. These instances were then trained and independently evaluated after training. All experimental setup details are listed in Table 5.12. The obtained results are presented in Table 5.13 and Table 5.14. The former table presents evaluation based on MAE while the latter uses RMSE metric.

Concept	**Configuration**
datasets for evaluation	[Synthetic, sub-sampled Libra 1]
evaluation metrics	[MAE, RMSE]
host architecture	merged encoder-only
model input size	400
model output sizes (forecasting horizons)	[100, 200, 400]
trained for epochs	10
evaluation repeated	10x per dataset, horizon and attention model combination

Table 5.12: The setup used for the attention models comparison experiment.

Merged encoder only (MAE)	**Synthetic 100**	**Synthetic 200**	**Synthetic 400**	**Libra 100**	**Libra 200**	**Libra 400**
passthrough attention	0.0143 ± 0.0009	0.0153 ± 0.0014	0.0139 ± 0.001	0.1059 ± 0.0015	0.1177 ± 0.0018	0.1688 ± 0.0075
multi-head attention	0.0149 ± 0.0013	0.0139 ± 0.0005	0.0142 ± 0.0011	0.0773 ± 0.0027	0.0993 ± 0.025	0.2479 ± 0.0714
adaptive span attention	0.0143 ± 0.0011	0.0144 ± 0.0009	0.014 ± 0.0006	0.0869 ± 0.0035	0.1018 ± 0.0102	0.1987 ± 0.0256
attention free full attention	0.0147 ± 0.0012	0.0146 ± 0.0019	0.0141 ± 0.0007	0.1041 ± 0.0021	0.1153 ± 0.0021	0.1659 ± 0.0074
big bird attention	0.0152 ± 0.0015	0.0144 ± 0.0007	0.0139 ± 0.0006	0.0817 ± 0.0047	0.0857 ± 0.0047	0.2011 ± 0.0364
conformer attention	0.0159 ± 0.0025	0.015 ± 0.0009	0.0155 ± 0.0014	**0.0689 ± 0.0026**	**0.0733 ± 0.002**	0.1567 ± 0.0085
synthesizer attention	0.0148 ± 0.0013	0.0148 ± 0.0013	0.0155 ± 0.001	0.1058 ± 0.0011	0.1215 ± 0.0038	0.1991 ± 0.0296
performer attention	0.018 ± 0.0036	0.0149 ± 0.0006	0.0149 ± 0.0011	0.1034 ± 0.0029	0.1143 ± 0.0047	0.209 ± 0.0506
fnet attention	0.0151 ± 0.0017	0.0166 ± 0.0024	0.0159 ± 0.0014	0.0967 ± 0.0031	0.1038 ± 0.0021	**0.1431 ± 0.0019**
routing attention	0.0154 ± 0.0016	0.0143 ± 0.0006	0.015 ± 0.0007	0.0817 ± 0.0008	0.0938 ± 0.0027	0.1815 ± 0.0223
linformer attention	0.0144 ± 0.0018	0.0148 ± 0.0007	0.0148 ± 0.0014	0.0879 ± 0.0026	0.1054 ± 0.0108	0.231 ± 0.0451
reformer attention	0.0153 ± 0.001	**0.0132 ± 0.0**	**0.013 ± 0.0**	0.0835 ± 0.0	0.0935 ± 0.0	0.2222 ± 0.0
long-short attention	**0.0141 ± 0.0008**	0.0154 ± 0.002	0.0142 ± 0.0008	0.082 ± 0.0018	0.0915 ± 0.01	0.1888 ± 0.0198
longformer attention	0.0149 ± 0.0011	0.0147 ± 0.0014	0.0149 ± 0.001	0.0779 ± 0.0034	0.083 ± 0.0029	0.1807 ± 0.0217
transformer-xl attention	0.0148 ± 0.0006	0.0143 ± 0.0005	0.0141 ± 0.0006	0.0815 ± 0.0023	0.0911 ± 0.0075	0.2277 ± 0.0479

Table 5.13: Results of attention models evaluation using *MAE* metric on combinations of datasets and forecasting horizons specified Table 5.12. The notation stands for: *mean ± std.* The best (lowest) values are shown in **bold**.

Merged encoder only (RMSE)	**Synthetic 100**	**Synthetic 200**	**Synthetic 400**	**Libra 100**	**Libra 200**	**Libra 400**
passthrough attention	0.0177 ± 0.0011	0.0189 ± 0.0015	0.0173 ± 0.0012	0.131 ± 0.0013	0.1453 ± 0.0018	0.2058 ± 0.0083
multi-head attention	0.0184 ± 0.0016	0.0173 ± 0.0006	0.0177 ± 0.0013	0.1009 ± 0.0031	0.126 ± 0.0282	0.3004 ± 0.0824
adaptive span attention	0.0176 ± 0.0012	0.0179 ± 0.001	0.0175 ± 0.0007	0.1111 ± 0.0034	0.1292 ± 0.0114	0.2416 ± 0.0269
attention free full attention	0.0181 ± 0.0014	0.018 ± 0.0022	0.0175 ± 0.0008	0.1289 ± 0.0024	0.1427 ± 0.0023	0.2036 ± 0.0092
big bird attention	0.0187 ± 0.0016	0.0179 ± 0.0009	0.0173 ± 0.0007	0.1062 ± 0.005	0.1114 ± 0.0056	0.2478 ± 0.0445
conformer attention	0.0194 ± 0.0029	0.0186 ± 0.001	0.0193 ± 0.0017	**0.0913 ± 0.0029**	**0.0977 ± 0.0021**	0.1935 ± 0.0105
synthesizer attention	0.0182 ± 0.0013	0.0183 ± 0.0015	0.0191 ± 0.0012	0.1307 ± 0.0012	0.1502 ± 0.0039	0.2437 ± 0.0359
performer attention	0.0214 ± 0.0035	0.0184 ± 0.0008	0.0184 ± 0.0013	0.1284 ± 0.003	0.1428 ± 0.0052	0.2533 ± 0.0585
fnet attention	0.0185 ± 0.002	0.0204 ± 0.0028	0.0197 ± 0.0016	0.1214 ± 0.0033	0.1318 ± 0.0025	**0.1746 ± 0.0021**
routing attention	0.0188 ± 0.0018	0.0178 ± 0.0007	0.0186 ± 0.0008	0.1059 ± 0.001	0.1206 ± 0.0028	0.2226 ± 0.0255
linformer attention	0.0177 ± 0.0019	0.0184 ± 0.0008	0.0183 ± 0.0017	0.1128 ± 0.003	0.1334 ± 0.0115	0.2816 ± 0.0529
reformer attention	0.019 ± 0.0012	**0.0165 ± 0.0**	**0.0162 ± 0.0**	0.1082 ± 0.0	0.1208 ± 0.0	0.2788 ± 0.0
long-short attention	**0.0174 ± 0.0009**	0.019 ± 0.0023	0.0177 ± 0.0009	0.1065 ± 0.002	0.1182 ± 0.0119	0.2325 ± 0.023
longformer attention	0.0184 ± 0.0013	0.0182 ± 0.0015	0.0184 ± 0.0012	0.102 ± 0.0041	0.1082 ± 0.0028	0.2229 ± 0.0237
transformer-xl attention	0.0182 ± 0.0007	0.0178 ± 0.0006	0.0175 ± 0.0006	0.1053 ± 0.0022	0.1174 ± 0.0091	0.2744 ± 0.0503

Table 5.14: Results of attention models evaluation using *RMSE* metric on combinations of datasets and forecasting horizons specified Table 5.12. The notation stands for: *mean ± std.* The best (lowest) values are shown in **bold**.

The obtained evaluation metrics allow for ordering of the attention models in order to determine a subset of models to be utilized in the following experiments. Unfortunately, ANOVA test showed *non-significant difference between the metrics of tested attention models*. After some investigation, it was concluded that the used *merged encoder-only* architecture is likely capable of capturing some degree of element relations through the final fully-connected layer (section 4.3) which has access to all sequence elements at once. Hence, the influence of the utilized attention model becomes less prominent. To counteract this problem, it was decided to perform the experiment once more but use the more standard *encoder-only* architecture which does not have the possibility of relating elements via the last layer. The results of the repeated experiment run are presented in Table 5.13 and Table 5.14. The experiment configuration stayed the same apart from the mentioned change of the host architecture.

Encoder only (MAE)	**Synthetic 100**	**Synthetic 200**	**Synthetic 400**	**Libra 100**	**Libra 200**	**Libra 400**
passthrough attention	0.1172 ± 0.0031	0.1137 ± 0.0025	0.1211 ± 0.0034	0.1861 ± 0.0009	0.1947 ± 0.0006	0.2178 ± 0.0008
multi-head attention	0.018 ± 0.0017	0.0192 ± 0.0027	0.0198 ± 0.0018	0.0921 ± 0.0034	0.1212 ± 0.0057	0.1564 ± 0.009
adaptive span attention	0.0217 ± 0.0036	0.0205 ± 0.0016	0.0195 ± 0.0004	0.1041 ± 0.0013	0.1272 ± 0.0075	0.1674 ± 0.0081
attention free full attention	0.1168 ± 0.0026	0.1149 ± 0.0038	0.1207 ± 0.0031	0.1889 ± 0.0047	0.2004 ± 0.0034	0.2199 ± 0.0019
big bird attention	**0.0167 ± 0.001**	0.0185 ± 0.0013	0.019 ± 0.0015	0.0872 ± 0.0014	0.1144 ± 0.0041	**0.1455 ± 0.004**
conformer attention	0.0173 ± 0.001	0.0182 ± 0.0008	0.0173 ± 0.0012	**0.0757 ± 0.0025**	**0.0823 ± 0.0024**	0.1556 ± 0.0123
synthesizer attention	0.0406 ± 0.0088	0.0418 ± 0.0128	0.0458 ± 0.0118	0.1554 ± 0.0093	0.163 ± 0.0028	0.2039 ± 0.009
performer attention	0.0216 ± 0.0017	0.0216 ± 0.0018	0.0247 ± 0.0027	0.1034 ± 0.0029	0.1143 ± 0.0047	0.209 ± 0.0506
fnet attention	0.0346 ± 0.0042	0.0327 ± 0.0043	0.0356 ± 0.0023	0.1243 ± 0.0028	0.1355 ± 0.002	0.1941 ± 0.0024
routing attention	0.0217 ± 0.0017	0.0239 ± 0.0048	0.0209 ± 0.0019	0.1019 ± 0.0047	0.1374 ± 0.0066	0.1722 ± 0.0056
linformer attention	0.0174 ± 0.0019	0.0183 ± 0.0028	0.0185 ± 0.0014	0.1268 ± 0.0034	0.1411 ± 0.0068	0.187 ± 0.0086
reformer attention	0.0224 ± 0.0004	0.0232 ± 0.0	0.0204 ± 0.0	0.0944 ± 0.0	0.1145 ± 0.0	0.1661 ± 0.0
long-short attention	0.0194 ± 0.0009	**0.0178 ± 0.0013**	0.0191 ± 0.0018	0.1011 ± 0.0042	0.118 ± 0.0029	0.1834 ± 0.0074
longformer attention	0.0169 ± 0.002	0.0185 ± 0.001	**0.0172 ± 0.0015**	0.0871 ± 0.0023	0.1127 ± 0.0046	0.1514 ± 0.0083
transformer-xl attention	0.0179 ± 0.0015	0.0198 ± 0.0015	0.0197 ± 0.0019	0.104 ± 0.0041	0.1334 ± 0.0044	0.1675 ± 0.0076

Table 5.15: Results of attention models evaluation using *MAE* metric on combinations of datasets and forecasting horizons specified Table 5.12. This time, *encoder-only* was used as the host architecture. The notation stands for: *mean ± std*. The best (lowest) values are shown in **bold**.

Encoder only (RMSE)	**Synthetic 100**	**Synthetic 200**	**Synthetic 400**	**Libra 100**	**Libra 200**	**Libra 400**
passthrough attention	0.1349 ± 0.0036	0.1342 ± 0.003	0.1428 ± 0.0041	0.2119 ± 0.0008	0.224 ± 0.0005	0.252 ± 0.0007
multi-head attention	0.0218 ± 0.0019	0.0234 ± 0.0031	0.0241 ± 0.0022	0.1175 ± 0.0034	0.1514 ± 0.0062	0.1917 ± 0.0093
adaptive span attention	0.0259 ± 0.004	0.0249 ± 0.0019	0.0238 ± 0.0005	0.1297 ± 0.0014	0.1579 ± 0.0087	0.2025 ± 0.0093
attention free full attention	0.1346 ± 0.0029	0.1356 ± 0.0045	0.1426 ± 0.0039	0.2156 ± 0.0053	0.2306 ± 0.0045	0.2533 ± 0.0018
big bird attention	**0.0203 ± 0.0011**	0.0226 ± 0.0015	0.0232 ± 0.0017	0.1121 ± 0.0018	0.144 ± 0.0047	**0.1798 ± 0.0035**
conformer attention	0.0209 ± 0.001	0.0223 ± 0.001	0.0213 ± 0.0014	**0.1003 ± 0.0021**	**0.1077 ± 0.0021**	0.1885 ± 0.0122
synthesizer attention	0.0477 ± 0.0097	0.0499 ± 0.0151	0.0546 ± 0.0138	0.1863 ± 0.0103	0.1979 ± 0.0029	0.2395 ± 0.008
performer attention	0.0259 ± 0.002	0.0262 ± 0.0021	0.0299 ± 0.0031	0.1284 ± 0.003	0.1428 ± 0.0052	0.2533 ± 0.0585
fnet attention	0.0407 ± 0.0045	0.0393 ± 0.0051	0.0426 ± 0.0028	0.1514 ± 0.0029	0.1659 ± 0.0024	0.2306 ± 0.0025
routing attention	0.0261 ± 0.002	0.0287 ± 0.0055	0.0255 ± 0.0022	0.1273 ± 0.0044	0.1709 ± 0.007	0.2073 ± 0.0052
linformer attention	0.0212 ± 0.0023	0.0223 ± 0.0031	0.0227 ± 0.0016	0.1544 ± 0.0042	0.1729 ± 0.0069	0.2237 ± 0.0092
reformer attention	0.027 ± 0.0005	0.0281 ± 0.0	0.0249 ± 0.0	0.1192 ± 0.0	0.1464 ± 0.0	0.2019 ± 0.0
long-short attention	0.0235 ± 0.001	**0.0218 ± 0.0016**	0.0234 ± 0.0021	0.127 ± 0.0047	0.1496 ± 0.003	0.2198 ± 0.0079
longformer attention	0.0207 ± 0.0023	0.0226 ± 0.0012	**0.0212 ± 0.0017**	0.1124 ± 0.0028	0.1424 ± 0.005	0.1847 ± 0.0092
transformer-xl attention	0.0218 ± 0.0018	0.024 ± 0.0017	0.0241 ± 0.0022	0.13 ± 0.0043	0.1656 ± 0.0045	0.2031 ± 0.0074

Table 5.16: Results of attention models evaluation using *RMSE* metric on combinations of datasets and forecasting horizons specified Table 5.12. This time, *encoder-only* was used as the host architecture. The notation stands for: *mean ± std*. The best (lowest) values are shown in **bold**.

