LUDWIG-MAXIMILIANS-UNIVERSITY MUNICH
DEPARTMENT OF STATISTICS
FACULTY OF MATHEMATICS, INFORMATICS AND STATISTICS

in Cooperation with

Neural Architecture Search for Genomic Sequence Data

Master Thesis

Author Amadeu Scheppach
Supervisor Prof. Dr. Bernd Bischl, Dr. Mina Rezaei, Martin Binder
Date Munich, December 6, 2021

Contents

List of Figures

List of Algorithms

List of Tables

List of symbols and abbreviations

DNA Deoxyribonucleic Acid

NAS Neural Architecture Search

genomeNAS Genome Neural Architecture Search

OSP-NAS Operation Set Pruning - Neural Architecture Search

DARTS Differentiable Architecture Search

ENAS Efficient Neural Architecture Search via Parameter Sharing

BONAS Bayesian Optimized Neural Architecture Search

P-DARTS Progressive Differentiable Architecture Search

CWP-DARTS Continuous Weight Sharing Progressive Differentiable Architecture Search

DEP-DARTS Discarding Edges Progressive Differentiable Architecture Search

genomeDARTS Genome Differentiable Architecture Search

genomeP-DARTS Genome Progressive Differentiable Architecture Search

genomeDEP-DARTS Genome Discarding Edges Progressive Differentiable Architecture Search

genomeCWP-DARTS Genome Continuous Weight Sharing Progressive Differentiable Architecture Search

genomeBONAS Genome Bayesian Optimized Neural Architecture Search

genomeOSP-NAS Genome Discarding Operation-Sets Neural Architecture Search

BSR Bayesian sigmoid regressor

GCN Graph Convolutional Network

UCB upper confidence bound

EI expected improvement

MPI maximum probability of improvement

AutoML Automated Machine Learning

CNN Convolutional Neural Network

RNN Recurrent Neural Network

DAG directed acyclic graph

LSTM Long Short Term Memory

RHN Recurrent Highway Networks

Abstract

The topic of this thesis is Neural Architecture Search (NAS) for genomic sequence data. We present the application of NAS algorithms to design high-performing deep learning architectures for genomic sequences. Based on popular NAS approaches, we implement new NAS algorithms for genomic data, which we call Genome Differentiable Architecture Search (genomeDARTS), Genome Progressive Differentiable Architecture Search (genomeP-DARTS), and Genome Bayesian Optimized Neural Architecture Search (genomeBONAS). Furthermore, we build novel NAS methods such as Continuous Weight Sharing Progressive Differentiable Architecture Search (CWP-DARTS), Discarding Edges Progressive Differentiable Architecture Search (DEP-DARTS), and Operation Set Pruning - Neural Architecture Search (OSP-NAS). The novel feature of our provided work is the unique combination of a convolutional architecture with a recurrent architecture, which showed superior performance compared to pure convolution neural network architectures. We benchmark these Genome Neural Architecture Search (genomeNAS) algorithms, by searching for the best architectures and comparing them against the current state-of-the-art genomic deep learning models. All NAS algorithms are applied to the DeepSEA data set. The proposed NAS algorithms show state-of-the-art performance on the DeepSEA task and outperform all baseline models as well as randomly sampled models. The self-designed algorithm genomeOSP-NAS achieves state-of-the-art performance and is among the best performing genomeNAS algorithms.

1 Introduction

Deep Learning is an attractive solution for a variety of tasks, including autonomous driving, image object recognition, or machine translation (Huang and Y. Chen, 2020; Zhao et al., 2019; Young et al., 2018). In the field of bioinformatics deep learning algorithms also gained popularity (Eraslan et al., 2019). In genomics for example, deep learning models can be used to model the properties and functions of Deoxyribonucleic Acid (DNA) sequences (Jian Zhou, 2015; Quang and Xie, 2015; H. Zhang et al., 2019). Especially, the prediction of the function of non-coding DNA is an interesting field, as 98 percent of the human genome is non-coding and 93 percent of disease-associated variants lie in these regions (H. Zhang et al., 2019). While in the past human experts developed and configured these deep learning models and their architectures based on literature or based on experience, Automated Machine Learning (AutoML) approaches are nowadays able to achieve better results than human experts by automatically searching for the appropriate settings of a deep learning model (Elsken, J. H. Metzen, and Hutter, 2019).
NAS is a sub-field of AutoML, which aims to automatically find the optimal architecture of a deep learning model (Hutter, Kotthoff, and Vanschoren, 2019). Recently there has been a rapid development of new NAS algorithms. While earlier methods needed a vast amount of computational resources, new algorithms such as Efficient Neural Architecture Search via Parameter Sharing (ENAS) or Differentiable Architecture Search (DARTS) focused on speeding up NAS algorithms (Pham et al., 2018; H. Liu, Simonyan, and Yang, 2019). For example, a popular RL based NAS approach required 2000 GPU days (Zoph, Vasudevan, et al., 2018) and an evolution based NAS algorithm (Real et al., 2019) required 3150 GPU days. On the other side, DARTS lasts 4 GPU days and ENAS 0.5 GPU days (Pham et al., 2018; H. Liu, Simonyan, and Yang, 2019).