The second experiment run produced the desired statistically significant differences in metrics. When comparing the results of both runs, the difference quickly becomes apparent. For example, *passthrough attention* model (i.e. no attention) performs much worse in *encoder-only* than in *merged encoder-only* which implies that attention modelling plays more important role in the former architecture.

To obtain a subset of attention models for further experiments, the evaluated models were incrementally sorted based on both metrics values from the second run (*encoder-only* host architecture), dataset type and forecasting horizon. The first sorting was done based on results from *Libra 400* (dataset and horizon), the second based on *Libra 200* and the final on based on *Libra 100*. The reasoning behind it is that Libra dataset represents a reasonable proxy to real-world time series and 400 forecasting horizon constitutes the most challenging situation for modelling attention. From the list of sorted models, it was decided to take *the better half i.e.* 7 *out of* 14 *available models*. The selected models together with metric values extracted from Table 5.15 and Table 5.16 are listed and ordered in Table 5.17. In addition, a training example with forecasts illustrating behaviour of the selected models is shown in Table 5.17.

Selected attention models (MAE, RMSE)	**Libra 400**	**Libra 200**	**Libra 100**
big bird attention	0.1455, 0.1798	0.1144, 0.144	0.0872, 0.1121
longformer attention	0.1514, 0.1847	0.1127, 0.1424	0.0871, 0.1124
conformer attention	0.1556, 0.1885	0.0823, 0.1077	0.0757, 0.1003
multi-head attention	0.1564, 0.1917	0.1212, 0.1514	0.0921, 0.1175
reformer attention	0.1661, 0.2019	0.1145, 0.1464	0.0944, 0.1192
adaptive span attention	0.1674, 0.2025	0.1272, 0.1579	0.1041, 0.1297
transformer-xl attention	0.1675, 0.2031	0.1334, 0.1656	0.104, 0.13

Table 5.17: Selected attention models sorted based on metrics obtained for the listed dataset/forecasting horizon combinations. The metrics values were extracted from Table 5.16 and Table 5.15.

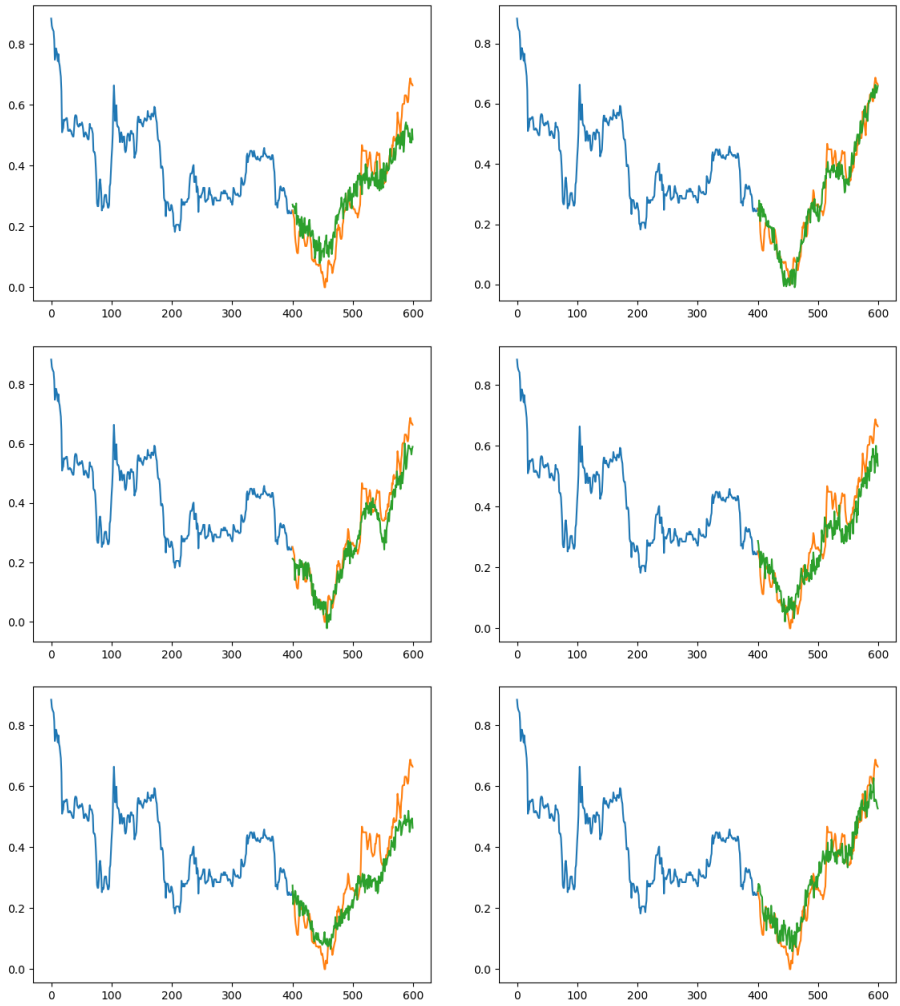

Figure 5.5: Example forecasts of one training example from Libra dataset produced by *encoder-only* architecture with different attention models. Blue color represents the model input, orange color the true values and green color the forecasted values. Top left: *Big Bird attention*, top right: *Longformer attention*, center left: *Conformer attention*, center right: *Reformer attention*, bottom left: *Adaptive span attention*, bottom right: *TransformerXL attention*.

5.3.3 Phase 3: Genotype Encoding

The third experimental phase aims to compare different variations of genotype encoding as presented in section 4.5. The goal is to select the genotype configuration which produces the most performant Transformer-based architectures optimized for time series forecasting. Another aspect to consider is the ability of each genotype variant to effectively utilize concepts and architectural patterns not commonly present in Transformers. For that reason, this phase is split into two experiments. The first experiment (section 5.3.3) focuses on the evaluation of genotype configurations while optimizing the original Transformer architecture given only its own components. The second experiment (section 5.3.3) broadens the scope and considers additional concepts, components and architectural patterns. The concepts, components and patterns come in as various modules originally introduced in section 4.7. To perform both of the experiments, the neural architecture search system described in section 4.5 was utilized.

Concept	**Configuration**
dataset for evaluation	sub-sampled Libra 1
model input size	400
model output size (forecasting horizon)	50
iterations	400
population size	150
cell initialization	2 cells per layer
smart initialization	8x *pre-norm* Transformer encoder seeded in the initial population
parent selection	best fitness tournament (size: 2)
survivor selection	best fitness tournament (size: 2)
replacement selection	worst fitness tournament (size: 4)
new hurdle after iterations	40
max epochs to train for	8
evaluation repeated	4x per encoding variant

Table 5.18: The base configuration shared by both genotype encoding evaluation experiments.

The subsection 4.5.1 introduced three variants of the described encoding based on how modules can be stacked in Genotype cell branches. To make referring to different variants easier, each of them was named based on the number of modules it allows: *1 module per branch*, *3 modules per branch* and *Unlimited modules per branch*. Additionally, section 4.6.2 claims that smart initialization based on seeding the initial population with known Transformer models has a great effect on the results produced by the search. Hence, to evaluate the influence of initialization and decouple its contribution from the genotype encoding effects, the first two variants were initialized randomly while the *Unlimited modules per branch* variant was considered in two forms: 1.) randomly initialized, 2.) randomly initialized and seeded with 8 instances of *pre-norm* Transformer encoder (section 3.4.2). To recapitulate, four alternatives based on three genotype encoding variants were considered and evaluated in this phase.

Details about the setup shared by both experiments are provided in Table 5.18. The search was performed using the first *sub-sampled Libra* dataset only. The *model input size* was set to 400 (larger size than in the first experimental phase, subsection 5.3.1) giving the candidate models more input data and enabling potentially better forecasts. The *forecasting horizon* was set to 50 which represents a challenging forecasting problem. When it comes to Progressive Dynamic Hurdles (section 4.6.2) configuration, new hurdles were introduced every 40 iterations giving the candidate models that surpass them more time to train. Each passed hurdle means one more epoch a candidate model can train for. The base amount was 1 epoch and the maximum amount was 8 epochs. The random initialization of genotype variants was configured to create layers containing exactly two cells as that is the same number of cells as in the original Transformer layer. Nevertheless, the number of cells can be altered by evolution during the search.

Parameter	**Parameter type**	**Available values**
patch size	Set	[1, 5, 10, 25, 50]
embedding dim	Set	[16, 32, 64]
layer count	Range	(1, 8), step: 1
merge before regression	Set	[true, false]
use memory	Constant	false

Table 5.19: A list of genotype parameters used in both encoding evaluation experiments. The type column specifies the used evolvable parameter type as defined in subsection 4.5.2

As presented in subsection 4.5.1, each genotype can have several evolvable hyperparameters. A list of hyperparameters used in the following experiments is provided in Table 5.19. The available parameter values were derived based on the results from Phase 1 (subsection 5.3.1). The inclusion of *merge before regression* parameter was motivated by findings from the previous phase (subsection 5.3.2) i.e. by the fact that performance of *encoder-only* Transformer variant is affected by the number of output values entering the final fully-connected layer. The parameter allows evolution to decide if to use just one output value (*encoder-only*) or a concatenation of all values (*merged encoder-only*). Unfortunately, preliminary testing showed that *use memory* parameter needs to be fixed to *false*. If set to *true*, the search would often run out of GPU memory as the feature proved to claim most of the available memory.

An important thing to emphasize is that the number of Transformer layers is specified explicitly by a hyperparameter. Even though the neural architecture search system is capable of evolving each layer independently, preliminary testing revealed that searches of this kind are more unstable and, as the complexity of the search space increases drastically, can easily take several weeks to finish. For that reason, it was decided to stick to evolving one Transformer layer only which is then replicated *layer count* times to produce the final model.

All genetic operators from section 4.6.1 were used, apart from the ones manipulating Genotype layers. These are not needed as the layers are not evolved independently in this phase. The probabilities of applying certain operators to different genotype variants are listed in Table 5.20. The intention was to keep the probabilities as similar as possible, however, some of them needed to be restricted (i.e. have their probability set to 0.0) as their application could violate the constraints imposed on the number of modules per branch.

Genetic operator probability	**1 module per branch**	**3 modules per branch**	**Unlimited modules per branch**
perform cell mutation	0.2	0.2	0.2
∗ *add empty cell*	**0.0**	**0.0**	**0.1**
∗ *duplicate cell*	0.1	0.1	0.1
∗ *swap two cells*	0.2	0.2	0.2
∗ *change cell inputs*	0.2	0.2	0.2
∗ *remove cell*	**0.1**	**0.1**	**0.2**
exchange cells crossover	0.2	0.2	0.2
exchange branches crossover	0.2	0.2	0.2
perform module mutation	0.3	0.3	0.3
∗ *add new module*	**0.0**	**0.0**	**0.1**
∗ *duplicate module*	**0.0**	**0.0**	**0.1**
∗ *swap two modules*	0.2	0.2	0.2
∗ *swap module for new*	0.2	0.2	0.2
∗ *remove module*	**0.0**	**0.0**	**0.2**
perform parameter mutation	0.3	0.3	0.3
∗ *change genotype parameter*	0.1	0.1	0.1
∗ *change module parameter*	0.2	0.2	0.2

Table 5.20: The probabilities of applying mutation and crossover operators used in both experiments. The utilized operators are described in section 4.6.1. The operators marked with ∗ form children of the nearest higher listed operator without a ∗ i.e. the higher listed operator needs to get selected first for the child operators to be eligible for selection (probabilities multiply). The values that differ for some configurations are highlighted in **bold**.

Ex1: Encoding evaluation on Transformer modules

The first experiment employs only modules encapsulating concepts from the original Transformer. In addition to these, *Single feed forward* module implementing a single fully-connected layer and *Generic passthrough* no-op module helping to create residual connections were included. All used modules are shown in Table 5.21. Also, their evolvable parameters are listed and described in Appendix (section 6.3). To replicate the positional encoding approach of the original Transformer, *Sinusoidal absolute positional encoding* is added to the model input and its module is then frozen i.e. cannot be modified by evolution. That way, only the internal *encoder-only* structure can be evolved.

Module category	**Included modules**
passthrough	Generic passthrough (empty module)
attention	Multi-head attention
feed forward	Single feed forward, Feed forward
activation	ReLU
normalization	Layer normalization
positional encoding	Sinusoidal absolute positional encoding

Table 5.21: Modules used for the first experiment.

Genotype variation	**Best genotype fitness**
1 module per branch	-0.004215 ± 0.000120
3 modules per branch	-0.00363 ± 0.000096
Unlimited modules per branch	-0.004275 ± 0.000293
Unlimited modules per branch (encoder seeded)	**-0.00346 ± 0.000044**

Table 5.22: Results of genotype encoding variants comparison. The notation stands for: *mean ± std*. The best (highest) value is shown in **bold**.

As the neural architecture search proved to be very time consuming (cca 24-72 hours per one execution), it was repeated only four times per each genotype variant. The results are listed in Table 5.22 which shows the mean fitness of the

best genotypes obtained during every run. One-way ANOVA confirmed that the differences in fitness are statistically significant, however, it was discovered that the data violated the assumption of normally distributed residuals. Nevertheless, Knief and Forstmeier [77] confirmed that ANOVA and other Gaussian methods stay remarkably robust even when the normality assumption is violated. For that reason, the results are considered valid. As can also be seen in the table, *Unlimited modules (encoder seeded)* variant scored the best, however, it performs only slightly worse than the *3 modules per branch* variant which was initialized randomly. On the other hand, one can argue that the restrictions over the placement of modules introduce relatively strong bias towards vanilla Transformer-like architectures. Hence, to assess differences in the produced architectures, the best genotypes of both *3 modules per branch* and *Unlimited modules (encoder seeded)* variants were visualized for comparison.

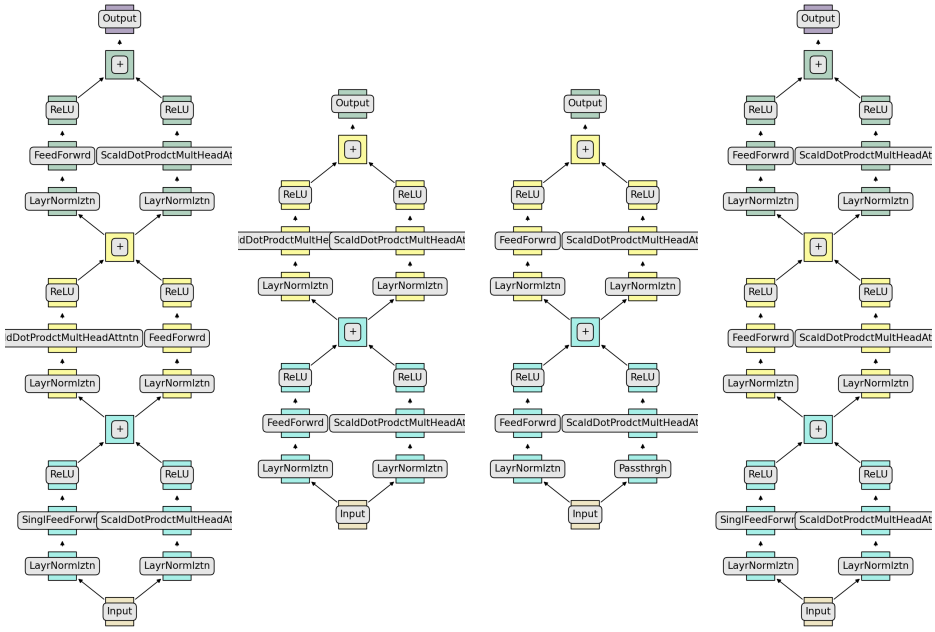

Figure 5.6: Visualisation of *evolved Genotype layers* extracted from the best genotype found during each search run which used *3 modules per branch* genotype variant.

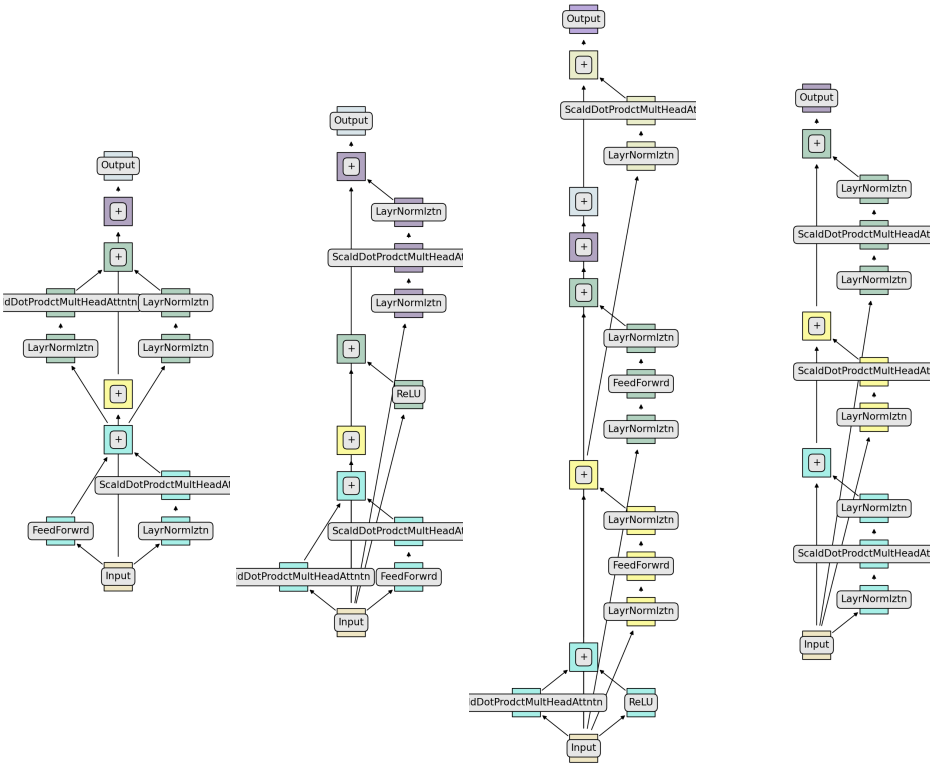

Figure 5.7: Visualisation of *evolved Genotype layers* extracted from the best genotype found during each search run which used *Unlimited modules per branch (encoder seeded)* genotype variant.

Figure 5.6 shows Genotype layers extracted from the best *3 modules per branch* genotypes. The first thing to notice, is that the evolved layers are very similar. All four layers have ReLU activation placed as the last module in all cell branches, apart from one case, always use layer normalization as the first module in all cell branches and very often combine feed forward type of a module in parallel with the multi-head attention module. Also, the number of cells in each layer is either 2 or 3. This seems to confirm the theory suggesting that there is some amount of bias towards vanilla Transformer-like architectures. In contrast, the Genotype layers of *Unlimited modules (encoder seeded)* variant are more diverse. A prominent difference is the fact that all the shown layers include unbroken residual path which connects their inputs to their outputs. Similar to

the previous variant, cell branches often start with layer normalization, however, there are also several occurrences of layer normalization being the last module in a cell branch. This configuration is not possible in *3 modules per branch* variant. Additionally, a "sandwich" configuration consisting of two layer normalization modules and either feed forward or multi-head attention module in between is also a common pattern.

To sum up, both of these well-performing variants evolved slightly different patterns. *Unlimited modules (encoder seeded)* variant showed more flexibility and achieved slightly better results. However, a very limited set of modules was used in this experiment. Hence, to further assess the qualities of all 4 variants, a second experiment considering substantially larger list of modules was performed. The experiment is described in the following section (section 5.3.3). Additionally, Figure 5.8 provides a deeper insight into how all searches conducted during this experiment looked like. The figure shows three plots per each genotype variant illustrating how *survivor fitness*, *progressive hurdles* and *difference factor* metric developed during each search.

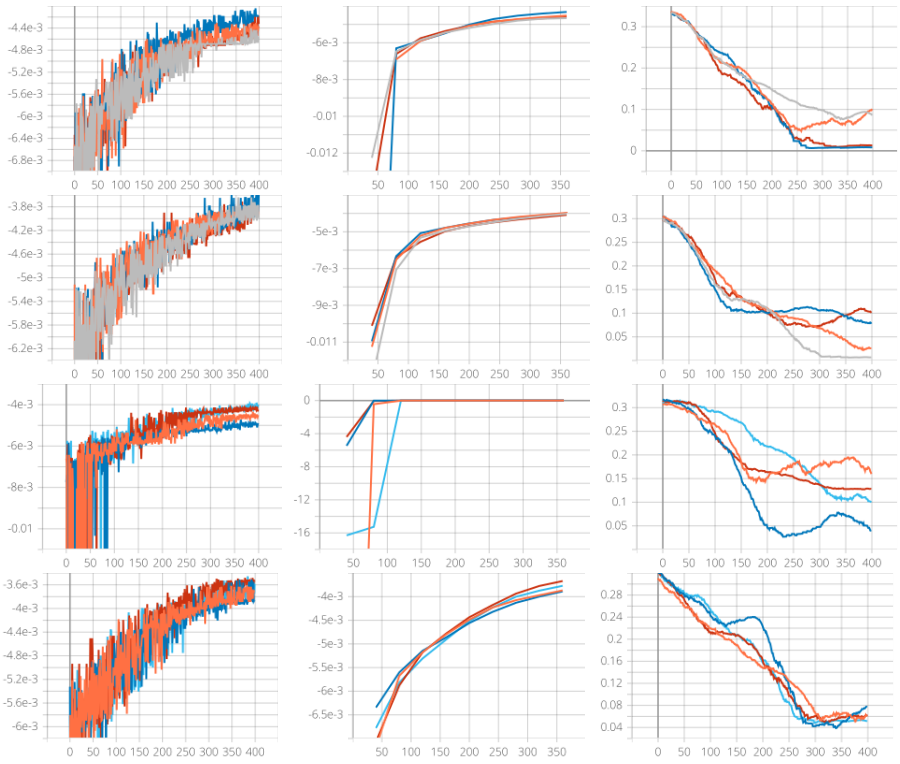

Figure 5.8: Plots showing the development of control metrics during all evolutionary searches performed in the first experiment of Phase 3. The columns represent *survivor fitness*, the introduced *progressive hurdles* and *difference factor*. Each row belongs to one evaluated genotype variant ordered as follows: *1 module per branch*, *3 modules per branch*, *Unlimited modules (randomly initialized)*, *Unlimited modules (seeded with Transformer encoder)*.

Ex2: Encoding evaluation on multiple module sets

In the second experiment, the number of available modules was increased significantly. This experiment focuses on the evaluation of genotype encoding flexibility and the ability to evolve architectures beyond what is known or used in Transformer-based architectures. The available modules are listed in Table 5.23. A detailed list of their evolvable parameters is also provided in Appendix (section 6.3). The set of included attention modules is based on the results of Phase 2 (subsection 5.3.2). This time, positional encoding is not added explicitly and it is up to the evolutionary search to decide if and where to place it.

Module category	**Included modules**
passthrough	Generic passthrough (empty module)
attention	Big Bird attention, Longformer attention, Conformer attention, Multi-head attention, Reformer attention, Adaptive span attention, TransformerXL attention
feed forward	Single feed forward, Feed forward
activation	ELU, GeLU, ReLU, SiLU, Leaky ReLU, Squared ReLU
normalization	Layer normalization, AdaNorm, PowerNorm, Scaled L2 normalization, Root mean square layer normalization
positional encoding	Sinusoidal absolute positional encoding, Axial positional encoding, BERT learned positional encoding, Simple absolute positional encoding
convolution	Convolution 1D, Convolution 2D, Max pooling 1D, Max pooling 2D, Depth-wise separable convolution, Inverted bottleneck, CoordConv
dropout	Alpha Dropout, Dropout, Dropout 2D

Table 5.23: Modules used for the second experiment of Phase 3.

As in the first experiment of this phase, the search was repeated 4 times per genotype variant. With more modules available, the duration of each search increased noticeably (up to cca 48-96 hours per one execution). The search results are shown in Table 5.24. In this case, the differences between the individual

variants are much smaller than in Experiment 1 (section 5.3.3). Also, ANOVA confirmed that there are *no significant differences between all tested variants*. Hence, even though there are some trends observed in the first experiment, it is *not possible to confidently conclude which genotype variant performs better*. Therefore, it was decided to investigate the experiment results further but employ a more qualitative approach i.e. visually compare the genotypes which obtained the highest fitness per each variant.

Genotype variation	**Best genotype fitness**
1 module per branch	**-0.002105 ± 0.000837**
3 modules per branch	-0.002892 ± 0.000289
Unlimited modules per branch	-0.002587 ± 0.000644
Unlimited modules per branch (encoder seeded)	-0.002862 ± 0.000129

Table 5.24: Results of genotype encoding variants comparison. The notation stands for: *mean ± std*. The best (highest) value is shown in **bold**.

Figure 5.9 displays Genotype layers extracted from the highest fitness genotypes per each genotype variant. They are listed in the same order as the genotype variants in Table 5.24 i.e. from left to right: *1 module per branch*, *3 modules per branch*, *Unlimited modules per branch*, *Unlimited modules per branch (encoder seeded)*. The first thing to notice is that the first and third Genotype layers are very similar. Ignoring that the first genotype can host just one module per branch, the only remaining difference is the placement of GeLU activation (after Inverted bottleneck) in contrast to the placement of Layer normalization (before Inverted bottleneck). Apart from these differences, the Genotype layers are identical and also quite simple compared to the rest. The second Genotype layer is very similar to the layers that the same genotype variant evolved when it had only the vanilla Transformer modules available. This confirms some degree of bias towards Transformer-like architectures as also seen in the first experiment.

A similar pattern can be seen in genotypes produced by the last variant which had the initial population seeded with *pre-norm encoder-only*. The corresponding evolved Genotype layer uses only the modules which were available in the first experiment even though there was more modules available during this experiment. Similarly, this seems to be due to the bias introduced by seeding the initial population. Most likely, the search gets trapped in a local optimum which

resembles the original Transformer encoder. In addition, the structure of the evolved Genotype layer is also similar to structures of layers discovered during the first experiment while using the same genotype variant.

As also provided in Experiment 1, Figure 5.10 shows the evolution of control metrics during all searches performed in this experiment. The figure shows three plots per each genotype variant illustrating how *survivor fitness*, *progressive hurdles* and *difference factor* metric developed during each search.

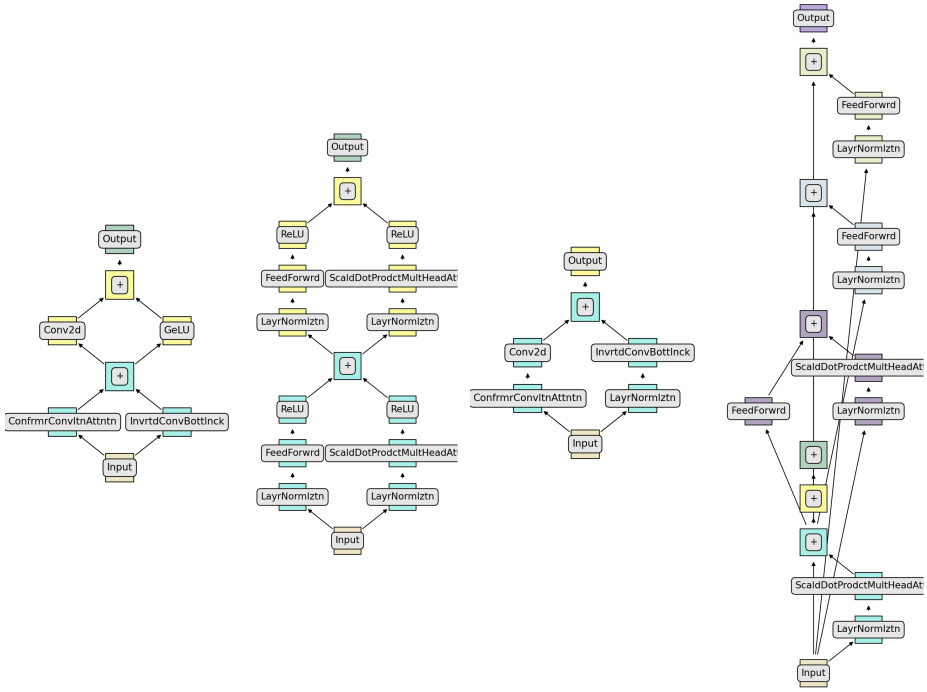

Figure 5.9: Visualisation of *evolved Genotype layers* extracted from the best genotype found for each genotype variant. Left: *1 module per branch*, Middle left: *3 modules per branch*, Middle right: *Unlimited modules per branch*, Right: *Unlimited modules per branch (encoder seeded)*.

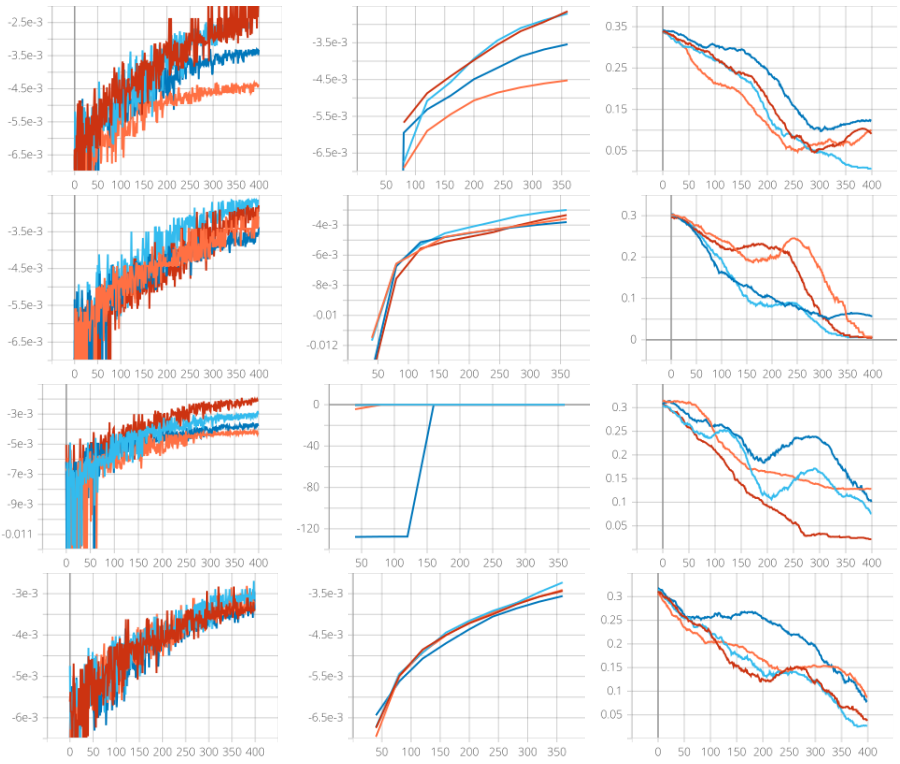

Figure 5.10: Plots showing the development of control metrics during all evolutionary searches performed in the second experiment of Phase 3. The columns represent *survivor fitness*, the introduced *progressive hurdles* and *difference factor*. Each row belongs to one evaluated genotype variant ordered as follows: *1 module per branch*, *3 modules per branch*, *Unlimited modules (randomly initialized)*, *Unlimited modules (seeded with Transformer encoder)*.

5.3.4 Phase 4: Forecasting Methods Comparison

The last experimental phase utilizes the best architectures evolved during the previous phase (subsection 5.3.3) and compares their forecasting performance with state-of-the-art forecasting methods presented in section 4.8. The *top 3 highest fitness architectures from the third experimental phase* are included to assess their forecasting capabilities and determine if there are significant differences in their performance. For convenience, all compared methods and architectures are listed in Table 5.25 together with their hyperparameters. The hyperparameters of the evolved architectures were also optimized by evolution during Phase 3.

Forecasting method	**Configuration**
ARIMA	p = 3, q = 3, d = 2
Recurrent Neural Network	hidden dimension = 64, number of layers = 2, dropout = 0.1
LSTM	hidden dimension = 64, number of layers = 2, dropout = 0.1
Temporal Convolutional Network	hidden dimension = 25, number of layers = 8, kernel size = 17, dropout = 0.1
N-BEATS	hidden dimension = 128, stack types = [Generic stack, Generic stack], theta dimensions = [8, 8], blocks per stack = 6
Pre-norm encoder-only variant	embedding dimension = 32, patch size = 5, head count = 4, layer count = 7, forward expansion = 6, dropout = 0.1
The evolved architecture #1	embedding dimension = 64, patch size = 1, layer count = 6, merge before regression = true, use memory = false
The evolved architecture #2	embedding dimension = 64, patch size = 5, layer count = 6, merge before regression = true, use memory = false
The evolved architecture #3	embedding dimension = 64, patch size = 5, layer count = 6, merge before regression = true, use memory = false

Table 5.25: A list of the compared forecasting methods and evolved architectures including the used hyperparameters.

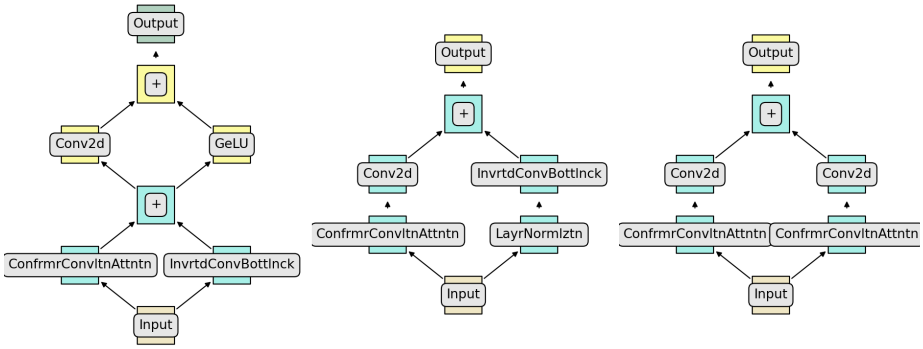

Figure 5.11: Visualisation of the *Genotype layers* stacked inside the three evolved architectures used in this experimental phase. Left: The evolved architecture #1, Center: The evolved architecture #2, Right: The evolved architecture #3.

Apart from the hyperparameters, the structures of the evolved Genotype layers stacked inside the evolved architectures also impact performance. The Genotype layers belonging to the three considered architectures are shown in Figure 5.11. Surprisingly, all layers rely heavily on *convolution* and quite little on *attention*. The only attention module they all utilized is *Conformer attention* which is built as a series of *point-wise* and *depth-wise convolutions* interleaved with normalizations and activations. To provide a full picture, the internal structure of *Conformer module* is shown in Figure 5.12. Additionally, the layers of all architectures are similar in structure and utilize almost the same set of modules which primarily consists of *Convolution 2D* and *Inverted bottleneck* modules. Also, most of the modules inside Genotype layers have parameters attached. The values of all parameters are presented in Table 5.26.

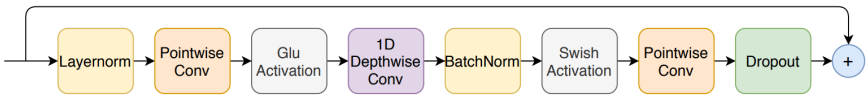

Figure 5.12: The internal structure of *Conformer attention* module [59].

Architecture	Module	Parameter	Branch	Value
#1	*Conformer attention*	dropout	left	0.1
#1	*Inverted bottleneck*	kernel size	right	12
#1	*Inverted bottleneck*	dilation	right	3
#1	*Inverted bottleneck*	forward expansion	right	2
#1	*Convolution 2D*	kernel width	left	3
#1	*Convolution 2D*	kernel height	left	6
#1	*Convolution 2D*	horizontal dilation	left	1
#1	*Convolution 2D*	vertical dilation	left	1
#2	*Conformer attention*	dropout	left	0.0
#2	*Convolution 2D*	kernel width	left	12
#2	*Convolution 2D*	kernel height	left	6
#2	*Convolution 2D*	horizontal dilation	left	1
#2	*Convolution 2D*	vertical dilation	left	2
#2	*Inverted bottleneck*	kernel size	right	6
#2	*Inverted bottleneck*	dilation	right	1
#2	*Inverted bottleneck*	forward expansion	right	4
#3	*Conformer attention*	dropout	left	0.0
#3	*Convolution 2D*	kernel width	left	12
#3	*Convolution 2D*	kernel height	left	6
#3	*Convolution 2D*	horizontal dilation	left	1
#3	*Convolution 2D*	vertical dilation	left	2
#3	*Conformer attention*	dropout	right	0.0
#3	*Convolution 2D*	kernel width	right	12
#3	*Convolution 2D*	kernel height	right	6
#3	*Convolution 2D*	horizontal dilation	right	1
#3	*Convolution 2D*	vertical dilation	right	2

Table 5.26: The values of parameters attached to the modules used by the evolved architectures utilized in this experimental phase.

The one and only experiment conducted in this phase uses a setup similar to the setups used by experiments in Phase 1 and 2 (subsection 5.3.1, subsection 5.3.2). Details are provided in Table 5.27. The first important difference is the use of *the second sub-sampled Libra dataset* to eliminate the advantage the evolved architectures would have if they were evaluated on the same dataset as they were optimized on during evolution. Another detail to mention is the use of *the same pregenerated training examples for training of all compared methods and architectures*. This aims to eliminate the influence of different training example generation methods as they might affect the achieved forecasting accuracies. The only exception to this rule is ARIMA. To keep the ARIMA method effective, it needs to be fit to only one time series at once. For that reason, ARIMA is hard to train on multiple series as it is done for the other compared methods. Hence, ARIMA was considered to be a baseline and was given a slight advantage. The method was trained and evaluated on each time series in the dataset separately and the mean value of the obtained accuracies was used as the final forecasting performance measure. Finally, it was decided to evaluate all methods on each Libra category separately in order to gain more insight into how the methods and architectures perform in different domains.

Concept	**Configuration**
dataset for evaluation	sub-sampled Libra 2
evaluation metrics	[MAE, RMSE]
model input size	400
model output sizes (forecasting horizons)	[1, 20, 50]
trained for epochs	20
evaluation repeated	10x per method, Libra category and forecasting horizon combination

Table 5.27: Setup and configuration of the forecasting methods and architectures comparison experiment.

Ex1: Forecasting Methods Comparison

The results of this experiment are provided in the four tables below (Table 5.28, Table 5.29, Table 5.30, Table 5.31). The tables compare metrics that each of the considered forecasting methods obtained on one category of the second sub-

sampled Libra dataset. The cells of each table display the means of MAE and RMSE metrics and the corresponding standard deviations. As usual, ANOVA was used to ensure that the differences between means are statistically significant.

To start with a more general assessment, all Transformer-based architectures (including the *pre-norm encoder-only*) demonstrated a competitive performance in time series forecasting. In addition, *the evolved architectures turned out to be mostly performance-wise superior to the other state-of-the-art forecasting methods*. In fact, in all four categories, at least one of them was always the best performing option for the 20 and 50 forecasting horizon case. For the forecasting horizon of size 1, the differences were less pronounced. In the *Economics* category, all Transformer-based architectures were surpassed by *N-BEATS*. In general, *N-BEATS* was the best performing method if not considering Transformer-based architectures. In the *Finance* category, all methods were slightly outperformed by *ARIMA*. However, given the training advantage it had, it is a negligible difference. The results achieved by *TCN* were usually among the worst. This is a surprising finding as *TCN* is conceptually close to the evolved architectures which also rely on convolution.

When it comes to the evolved architectures only, the number #2 performed the worst. It usually managed to perform slightly better or at least on par with the vanilla *pre-norm encoder-only*, however, it never surpassed the other evolved architectures. Based on the metrics, the architecture number #1 performs better with longer forecasting horizons. In 3 out of the 4 Libra categories, it reached the lowest forecast error on the 20 and 50 forecasting horizon cases. In contrast, the number #3 architecture was stronger in the forecasting horizon 1 case and dominated in 2 out of 4 categories. To sum up, the evolved Transformer-based architectures number #1 and #3 seem to be good candidates for real-world time series forecasting.

Libra - Economics	**Forecasting horizon: 1**	**Forecasting horizon: 20**	**Forecasting horizon: 50**
ARIMA	0.2067 ± 0.0, 0.2067 ± 0.0	0.7841 ± 0.0, 0.9003 ± 0.0	2.0252 ± 0.0, 2.3889 ± 0.0
RNN	0.0552 ± 0.0008, 0.0552 ± 0.0008	0.0666 ± 0.001, 0.0879 ± 0.001	0.0692 ± 0.0005, 0.0971 ± 0.0005
LSTM	0.0499 ± 0.0032, 0.0499 ± 0.0032	0.065 ± 0.0022, 0.0873 ± 0.0018	0.0678 ± 0.0008, 0.0992 ± 0.0009
TCN	0.0834 ± 0.0027, 0.0834 ± 0.0027	0.1317 ± 0.0025, 0.1702 ± 0.0024	0.1349 ± 0.0008, 0.175 ± 0.0004
N-BEATS	**0.0437 ± 0.003, 0.0437 ± 0.003**	0.0549 ± 0.0009, 0.07 ± 0.001	0.064 ± 0.0014, 0.0853 ± 0.0029
Pre-norm encoder-only	0.0812 ± 0.0292, 0.0812 ± 0.0292	0.0628 ± 0.0019, 0.0834 ± 0.0025	0.0642 ± 0.0018, 0.0873 ± 0.0031
The evolved architecture #1	0.065 ± 0.0189, 0.065 ± 0.0189	**0.0342 ± 0.0079, 0.0419 ± 0.0086**	**0.0463 ± 0.017, 0.0583 ± 0.022**
The evolved architecture #2	0.0452 ± 0.0056, 0.0452 ± 0.0056	0.0549 ± 0.0091, 0.0704 ± 0.0145	0.179 ± 0.1214, 0.2119 ± 0.1135
The evolved architecture #3	0.0481 ± 0.0063, 0.0481 ± 0.0063	0.052 ± 0.0018, 0.0653 ± 0.0021	0.0579 ± 0.0105, 0.0772 ± 0.0169

Table 5.28: Results of forecasting methods and architectures comparison on *Libra Economics* dataset. Every cell shows the values of MAE and RMSE metrics separated by a comma. The notation stands for: *mean ± std*. The best (lowest) values are shown in **bold**.

Libra - Finance	**Forecasting horizon: 1**	**Forecasting horizon: 20**	**Forecasting horizon: 50**
ARIMA	**0.0091 ± 0.0, 0.0091 ± 0.0**	0.0741 ± 0.0, 0.0882 ± 0.0	0.2549 ± 0.0, 0.3199 ± 0.0
RNN	0.0221 ± 0.0076, 0.0221 ± 0.0076	0.0274 ± 0.0014, 0.031 ± 0.0013	0.0342 ± 0.001, 0.0394 ± 0.0009
LSTM	0.0115 ± 0.0005, 0.0115 ± 0.0005	0.0295 ± 0.0017, 0.0329 ± 0.0017	0.0297 ± 0.0008, 0.0345 ± 0.0009
TCN	0.0162 ± 0.0013, 0.0162 ± 0.0013	0.0255 ± 0.0016, 0.0287 ± 0.0016	0.0309 ± 0.0015, 0.0355 ± 0.0015
N-BEATS	0.0105 ± 0.0033, 0.0105 ± 0.0033	0.018 ± 0.0028, 0.0211 ± 0.0031	0.0181 ± 0.0029, 0.0221 ± 0.0035
Pre-norm encoder-only	0.0093 ± 0.0006, 0.0093 ± 0.0006	0.0222 ± 0.0012, 0.0254 ± 0.0013	0.0227 ± 0.0024, 0.0269 ± 0.0027
The evolved architecture #1	0.0096 ± 0.0012, 0.0096 ± 0.0012	0.0155 ± 0.0022, 0.018 ± 0.0022	0.0152 ± 0.0028, 0.0181 ± 0.0032
The evolved architecture #2	0.0194 ± 0.0167, 0.0194 ± 0.0167	0.0154 ± 0.0048, 0.018 ± 0.0052	0.014 ± 0.004, 0.0169 ± 0.0046
The evolved architecture #3	0.0107 ± 0.0016, 0.0107 ± 0.0016	**0.0118 ± 0.0009, 0.0142 ± 0.0011**	**0.012 ± 0.0007, 0.0147 ± 0.0008**

Table 5.29: Results of forecasting methods and architectures comparison on *Libra Finance* dataset. Every cell shows the values of MAE and RMSE metrics separated by a comma. The notation stands for: *mean ± std*. The best (lowest) values are shown in **bold**.

Libra - Human access	**Forecasting horizon: 1**	**Forecasting horizon: 20**	**Forecasting horizon: 50**
ARIMA	0.0519 ± 0.0, 0.0519 ± 0.0	0.5146 ± 0.0, 0.6166 ± 0.0	1.4105 ± 0.0, 1.6809 ± 0.0
RNN	0.0424 ± 0.0025, 0.0424 ± 0.0025	0.0511 ± 0.0025, 0.0645 ± 0.0024	0.0526 ± 0.0015, 0.0699 ± 0.0015
LSTM	0.0379 ± 0.0007, 0.0379 ± 0.0007	0.0418 ± 0.0007, 0.0554 ± 0.0007	0.0463 ± 0.0007, 0.0629 ± 0.0007
TCN	0.0416 ± 0.0027, 0.0416 ± 0.0027	0.0484 ± 0.0007, 0.0626 ± 0.001	0.0522 ± 0.0007, 0.0702 ± 0.0008
N-BEATS	0.0336 ± 0.0011, 0.0336 ± 0.0011	0.0416 ± 0.0012, 0.0551 ± 0.0014	0.0445 ± 0.0008, 0.0607 ± 0.0008
Pre-norm encoder-only	0.0356 ± 0.0006, 0.0356 ± 0.0006	0.038 ± 0.0008, 0.0502 ± 0.0009	0.0386 ± 0.0005, 0.0531 ± 0.001
The evolved architecture #1	0.0366 ± 0.0037, 0.0366 ± 0.0037	**0.0183 ± 0.0004, 0.0226 ± 0.0005**	**0.0127 ± 0.0008, 0.0157 ± 0.001**
The evolved architecture #2	0.0358 ± 0.0033, 0.0358 ± 0.0033	0.0304 ± 0.0008, 0.039 ± 0.0011	0.0232 ± 0.0012, 0.0301 ± 0.0013
The evolved architecture #3	**0.0332 ± 0.0026, 0.0332 ± 0.0026**	0.0295 ± 0.0016, 0.0379 ± 0.0019	0.0257 ± 0.0006, 0.0338 ± 0.0007

Table 5.30: Results of forecasting methods and architectures comparison on *Libra Human access* dataset. Every cell shows the values of MAE and RMSE metrics separated by a comma. The notation stands for: *mean ± std*. The best (lowest) values are shown in **bold**.

Libra - Nature and demographics	**Forecasting horizon: 1**	**Forecasting horizon: 20**	**Forecasting horizon: 50**
ARIMA	0.1350 ± 0.0, 0.1350 ± 0.0	0.8663 ± 0.0, 1.0184 ± 0.0	2.1679 ± 0.0, 2.5999 ± 0.0
RNN	0.0611 ± 0.0053, 0.0611 ± 0.0053	0.063 ± 0.0005, 0.0833 ± 0.0005	0.0678 ± 0.0013, 0.0916 ± 0.0012
LSTM	0.049 ± 0.0015, 0.049 ± 0.0015	0.0573 ± 0.0005, 0.0763 ± 0.0004	0.06 ± 0.0008, 0.0839 ± 0.0008
TCN	0.0764 ± 0.0018, 0.0764 ± 0.0018	0.0952 ± 0.0065, 0.1254 ± 0.008	0.0975 ± 0.0083, 0.1311 ± 0.0089
N-BEATS	0.0493 ± 0.0019, 0.0493 ± 0.0019	0.0564 ± 0.0013, 0.0731 ± 0.0014	0.0607 ± 0.0022, 0.0811 ± 0.0026
Pre-norm encoder-only	0.0513 ± 0.0026, 0.0513 ± 0.0026	0.0475 ± 0.0008, 0.0619 ± 0.001	0.0488 ± 0.0008, 0.0642 ± 0.0011
The evolved architecture #1	0.0484 ± 0.003, 0.0484 ± 0.003	**0.0191 ± 0.0016, 0.0239 ± 0.002**	**0.0183 ± 0.0017, 0.0241 ± 0.0019**
The evolved architecture #2	0.0446 ± 0.0031, 0.0446 ± 0.0031	0.038 ± 0.0022, 0.0481 ± 0.0026	0.0293 ± 0.0012, 0.0383 ± 0.0014
The evolved architecture #3	**0.0411 ± 0.0014, 0.0411 ± 0.0014**	0.0387 ± 0.0005, 0.0491 ± 0.0003	0.0341 ± 0.0004, 0.0442 ± 0.0006

Table 5.31: Results of forecasting methods and architectures comparison on *Libra Nature and demographics* dataset. Every cell shows the values of MAE and RMSE metrics separated by a comma. The notation stands for: *mean ± std.* The best (lowest) values are shown in **bold**.

Chapter 6

Conclusion and Future Work

In this chapter, the designed artifacts (chapter 4) and the obtained experimental results (chapter 5) are discussed in the context of the hypothesis, research goal and research questions of this work. Then, an assessment of limitations of the presented solutions is provided. Finally, the opportunities for future work are outlined. The content of this chapter maps to the *Conclusion* phase of *Design Science Research*.

6.1 Discussion

This section discusses and reflects on the experimental results, designed artifacts and other knowledge obtained in this work. The section is split into subsections which map to the research questions, the main goal and the hypothesis of the work. Each of them of is addressed separately.

Research question 1 *What combination of Transformer encoder and decoder stages achieves better forecasting accuracy?*

The first research question was answered in the first experimental phase which compared 5 different Transformer variants and managed to reach a statistically significant conclusion. After obtaining the optimal hyperparamters for each architectural variant, *merged encoder-only* achieved the best forecasting accuracy on all tested datasets and for all considered forecasting horizons. Therefore, *encoder-based architectures*, based on the result of the conducted experiments, *constitute the best option for time series forecasting.*

Research question 2 *Which attention models capture long-term dependencies in time series the best?*

The second research question was addressed by the second experimental phase. The phase compared a set of attention models discovered during the literature review while focusing on their ability to deal with long-term dependencies in time series. The models were tested in the *merged encoder-only* variant where they replaced the original attention model forming several architectural instances differing in the used attention model. The experiments comparing these instances on a combination of difficult datasets and long forecasting horizons produced a statistically significant ordering of the attention models which represented their suitability for time series forecasting and the ability to capture long-term dependencies. The later experimental phases used this knowledge to exclude models that did not perform well. To sum up, based on these experimental results and the obtained ordering, it is possible to form a subset of more suitable attention models. The size of the set can be specified as needed.

Research question 3 *What genotype encoding is suitable for representing Transformer-based architectures?*

The third research question was addressed by the third experimental phase. The base of the genotype encoding used in this work was designed as a part of the neural architecture search system while having certain aspects that can be tweaked. The third experimental phase consisted of two experiments that attempted to determine the optimal configuration of the genotype encoding. The first experiment revealed promising results of certain genotype encoding configurations, however, the more general comparison case from the second experiment showed that it cannot be concluded which configuration works the best. Nevertheless, all genotype configurations were later used to evolve Transformer-based architectures and proved to be capable of producing strong architectures for time series forecasting. For that reason, it is possible to conclude that the designed encoding is at least one of the possible encodings suitable for representing Transformer-based architectures.

Research question 4 *How does the accuracy of the evolved architectures compare to other time series forecasting methods?*

The comparison experiment performed during the fourth experimental phase demonstrated that the best evolved architectures are capable of performing on par or even surpass the accuracy of the commonly used state-of-the-art forecasting methods on univariate time series without timestamps.

Goal *Investigate evolution of Transformer-based architectures for time series forecasting.*

Based on the fact that during this work, a neural architecture search system for evolving Transformer-based architectures was designed, implemented and successfully used to evolve architectures which managed to outperform state-of-the-art forecasting methods, it is concluded that the investigation was thorough enough to produce the planned outcomes. On the other hand, there are still limits and future work opportunities as outlined in the following sections.

Hypothesis *Neural evolution is capable of designing Transformer-based neural network architectures achieving state-of-the-art performance in time series forecasting.*

The designed artifacts and conducted experiments showed that neural evolution truly is capable of producing state-of-the-art forecasting architectures based on Transformer concepts. Hence, the hypothesis is considered to be confirmed.

6.2 Limitations

The most constraining limitation of the proposed neural architecture search system is probably the amount of time needed to run a single evolutionary search. The system spends most of its run time training and evaluating candidate models, therefore, a more efficient way of estimating the performance of candidate models i.e. a faster *performance estimation strategy* would be beneficial. If that was the case, more optimal architectures can potentially be discovered as it would be possible to run the search for more iterations, with larger populations and/or including more modules in it. Also, it might then become feasible to run a search utilizing the independent evolution of individual Transformer layers which has a potential to further improve the produced architectures.

Nextly, *evolutionary algorithms* were chosen as the *search strategy* to employ, however, the other presented strategies can constitute a better fit for designing Transformer-based architectures. Also, the same goes for the Transformer architecture variant selected as a base for genotype encoding design. It is probably worth exploring how well would the other variants perform when utilized by the neural architecture search system. The last point brings up another limitation which is the limited support for representing Transformer variants in the designed encoding. In this work, the encoding was used for representing just the *encoder-only* and *merged encoder-only* variants.

Finally, the NAS system currently supports only *univariate time series without timestamps*. Extending the support to multivariate time series would require just a minor modification of the system and can possibly expand its application area as many time series datasets contain multivariate time series and there is a demand for forecasting these kinds of time series. Additionally, there is the lack of support for timestamped time series which is another potential improvement worth considering. Timestamps can be very useful when forecasting time series containing a lot of seasonality or other kinds of periodic patterns.

6.3 Future Work

In fact, all the limitations presented in the previous section (section 6.2) can be seen as opportunities for future work. The alternative *performance estimation strategies* presented subsection 3.2.3 and subsection 3.3.3 constitute some of the possible strategies that can be implemented and evaluated in the designed neural architecture search system. A particularly interesting approach presented in section 3.2.3 is the nearly instant *performance estimation without training* method by Mellor et al. [109]. Successful application of this technique would result in a massive speed up of the evolutionary searches. When it comes to the alternative Transformer variants, the designed genotype can likely be adapted to represent any of them. The modified genotype would then be evaluated for its capability to evolve strong Transformer-based architectures for time series forecasting.

Apart from addressing the actual limitations, new research directions can be investigated. A tempting research goal might be focusing on evolving architectures for resource constrained environments or simply taking the efficiency of the evolved architectures into account. The works of Tsai et al. [157], So et al. [144] and Wang et al. [161] have already laid some foundation for automated design of efficient Transformers. Finally, the masked pretraining approach presented in subsection 3.4.6 and applied to Transformers by Zerveas et al. [184] can likely be used as a performance estimation strategy on its own or at least complement the trainining of candidate architectures to obtain more accurate evaluation. Also, the same technique can be employed to pretrain the final evolved architectures attempting to push their forecasting performance even further.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

[2] J. Ainslie, S. Ontanon, C. Alberti, V. Cvicek, Z. Fisher, P. Pham, A. Ravula, S. Sanghai, Q. Wang, and L. Yang, "Etc: Encoding long and structured inputs in transformers," 2020.

[3] J. Alammar, Jun 2018. [Online]. Available: https://jalammar.github.io/illustrated-transformer/

[4] G. Athanasopoulos, R. J. Hyndman, H. Song, and D. C. Wu, "The tourism forecasting competition," *International Journal of Forecasting*, vol. 27, no. 3, pp. 822–844, 2011, special Section 1: Forecasting with Artificial Neural Networks and Computational Intelligence Special Section 2: Tourism Forecasting. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016920701000107X

[5] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2015.

[6] S. Bai, J. Z. Kolter, and V. Koltun, "An empirical evaluation of generic convolutional and recurrent networks for sequence modeling," *ArXiv*, vol. abs/1803.01271, 2018.

[7] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," 2017.

[8] B. Baker, O. Gupta, R. Raskar, and N. Naik, "Accelerating neural architecture search using performance prediction," *arXiv: Learning*, 2018.

[9] A. Bauer, M. Züfle, S. Eismann, J. Grohmann, N. Herbst, and S. Kounev, "Libra: A benchmark for time series forecasting methods," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 189–200. [Online]. Available: https://doi.org/10.1145/3427921.3450241

[10] I. Bello, "Lambdanetworks: Modeling long-range interactions without attention," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=xTJEN-ggl1b

[11] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," 2020.

[12] G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le, "Understanding and simplifying one shot architecture search," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 550–559. [Online]. Available: https://proceedings.mlr.press/v80/bender18a.html

[13] S. Borgeaud, A. Mensch, J. Hoffmann, T. Cai, E. Rutherford, K. Millican, G. van den Driessche, J.-B. Lespiau, B. Damoc, A. Clark, D. de Las Casas, A. Guy, J. Menick, R. Ring, T. W. Hennigan, S. Huang, L. Maggiore, C. Jones, A. Cassirer, A. Brock, M. Paganini, G. Irving, O. Vinyals, S. Osindero, K. Simonyan, J. W. Rae, E. Elsen, and L. Sifre, "Improving language models by retrieving from trillions of tokens," *ArXiv*, vol. abs/2112.04426, 2021.

[14] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, "Smash: One-shot model architecture search through hypernetworks," 2017.

[15] P. Brockwell and R. Davis, *Introduction to Time Series and Forecasting*, ser. Springer Texts in Statistics. Springer International Publishing, 2016. [Online]. Available: https://books.google.no/books?id=P3fhDAAAQBAJ

[16] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[17] H. Cai, T. Chen, W. Zhang, Y. Yu, and J. Wang, "Efficient architecture search by network transformation," 2017.

[18] H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu, "Path-level network transformation for efficient architecture search," 2018.

[19] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," 2019.

[20] L. Cai, K. Janowicz, G. Mai, B. Yan, and R. Zhu, "Traffic transformer: Capturing the continuity and periodicity of time series for traffic forecasting," *Transactions in GIS*, vol. 24, no. 3, pp. 736–755, 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/tgis.12644

[21] T. A. Chang, Y. Xu, W. Xu, and Z. Tu, "Convolutions and self-attention: Re-interpreting relative positions in pre-trained language models," 2021.

[22] C. Chatfield, *The Analysis of Time Series: An Introduction with R*, ser. Texts in statistical science. Chapman & Hall/CRC, 2019. [Online]. Available: https://books.google.no/books?id=-i2owwEACAAJ

[23] B. Chen, P. Li, C. Li, B. Li, L. Bai, C. Lin, M. Sun, J. Yan, and W. Ouyang, "Glit: Neural architecture search for global and local image transformer," *CoRR*, vol. abs/2107.02960, 2021.

[24] M. Chen, H. Peng, J. Fu, and H. Ling, "Autoformer: Searching transformers for visual recognition," 07 2021.

[25] M. Chen, K. Wu, B. Ni, H. Peng, B. Liu, J. Fu, H. Chao, and H. Ling, "Searching the search space of vision transformer," 11 2021.

[26] T. Chen, I. Goodfellow, and J. Shlens, "Net2net: Accelerating learning via knowledge transfer," 2016.

[27] R. Child, S. Gray, A. Radford, and I. Sutskever, "Generating long sequences with sparse transformers," 2019.

[28] K. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlos, P. Hawkins, J. Davis, A. Mohiuddin, L. Kaiser, D. Belanger, L. Colwell, and A. Weller, "Rethinking attention with performers," 2021.

[29] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, P. Schuh, K. Shi, S. Tsvyashchenko, J. Maynez, A. B. Rao, P. Barnes, Y. Tay, N. M. Shazeer,

V. Prabhakaran, E. Reif, N. Du, B. C. Hutchinson, R. Pope, J. Bradbury, J. Austin, M. Isard, G. Gur-Ari, P. Yin, T. Duke, A. Levskaya, S. Ghemawat, S. Dev, H. Michalewski, X. García, V. Misra, K. Robinson, L. Fedus, D. Zhou, D. Ippolito, D. Luan, H. Lim, B. Zoph, A. Spiridonov, R. Sepassi, D. Dohan, S. Agrawal, M. Omernick, A. M. Dai, T. S. Pillai, M. Pellat, A. Lewkowycz, E. O. Moreira, R. Child, O. Polozov, K. Lee, Z. Zhou, X. Wang, B. Saeta, M. Diaz, O. Firat, M. Catasta, J. Wei, K. S. Meier-Hellstern, D. Eck, J. Dean, S. Petrov, and N. Fiedel, "Palm: Scaling language modeling with pathways," *ArXiv*, vol. abs/2204.02311, 2022.

[30] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (elus)," *arXiv: Learning*, 2016.

[31] S. F. Crone, M. Hibon, and K. Nikolopoulos, "Advances in forecasting with neural networks? empirical evidence from the nn3 competition on time series prediction," *International Journal of Forecasting*, vol. 27, no. 3, pp. 635–660, 2011, special Section 1: Forecasting with Artificial Neural Networks and Computational Intelligence Special Section 2: Tourism Forecasting. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169207011000616

[32] E. Dagum and S. Bianconcini, *Seasonal Adjustment Methods and Real Time Trend-Cycle Estimation*. Springer, 08 2016.

[33] Z. Dai, Z. Yang, Y. Yang, J. G. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-xl: Attentive language models beyond a fixed-length context," *CoRR*, vol. abs/1901.02860, 2019.

[34] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *ArXiv*, vol. abs/1810.04805, 2019.

[35] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *ArXiv*, vol. abs/1810.04805, 2019.

[36] J. Devore and K. Berk, *Modern Mathematical Statistics with Applications*, ser. Springer Texts in Statistics. Springer New York, 2011. [Online]. Available: https://books.google.no/books?id=cv3pcEJ7amMC

[37] D. Dickey and W. Fuller, "Distribution of the estimators for autoregressive time series with a unit root," *JASA. Journal of the American Statistical Association*, vol. 74, 06 1979.

[38] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves," in *IJCAI*, 2015.

[39] X. Dong and Y. Yang, "Searching for a robust neural architecture in four gpu hours," 2019.

[40] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," 2021.

[41] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*, 2nd ed. Springer Publishing Company, Incorporated, 2015.

[42] T. Elsken, J.-H. Metzen, and F. Hutter, "Simple and efficient architecture search for convolutional neural networks," 2017.

[43] T. Elsken, J. H. Metzen, and F. Hutter, "Efficient multi-objective neural architecture search via lamarckian evolution," 2019.

[44] T. Elsken, J.-H. Metzen, and F. Hutter, "Neural architecture search: A survey," *The Journal of Machine Learning Research*, vol. 20, no. 1, pp. 1997–2017, 2019.

[45] W. Fedus, B. Zoph, and N. M. Shazeer, "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity," *ArXiv*, vol. abs/2101.03961, 2021.

[46] B. Feng, D. Liu, and Y. Sun, *Evolving Transformer Architecture for Neural Machine Translation.* New York, NY, USA: Association for Computing Machinery, 2021, p. 273–274. [Online]. Available: https://doi.org/10.1145/3449726.3459441

[47] D. Floreano and C. Mattiussi, *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies.* The MIT Press, 2008.

[48] V. Flunkert, D. Salinas, and J. Gasthaus, "Deepar: Probabilistic forecasting with autoregressive recurrent networks," *ArXiv*, vol. abs/1704.04110, 2017.

[49] E. Galván and P. Mooney, "Neuroevolution in deep neural networks: Current trends and future challenges," *CoRR*, vol. abs/2006.05415, 2020.

[50] R. B. Girshick, "Fast r-cnn," *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1440–1448, 2015.

[51] A. Glassner, *Deep Learning: A Visual Approach*. No Starch Press, 2021. [Online]. Available: https://books.google.no/books?id=NgTyDwAAQBAJ

[52] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," in *Foundations of Genetic Algorithms*, ser. Foundations of Genetic Algorithms, G. J. RAWLINS, Ed. Elsevier, 1991, vol. 1, pp. 69–93. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780080506845500082

[53] F. Gomez and R. Miikkulainen, "Solving non-markovian control tasks with neuro-evolution," in *IJCAI*, 01 1999, pp. 1356–1361.

[54] I. J. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016, http://www.deeplearningbook.org.

[55] S. Gregor and A. Hevner, "Positioning and presenting design science research for maximum impact," *MIS Quarterly*, vol. 37, pp. 337–356, 06 2013.

[56] J. Grigsby, Z. Wang, and Y. Qi, "Long-range transformers for dynamic spatiotemporal forecasting," 2021.

[57] F. Gruau, "Automatic definition of modular neural networks," *Adaptive Behavior - ADAPT BEHAV*, vol. 3, pp. 151–183, 09 1994.

[58] C. Guan, X. Wang, and W. Zhu, "Autoattend: Automated attention representation search," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 3864–3874. [Online]. Available: https://proceedings.mlr.press/v139/guan21a.html

[59] A. Gulati, J. Qin, C.-C. Chiu, N. Parmar, Y. Zhang, J. Yu, W. Han, S. Wang, Z. Zhang, Y. Wu, and R. Pang, "Conformer: Convolution-augmented transformer for speech recognition," *ArXiv*, vol. abs/2005.08100, 2020.

[60] D. Ha, A. Dai, and Q. V. Le, "Hypernetworks," 2016.

[61] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015.

[62] P. He, X. Liu, J. Gao, and W. Chen, "Deberta: Decoding-enhanced bert with disentangled attention," *ArXiv*, vol. abs/2006.03654, 2021.

[63] D. Hendrycks and K. Gimpel, "Bridging nonlinearities and stochastic regularizers with gaussian error linear units," *ArXiv*, vol. abs/1606.08415, 2016.

[64] T. Hong, P. Pinson, and S. Fan, "Global energy forecasting competition 2012," *International Journal of Forecasting*, vol. 30, no. 2, pp. 357–363, 2014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169207013000745

[65] Y.-Q. Hu and Y. Yu, "A technical view on neural architecture search," *International Journal of Machine Learning and Cybernetics*, vol. 11, 04 2020.

[66] R. Hyndman and G. Athanasopoulos, *Forecasting: principles and practice*. OTexts, 2014. [Online]. Available: https://books.google.cz/books?id=gDuRBAAAQBAJ

[67] R. J. Hyndman and A. B. Koehler, "Another look at measures of forecast accuracy," *International Journal of Forecasting*, vol. 22, no. 4, pp. 679–688, 2006. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169207006000239

[68] A. Jaegle, F. Gimeno, A. Brock, A. Zisserman, O. Vinyals, and J. Carreira, "Perceiver: General perception with iterative attention," *CoRR*, vol. abs/2103.03206, 2021.

[69] K. Jing, J. Xu, and H. X. Zugeng, "Nasabn: A neural architecture search framework for attention-based networks," in *2020 International Joint Conference on Neural Networks (IJCNN)*, 2020, pp. 1–7.

[70] A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret, "Transformers are rnns: Fast autoregressive transformers with linear attention," in *ICML*, 2020.

[71] S. M. Kazemi, R. Goel, S. Eghbali, J. Ramanan, J. Sahota, S. Thakur, S. Wu, C. Smyth, P. Poupart, and M. Brubaker, "Time2vec: Learning a vector representation of time," 2019.

[72] G. Ke, D. He, and T.-Y. Liu, "Rethinking positional encoding in language pre-training," 2021.

[73] J. Kim, J. Wang, S. Kim, and Y. Lee, "Evolved Speech-Transformer: Applying Neural Architecture Search to End-to-End Automatic Speech Recognition," in *Proc. Interspeech 2020*, 2020, pp. 1788–1792. [Online]. Available: http://dx.doi.org/10.21437/Interspeech.2020-1233

[74] N. Kitaev, Łukasz Kaiser, and A. Levskaya, "Reformer: The efficient transformer," 2020.

[75] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, "Self-normalizing neural networks," *ArXiv*, vol. abs/1706.02515, 2017.

[76] A. Klein, S. Falkner, S. Bartels, P. Hennig, and F. Hutter, "Fast bayesian optimization of machine learning hyperparameters on large datasets," 2017.

[77] U. Knief and W. Forstmeier, "Violating the normality assumption may be the lesser of two evils," *bioRxiv*, 2018.

[78] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "Albert: A lite bert for self-supervised learning of language representations," *ArXiv*, vol. abs/1909.11942, 2020.

[79] J. Lee, Y. Lee, J. Kim, A. R. Kosiorek, S. Choi, and Y. W. Teh, "Set transformer: A framework for attention-based permutation-invariant neural networks," 2019.

[80] J. Lee-Thorp, J. Ainslie, I. Eckstein, and S. Ontañón, "Fnet: Mixing tokens with fourier transforms," *CoRR*, vol. abs/2105.03824, 2021.

[81] H. Levene, "Robust tests for equality of variance. in 'contributions to probability and statistics'.(eds i olkin, sg ghurye, w hoeffeling, wg madow, hb mann) pp. 278–292," 1960.

[82] C. Li, J. Peng, L. Yuan, G. Wang, X. Liang, L. Lin, and X. Chang, "Block-wisely supervised neural architecture search with knowledge distillation," 11 2019.

[83] H. Li, A. Y. C. Wang, Y. Liu, D. Tang, Z. Lei, and W. Li, "An augmented transformer architecture for natural language generation tasks," 2019.

[84] S. Li, X. Jin, Y. Xuan, X. Zhou, W. Chen, Y.-X. Wang, and X. Yan, "Enhancing the locality and breaking the memory bottleneck of transformer on time series forecasting," 2020.

[85] T. Lin, Y. Wang, X. Liu, and X. Qiu, "A survey of transformers," *ArXiv*, vol. abs/2106.04554, 2021.

[86] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," 2018.

[87] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, "Hierarchical representations for efficient architecture search," 2018.

[88] H. Liu, K. Simonyan, and Y. Yang, "Darts: Differentiable architecture search," 2019.

[89] J. Liu, S. Zhou, Y. Wu, K. Chen, W. Ouyang, and D. Xu, "Block proposal neural architecture search," *Trans. Img. Proc.*, vol. 30, p. 15–25, jan 2021. [Online]. Available: https://doi.org/10.1109/TIP.2020.3028288

[90] J. Liu, H. Li, G. Song, X. Huang, and Y. Liu, "Uninet: Unified architecture search with convolution, transformer, and MLP," *CoRR*, vol. abs/2110.04035, 2021.

[91] R. Liu, J. Lehman, P. Molino, F. P. Such, E. Frank, A. Sergeev, and J. Yosinski, "An intriguing failing of convolutional neural networks and the coordconv solution," in *NeurIPS*, 2018.

[92] X. Liu, H.-F. Yu, I. Dhillon, and C.-J. Hsieh, "Learning to encode position for transformer with continuous dynamical model," 2020.

[93] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *ArXiv*, vol. abs/1907.11692, 2019.

[94] Z. Liu, H. Mao, C. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," in *A ConvNet for the 2020s*, 2022.

[95] V. Lopes, S. Alirezazadeh, and L. A. Alexandre, "Epe-nas: Efficient performance estimation without training for neural architecture search," in *Artificial Neural Networks and Machine Learning – ICANN 2021*, I. Farkaš, P. Masulli, S. Otte, and S. Wermter, Eds. Cham: Springer International Publishing, 2021, pp. 552–563.

[96] V. Lopes, M. Santos, B. Degardin, and L. Alexandre, "Guided evolution for neural architecture search," 10 2021.

[97] I. Loshchilov and F. Hutter, "Fixing weight decay regularization in adam," *ArXiv*, vol. abs/1711.05101, 2017.

[98] R. Luo, F. Tian, T. Qin, and T.-Y. Liu, "Neural architecture optimization," in *NeurIPS*, ser. NIPS'18. Red Hook, NY, USA: Curran Associates Inc., 2018, p. 7827–7838.

[99] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, "Neural architecture optimization," 2019.

[100] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," 2015.

[101] A. Mahmoud and A. Mohammed, *A Survey on Deep Learning for Time-Series Forecasting*. Cham: Springer International Publishing, 2021, pp. 365–392. [Online]. Available: https://doi.org/10.1007/978-3-030-59338-4_19

[102] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, "The m5 competition: Background, organization, and implementation," *International Journal of Forecasting*, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169207021001187

[103] S. Makridakis and M. Hibon, "The m3 competition: results, conclusions and implications," *International Journal of Forecasting*, vol. 16, no. 4, pp. 451–476, 2000. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169207000000571

[104] S. Makridakis, C. Chatfield, M. Hibon, M. Lawrence, T. Mills, K. Ord, and L. F. Simmons, "The m2-competition: A real-time judgmentally based forecasting study," *International Journal of Forecasting*, vol. 9, no. 1, pp. 5–22, 1993. [Online]. Available: https://www.sciencedirect.com/science/article/pii/016920709390044N

[105] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, "The m4 competition: 100,000 time series and 61 forecasting methods," *International Journal of Forecasting*, vol. 36, no. 1, pp. 54–74, 2020, m4 Competition. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0169207019301128

[106] E. M. Malta, S. Avila, and E. Borin, "Exploring the cost-benefit of aws ec2 gpu instances for deep learning applications," in *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, ser. UCC'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 21–29. [Online]. Available: https://doi.org/10.1145/3344341.3368814

[107] S. Mandava, S. Migacz, and A. F. Florea, "Pay attention when required," *CoRR*, vol. abs/2009.04534, 2020.

[108] K. Maziarz, M. Tan, A. Khorlin, M. Georgiev, and A. Gesmundo, "Evolutionary-neural hybrid agents for architecture search," 2020.

[109] J. Mellor, J. Turner, A. Storkey, and E. J. Crowley, "Neural architecture search without training," 2021. [Online]. Available: https://openreview.net/forum?id=g4E6SAAvACo

[110] R. Miikkulainen, J. Z. Liang, E. Meyerson, A. Rawal, D. Fink, O. Francon, B. Raju, H. Shahrzad, A. Navruzyan, N. Duffy, and B. Hodjat, "Evolving deep neural networks," *CoRR*, vol. abs/1703.00548, 2017.

[111] D. J. Montana and L. Davis, "Training feedforward neural networks using genetic algorithms," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence - Volume 1*, ser. IJCAI'89. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1989, p. 762–767.

[112] D. Montgomery, C. Jennings, and M. Kulahci, *Introduction to Time Series Analysis and Forecasting*, ser. Wiley Series in Probability and Statistics. Wiley, 2015. [Online]. Available: https://books.google.no/books?id=-qaFi0oOPAYC

[113] D. Moriarty and R. Miikkulainen, "Efficient reinforcement learning through symbiotic evolution," *Machine Learning*, vol. 22, 01 1995.

[114] T. Q. Nguyen and J. Salazar, "Transformers without tears: Improving the normalization of self-attention," *ArXiv*, vol. abs/1910.05895, 2019.

[115] B. Oates, *Researching Information Systems and Computing*. SAGE Publications, 2006.

[116] B. N. Oreshkin, D. Carpov, N. Chapados, and Y. Bengio, "N-beats: Neural basis expansion analysis for interpretable time series forecasting," *ArXiv*, vol. abs/1905.10437, 2020.

[117] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *NeurIPS*, 2019.

[118] H. Peng, N. Pappas, D. Yogatama, R. Schwartz, N. A. Smith, and L. Kong, "Random feature attention," *CoRR*, vol. abs/2103.02143, 2021.

[119] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, "Efficient neural architecture search via parameter sharing," 2018.

[120] J. Qiu, H. Ma, O. Levy, S. W. tau Yih, S. Wang, and J. Tang, "Blockwise self-attention for long document understanding," 2020.

[121] X. Qiu, T. Sun, Y. Xu, Y. Shao, N. Dai, and X. Huang, "Pre-trained models for natural language processing: A survey," *ArXiv*, vol. abs/2003.08271, 2020.

[122] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," *OpenAI*, 2018.

[123] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," in *OpenAI*, 2019.

[124] J. W. Rae, A. Potapenko, S. M. Jayakumar, and T. P. Lillicrap, "Compressive transformers for long-range sequence modelling," *CoRR*, vol. abs/1911.05507, 2019.

[125] C. Raffel, N. M. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *ArXiv*, vol. abs/1910.10683, 2020.

[126] Ramachandran, B. Zoph, and Q. V. Le, "Searching for activation functions," 2017.

[127] P. Ramachandran, B. Zoph, and Q. V. Le, "Swish: a self-gated activation function," *arXiv: Neural and Evolutionary Computing*, 2017.

[128] E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. Le, and A. Kurakin, "Large-scale evolution of image classifiers," 2017.

[129] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," 2019.

[130] P. Ren, Y. Xiao, X. Chang, P. Huang, Z. Li, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," *CoRR*, vol. abs/2006.02903, 2020.

[131] A. Rives, J. Meier, T. Sercu, S. Goyal, Z. Lin, J. Liu, D. Guo, M. Ott, C. L. Zitnick, J. Ma, and R. Fergus, "Biological structure and function emerge from scaling unsupervised learning to 250 million protein sequences," *Proceedings of the National Academy of Sciences*, vol. 118, no. 15, 2021. [Online]. Available: https://www.pnas.org/content/118/15/e2016239118

[132] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65 6, pp. 386–408, 1958.

[133] A. Roy, M. Saffar, A. Vaswani, and D. Grangier, "Efficient content-based sparse attention with routing transformers," 2020.

[134] S. J. Russell and P. Norvig, *Artificial Intelligence: a modern approach*, 4th ed. Pearson, 2020.

[135] S. S. Makridakis and M. Hibon, "Accuracy of forecasting: An empirical investigation," *Journal of the Royal Statistical Society. Series A (General)*, vol. 142, 01 1979.

[136] S. S. SHAPIRO and M. B. WILK, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3-4, pp. 591–611, dec 1965. [Online]. Available: https://doi.org/10.1093/biomet/52.3-4.591

[137] P. Shaw, J. Uszkoreit, and A. Vaswani, "Self-attention with relative position representations," 2018.

[138] N. M. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. V. Le, G. E. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," *ArXiv*, vol. abs/1701.06538, 2017.

[139] L. Shen and Y. Wang, "Tcct: Tightly-coupled convolutional transformer on time series forecasting," 2021.

[140] S. Shen, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "Rethinking batch normalization in transformers," *ArXiv*, vol. abs/2003.07845, 2020.

[141] T. Shi, Y. Keneshloo, N. Ramakrishnan, and C. Reddy, "Neural abstractive text summarization with sequence-to-sequence models," 12 2018.

[142] R. Shin, C. Packer, and D. X. Song, "Differentiable neural network architecture search," in *ICLR*, 2018.

[143] D. R. So, C. Liang, and Q. V. Le, "The evolved transformer," 2019.

[144] D. R. So, W. Mańke, H. Liu, Z. Dai, N. Shazeer, and Q. V. Le, "Primer: Searching for efficient transformers for language modeling," 2021.

[145] K. Song, X. Tan, T. Qin, J. Lu, and T.-Y. Liu, "Mass: Masked sequence to sequence pre-training for language generation," in *ICML*, 2019.

[146] K. Stanley, D. D'Ambrosio, and J. Gauci, "A hypercube-based encoding for evolving large-scale neural networks," *Artificial life*, vol. 15, pp. 185–212, 02 2009.

[147] K. O. Stanley and R. Miikkulainen, "Evolving neural networks through augmenting topologies," *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002. [Online]. Available: http://nn.cs.utexas.edu/?stanley:ec02

[148] J. Su, Y. Lu, S. Pan, B. Wen, and Y. Liu, "Roformer: Enhanced transformer with rotary position embedding," *ArXiv*, vol. abs/2104.09864, 2021.

[149] M. Suganuma, S. Shirakawa, and T. Nagao, "A genetic programming approach to designing convolutional neural network architectures," 2017.

[150] S. Sukhbaatar, E. Grave, P. Bojanowski, and A. Joulin, "Adaptive attention span in transformers," *CoRR*, vol. abs/1905.07799, 2019.

[151] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," 2014.

[152] A. Tadjer, A. Hong, and R. Bratvold, "Machine learning based decline curve analysis for short-term oil production forecast," *Energy Exploration & Exploitation*, vol. 39, p. 014459872110117, 05 2021.

[153] Y. Tay, D. Bahri, D. Metzler, D. Juan, Z. Zhao, and C. Zheng, "Synthesizer: Rethinking self-attention in transformer models," *CoRR*, vol. abs/2005.00743, 2020.

[154] Y. Tay, D. Bahri, L. Yang, D. Metzler, and D.-C. Juan, "Sparse sinkhorn attention," 2020.

[155] S. J. Taylor and B. Letham, "Forecasting at scale," *PeerJ Prepr.*, vol. 5, p. e3190, 2017.

[156] O. Triebe, H. Hewamalage, P. Pilyugina, N. P. Laptev, C. Bergmeir, and R. Rajagopal, "Neuralprophet: Explainable forecasting at scale," *ArXiv*, vol. abs/2111.15397, 2021.

[157] H. Tsai, J. Ooi, C.-S. Ferng, H. W. Chung, and J. Riesa, "Finding fast transformers: One-shot neural architecture search by component composition," 2020.

[158] V. Vaishnavi and B. Kuechler, "Design science research in information systems," *Association for Information Systems*, 01 2004.

[159] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[160] A. Vyas, A. Katharopoulos, and F. Fleuret, "Fast transformers with clustered attention," *CoRR*, vol. abs/2007.04825, 2020.

[161] H. Wang, Z. Wu, Z. Liu, H. Cai, L. Zhu, C. Gan, and S. Han, "Hat: Hardware-aware transformers for efficient natural language processing," *ArXiv*, vol. abs/2005.14187, 2020.

[162] Q. Wang, B. Li, T. Xiao, J. Zhu, C. Li, D. F. Wong, and L. S. Chao, "Learning deep transformer models for machine translation," *ArXiv*, vol. abs/1906.01787, 2019.

[163] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," 2020.

[164] T. Wei, C. Wang, Y. Rui, and C. W. Chen, "Network morphism," 2016.

[165] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, no. 3–4, p. 229–256, may 1992. [Online]. Available: https://doi.org/10.1007/BF00992696

[166] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2601248.2601268

[167] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search," 2019.

[168] F. Wu, A. Fan, A. Baevski, Y. N. Dauphin, and M. Auli, "Pay less attention with lightweight and dynamic convolutions," *CoRR*, vol. abs/1901.10430, 2019.

[169] M.-T. Wu, H.-I. Lin, and C.-W. Tsai, "A training-free genetic neural architecture search," in *Proceedings of the 2021 ACM International Conference on Intelligent Computing and Its Emerging Applications*, ser. ACM ICEA '21. New York, NY, USA: Association for Computing Machinery, 2022, p. 65–70. [Online]. Available: https://doi.org/10.1145/3491396.3506510

[170] N. Wu, B. Green, X. Ben, and S. O'Banion, "Deep transformer models for time series forecasting: The influenza prevalence case," 2020.

[171] R. Xiong, Y. Yang, D. He, K. Zheng, S. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T.-Y. Liu, "On layer normalization in the transformer architecture," *ArXiv*, vol. abs/2002.04745, 2020.

[172] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *ArXiv*, vol. abs/1505.00853, 2015.

[173] J. Xu, X. Sun, Z. Zhang, G. Zhao, and J. Lin, "Understanding and improving layer normalization," *ArXiv*, vol. abs/1911.07013, 2019.

[174] R. Yamashita, M. Nishio, R. K. G. Do, and K. Togashi, "Convolutional neural networks: an overview and application in radiology," *Insights into Imaging*, vol. 9, pp. 611 – 629, 2018.

[175] H. Yan, B. Deng, X. Li, and X. Qiu, "Tener: Adapting transformer encoder for named entity recognition," 2019.

[176] A. Yang, J. Lin, R. Men, C. Zhou, L. Jiang, X. Jia, A. Wang, J. Zhang, J. Wang, Y. Li, D. Zhang, W. Lin, L. Qu, J. Zhou, and H. Yang, "M6-t: Exploring sparse expert models and beyond," 2021.

[177] S. Yang, Y. Tian, X. Xiang, S. Peng, and X. Zhang, "Accelerating evolutionary neural architecture search via multi-fidelity evaluation," *CoRR*, vol. abs/2108.04541, 2021.

[178] Y. Yang, L. Wang, S. Shi, P. Tadepalli, S. Lee, and Z. Tu, "On the sub-layer functionalities of transformer decoder," in *FINDINGS*, 2020.

[179] X. Yao, "A review of evolutionary artificial neural networks," *International Journal of Intelligent Systems*, vol. 8, no. 4, pp. 539–567, 1993. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/int.4550080406

[180] S. Yi, X. Chen, and C. Tang, "Tsformer: Time series transformer for tourism demand forecasting," 2021.

[181] K. Yu, C. Sciuto, M. Jaggi, C. Musat, and M. Salzmann, "Evaluating the search phase of neural architecture search," 2019.

[182] M. Zaheer, G. Guruganesh, A. Dubey, J. Ainslie, C. Alberti, S. Ontañón, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, "Big bird: Transformers for longer sequences," *CoRR*, vol. abs/2007.14062, 2020.

[183] A. Zela, A. Klein, S. Falkner, and F. Hutter, "Towards automated deep learning: Efficient joint neural architecture and hyperparameter search," 2018.

[184] G. Zerveas, S. Jayaraman, D. Patel, A. Bhamidipaty, and C. Eickhoff, "A transformer-based framework for multivariate time series representation learning," 2020.

[185] S. Zhai, W. Talbott, N. Srivastava, C. Huang, H. Goh, R. Zhang, and J. M. Susskind, "An attention free transformer," *CoRR*, vol. abs/2105.14103, 2021.

[186] B. Zhang and R. Sennrich, "Root mean square layer normalization," in *NeurIPS*, 2019.

[187] H. Zhang, Y. Jin, R. Cheng, and K. Hao, "Efficient evolutionary search of attention convolutional networks via sampled training and node inheritance," *IEEE Transactions on Evolutionary Computation*, vol. 25, no. 2, pp. 371–385, 2021.

[188] Y. Zhao, L. Dong, Y. Shen, Z. Zhang, F. Wei, and W. Chen, "Memory-efficient differentiable transformer architecture search," *CoRR*, vol. abs/2105.14669, 2021.

[189] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu, "Practical block-wise neural network architecture generation," 2018.

[190] H. Zhou, S. Zhang, J. Peng, S. Zhang, J. Li, H. Xiong, and W. Zhang, "Informer: Beyond efficient transformer for long sequence time-series forecasting," 2021.

[191] C. Zhu, W. Ping, C. Xiao, M. Shoeybi, T. Goldstein, A. Anandkumar, and B. Catanzaro, "Long-short transformer: Efficient transformers for language and vision," in *NeurIPS*, 2021.

[192] H. Zhu, Z. An, C. Yang, K. Xu, and Y. Xu, "EENA: Efficient evolution of neural architecture," 2019.

[193] W. Zhu, X. Wang, X. Qiu, Y. Ni, and G. Xie, "Autotrans: Automating transformer design via reinforced architecture search," *CoRR*, vol. abs/2009.02070, 2020.

[194] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2017.

[195] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," pp. 8697–8710, 2018.

Appendices

Appendix A: Difference Factor

This section provides the source code used to calculate the *difference factor* metric described in section 4.6.2. The calculation is split into several functions which evaluate differences between different parts of the designed genotype encoding (section 4.5). The source code of all functions is provided in the following environments: 6.3.1, 6.3.2, 6.3.3, 6.3.4, 6.3.5 and 6.3.6.

```
def parameter_difference_factor(
    parameter1: IParameter, parameter2: IParameter, eps=1e-5
) -> float:
    if parameter1.get_type() != parameter2.get_type():
        return 1.0

    param1_allowed_values = parameter1.get_allowed_values()
    param2_allowed_values = parameter2.get_allowed_values()
    if param1_allowed_values != param2_allowed_values:
        return 1.0

    param1_value = parameter1.get_value()
    param2_value = parameter2.get_value()
    parameter1_index = param1_allowed_values.index(param1_value)
    parameter2_index = param2_allowed_values.index(param2_value)

    param_diff_factor = abs(parameter1_index - parameter2_index) / (
        len(param1_allowed_values) - 1 + eps
    )
    return param_diff_factor
```

Source code 6.3.1: The implementation of *parameter difference factor* function which calculates the difference between values of two evolvable parameters.

```python
def module_difference_factor(
    module1: GenotypeModule, module2: GenotypeModule, eps=1e-5
) -> float:
    if module1.get_category() != module2.get_category():
        return 1.0
    # If a module is from the same category but not the same module, return 0.75
    if module1.get_module_path() != module2.get_module_path():
        return 0.75

    module1_parameters = module1.get_parameters()
    module2_parameters = module2.get_parameters()
    if len(module1_parameters) == 0 or len(module2_parameters) == 0:
        return 0.0

    param_diff_factors = []
    for idx in range(len(module1_parameters)):
        module1_param = module1_parameters[idx]
        module2_param = module2_parameters[idx]
        diff_factor = parameter_difference_factor(module1_param, module2_param, eps)
        param_diff_factors.append(diff_factor)

    # Compensate for the "same category, different module" case from above
    module_diff_factor = mean(param_diff_factors) * 0.75
    return module_diff_factor
```

Source code 6.3.2: The implementation of *module difference factor* function which calculates the difference between two *Genotype modules*.

```python
def branch_difference_factor(
    cell1: GenotypeCell, cell2: GenotypeCell, branch: GenotypeModuleBranch, eps=1e-5
) -> float:
    cell1_branch_module_count = cell1.module_count(branch)
    cell2_branch_module_count = cell2.module_count(branch)
    min_number_of_modules = min(cell1_branch_module_count, cell2_branch_module_count)

    module_diff_factors = []
    for idx in range(min_number_of_modules):
        module1 = cell1.get_module(branch, idx)
        module2 = cell2.get_module(branch, idx)
        diff_factor = module_difference_factor(module1, module2, eps)
        module_diff_factors.append(diff_factor)

    mean_diff_factor = mean(module_diff_factors) if len(module_diff_factors) > 0 else 0
    module_count_difference = abs(
        cell1_branch_module_count - cell2_branch_module_count
    ) / (max(cell1_branch_module_count, cell2_branch_module_count) + eps)
    branch_diff_factor = mean([mean_diff_factor, module_count_difference])
    return branch_diff_factor
```

Source code 6.3.3: The implementation of *branch difference factor* function which calculates the difference between two branches of *Genotype cell*.

```python
def cell_difference_factor(cell1: GenotypeCell, cell2: GenotypeCell, eps=1e-5) -> float:
    cell1_left_branch_module_count = cell1.module_count(GenotypeModuleBranch.LEFT)
    cell2_left_branch_module_count = cell2.module_count(GenotypeModuleBranch.LEFT)
    left_branch_min_number_of_modules = min(
        cell1_left_branch_module_count, cell2_left_branch_module_count
    )

    cell1_right_branch_module_count = cell1.module_count(GenotypeModuleBranch.RIGHT)
    cell2_right_branch_module_count = cell2.module_count(GenotypeModuleBranch.RIGHT)
    right_branch_min_number_of_modules = min(
        cell1_right_branch_module_count, cell2_right_branch_module_count
    )

    if (
        left_branch_min_number_of_modules == 0
        and right_branch_min_number_of_modules == 0
    ):
        return 0.0

    left_branch_diff_factor = branch_difference_factor(
        cell1, cell2, GenotypeModuleBranch.LEFT, eps
    )
    right_branch_diff_factor = branch_difference_factor(
        cell1, cell2, GenotypeModuleBranch.RIGHT, eps
    )
    cell_diff_factor = mean([left_branch_diff_factor, right_branch_diff_factor])
    return cell_diff_factor
```

Source code 6.3.4: The implementation of *cell difference factor* function which calculates the difference between two *Genotype cells*.

```python
def layer_difference_factor(
    layer1: GenotypeLayer, layer2: GenotypeLayer, eps=1e-5
) -> float:
    layer1_cells = layer1.cells()
    layer2_cells = layer2.cells()
    min_number_of_cells = min(len(layer1_cells), len(layer2_cells))

    if min_number_of_cells == 0:
        return 0.0

    cell_diff_factors = []
    for idx in range(min_number_of_cells):
        cell1 = layer1_cells[idx]
        cell2 = layer2_cells[idx]

        # Consider the difference between inputs of the compared cells
        left_cell_input_index_diff_factor = abs(cell1[0] - cell2[0]) / (
            max(len(layer1_cells), len(layer2_cells)) + eps
        )
        right_cell_input_index_diff_factor = abs(cell1[1] - cell2[1]) / (
            max(len(layer1_cells), len(layer2_cells)) + eps
        )
        index_diff_factor = mean(
            [left_cell_input_index_diff_factor, right_cell_input_index_diff_factor]
        )

        cell_diff_factor = cell_difference_factor(cell1[2], cell2[2], eps)
        cell_diff_factors.append(mean([index_diff_factor, cell_diff_factor]))

    layer_diff_factor = mean(cell_diff_factors)
    return layer_diff_factor
```

Source code 6.3.5: The implementation of *layer difference factor* function which calculates the difference between two *Genotype layers*.

```
def genotype_difference_factor(
    genotype1: Genotype, genotype2: Genotype, eps=1e-5
) -> float:
    # Genotype hyperparameters
    genotype1_params = genotype1.parameters()
    genotype2_params = genotype2.parameters()

    param_diff_factors = []
    for idx in range(len(genotype1_params)):
        param1 = genotype1_params[idx]
        param2 = genotype2_params[idx]
        diff_factor = parameter_difference_factor(param1, param2, eps)
        param_diff_factors.append(diff_factor)

    # Genotype layer
    # There is only one Genotype layer per Genotype stacked k times as defined by hyperparameters
    genotype1_layer = genotype1.get_layer(0)
    genotype2_layer = genotype2.get_layer(0)
    layer_diff_factor = layer_difference_factor(genotype1_layer, genotype2_layer, eps)

    genotype_diff_factor = mean([layer_diff_factor] + param_diff_factors)
    return genotype_diff_factor
```

Source code 6.3.6: The implementation of *genotype difference factor* function which calculates the difference between two *Genotypes*.

Appendix B: Module Parameters Configuration

The experiments performed during the third experimental phase (subsection 5.3.3) utilized several Genotype modules some of which had evolvable parameters attached. Table 6.1 lists parameters of the modules used in the first experiment (section 5.3.3) while Table 6.2 and Table 6.3 describe parameters of the modules utilized in the second experiment (section 5.3.3).

Module	**Parameter**	**Type**	**Allowed values**	**Description**
Multi-head attention	head count	Set	[1, 2, 4, 8]	Number of attention heads
Feed forward	forward expansion	Set	(1, 8), step: 1	Expansion ratio of the fully-connected layers
Feed forward	dropout	Range	(0.0, 0.3), step: 0.05	Dropout probability

Table 6.1: The configuration of evolvable parameters attached to the modules used in Experiment 1 (section 5.3.3) performed during Phase 3 (subsection 5.3.3).

Module (part 1)	Parameter	Type	Allowed values	Description
Big Bird attention	head count	Set	[1, 2, 4, 8]	Number of attention heads
Longformer attention	head count	Set	[1, 2, 4, 8]	Number of attention heads
Longformer attention	dropout	Range	(0.0, 0.3), step: 0.05	Dropout probability
Conformer attention	dropout	Range	(0.0, 0.3), step: 0.05	Dropout probability
Reformer attention	head count	Set	[1, 2, 4, 8]	Number of attention heads
Reformer attention	chunk length divider	Set	[2, 4, 8]	Length of chunks of clustered elements specified relative to the model input size
Reformer attention	dropout	Range	(0.0, 0.3), step: 0.05	Dropout probability
Adaptive span attention	head count	Set	[1, 2, 4, 8]	Number of attention heads
Adaptive span attention	span length init	Range	(0.0, 1.0), step: 0.5	Initial length of the attention span
Adaptive span attention	span ramp	Set	[10, 20, 40, 80]	Length of the soft mask ramp on each side of the span
TransformerXL attention	head count	Set	[1, 2, 4, 8]	Number of attention heads
Multi-head attention	head count	Set	[1, 2, 4, 8]	Number of attention heads
Multi-head attention	query preprocess	Set	[None, Rotary positional encoding, TUPE absolute positional encoding, Spatial depth-wise convolution]	Augmentation applied to the query matrix
Multi-head attention	key preprocess	Set	[None, Rotary positional encoding, TUPE absolute positional encoding, Spatial depth-wise convolution]	Augmentation applied to the key matrix
Multi-head attention	value preprocess	Set	[None, Rotary positional encoding, TUPE absolute positional encoding, Spatial depth-wise convolution]	Augmentation applied to the value matrix
Multi-head attention	scores preprocess	Set	[None, DeBERTa relative positional encoding, T5 relative positional encoding, TUPE relative positional encoding]	Augmentation applied to the attention scores matrix

Table 6.2: The configuration of evolvable parameters attached to the modules used in Experiment 2 (section 5.3.3) performed during Phase 3 (subsection 5.3.3).

Module (part 2)	Parameter	Type	Allowed values	Description
Convolution 1D	kernel size	Set	[3, 6, 12]	Size of the convolution kernel
Convolution 1D	dilation	Set	[1, 2, 3]	Spacing between the kernel elements
Convolution 2D	kernel width	Set	[3, 6, 12]	Width of the convolution kernel
Convolution 2D	kernel height	Set	[3, 6, 12]	Height of the convolution kernel
Convolution 2D	horizontal dilation	Set	[1, 2, 3]	Horizontal spacing between the kernel elements
Convolution 2D	vertical dilation	Set	[1, 2, 3]	Vertical spacing between the kernel elements
Depth-wise separable convolution	kernel size	Set	[3, 6, 12]	Size of the convolution kernel
Depth-wise separable convolution	dilation	Set	[1, 2, 3]	Spacing between the kernel elements
Inverted bottleneck	kernel size	Set	[3, 6, 12]	Size of the convolution kernel
Inverted bottleneck	dilation	Set	[1, 2, 3]	Spacing between the kernel elements
Inverted bottleneck	forward expansion	Set	[1, 2, 4, 8]	Expansion ratio of the convolution layers
CoordConv	kernel size	Set	[3, 6, 12]	Size of the convolution kernel
CoordConv	dilation	Set	[1, 2, 3]	Spacing between the kernel elements
Max pooling 1D	kernel size	Set	[3, 6, 12, 24]	Size of the pooling kernel
Max pooling 2D	kernel width	Set	[3, 6, 12, 24]	Width of the pooling kernel
Max pooling 2D	kernel height	Set	[3, 6, 12, 24]	Height of the pooling kernel
Alpha Dropout	dropout	Range	(0.0, 0.3), step: 0.05	Dropout probability
Dropout	dropout	Range	(0.0, 0.3), step: 0.05	Dropout probability
Dropout 2D	dropout	Range	(0.0, 0.3), step: 0.05	Dropout probability
Feed forward	forward expansion	Range	(1, 8), step: 1	Expansion ratio of the fully-connected layers
Feed forward	activation	Set	[ELU, GeLU, Leaky ReLU, ReLU, SiLU, Squared ReLU]	Activation function placed in between the fully connected layers
Feed forward	dropout	Range	(0.0, 0.3), step: 0.05	Dropout probability

Table 6.3: The configuration of evolvable parameters attached to the modules used in Experiment 2 (section 5.3.3) performed during Phase 3 (subsection 5.3.3).