Main Contribution. NAS algorithms showed competitive performance on image classification tasks (H. Liu, Simonyan, and Yang, 2019; Pham et al., 2018). However, there is little research on how NAS algorithms perform on genomic sequences, because most of the algorithms are applied on image classification (Zoph, Vasudevan, et al., 2018; Real et al., 2019), object detection (Zoph, Vasudevan, et al., 2018) or semantic segmentation (L.-C. Chen et al., 2018).
Due to this lack of research, we investigate how state-of-the art NAS algorithms, such as DARTS, Progressive Differentiable Architecture Search (P-DARTS) and Bayesian Optimized Neural Architecture Search (BONAS) can be used to find high-performance deep learning architectures in the field of genomics. Amber, a recently published concurrent work, focused also on the application of NAS for genome data (Z. Zhang, Park, et al., 2020). Our provided framework has some advantages over Amber: unlike the search space of Amber, which only consists of Convolutional Neural Network (CNN) operations, our search space combines convolutional and recurrent layers. Popular genome models such as DanQ or NCNet also used hybrid models, which consist of convolutional layers together with recurrent layers. These hybrid models showed

superior performance compared to pure convolution neural network architectures (Quang and Xie, 2015; H. Zhang et al., 2019). We implement new NAS approaches with a new search space which includes CNN and Recurrent Neural Network (RNN) operations. We call these algorithms genomeDARTS, genomeP-DARTS, and genomeBONAS. Furthermore, we build novel DARTS algorithms such as CWP-DARTS, which enables continuous weight sharing across different P-DARTS stages by transferring the neural network weights and architecture weights between P-DARTS iterations. In another P-DARTS extension, we discard not only bad performing operations but also bad performing edges. Additionally, we implement an algorithm, which we call OSP-NAS. OSP-NAS starts with a super-network model which includes all randomly sampled operations and edges and then gradually discards randomly sampled operations based on the validation accuracy of the remaining super-network. Moreover, we benchmark all presented genomeNAS algorithms against state-of-the-art genome deep learning models, as well as against randomly searched models, using the DeepSEA data.

Structure. We first present the theoretical fundamentals of the methods used within this work. Then, an overview of the related work is provided, with a detailed formulation of the NAS algorithms, DARTS, P-DARTS and BONAS. Aside from that, our novel implementations genomeDARTS, genomeP-DARTS, genomeBONAS, genomeCWP-DARTS, genomeDEP-DARTS and genomeOSP-NAS are introduced. In the next chapter some important hyperparamters of the NAS framework are compared. Afterwards, the benchmark section provides a deep investigation of the results and compares genomeNAS algorithms with state-of-the-art baseline architectures.

2 Related Work

2.1 Deep learning

Deep Learning is a machine learning approach that uses neural networks to transform an input x to an output y (Goodfellow, Bengio, and Courville, 2016). The transformation $y = f(x, \theta)$ learns and adjusts the parameter values θ to reach the most accurate prediction. θ represents the weights of the deep learning model.

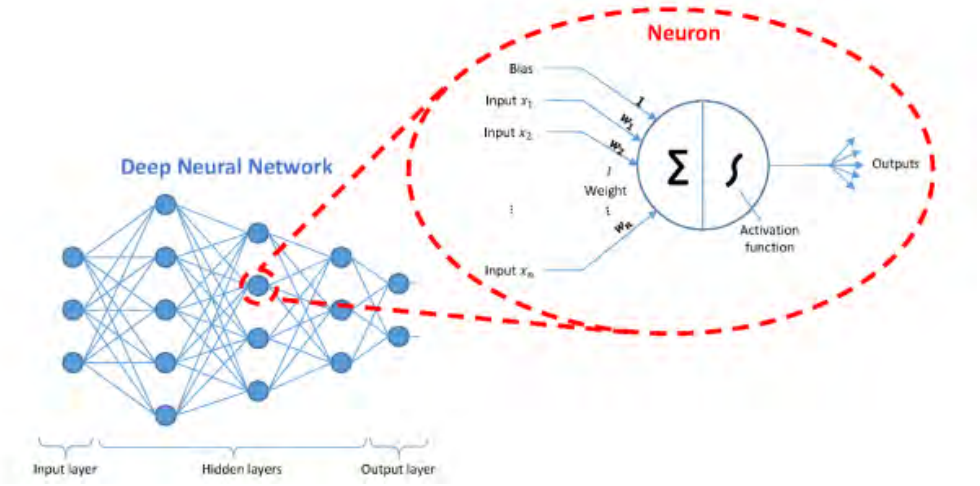


Figure 1: Fully connected neural network (from Sun, Wu, and Hwang, 2020). A typical fully connected neural network usually contains sequential layers and each of these layers consist of multiple neurons which are connected to the neurons from previous layers via weight matrices.

Figure 1 depicts a neural network, which starts with an input layer and is followed by some hidden layers and the final output layer which gives us the predictions of the input. A layer l transforms their input, by multiplying the output from the previous layer x_{l-1} with a weight matrix W_l and applying a nonlinear transformation $F(o_l)$ on the result with an activation function. More formally, each layer executes the following computation:

$$o_l = W_l x_{l-1} \tag{2.1}$$

$$x_l = F(o_l) \tag{2.2}$$

Doing this procedure for each layer results in a transformation from input features to output predictions, the so-called forward propagation (Lecun et al., 1998).

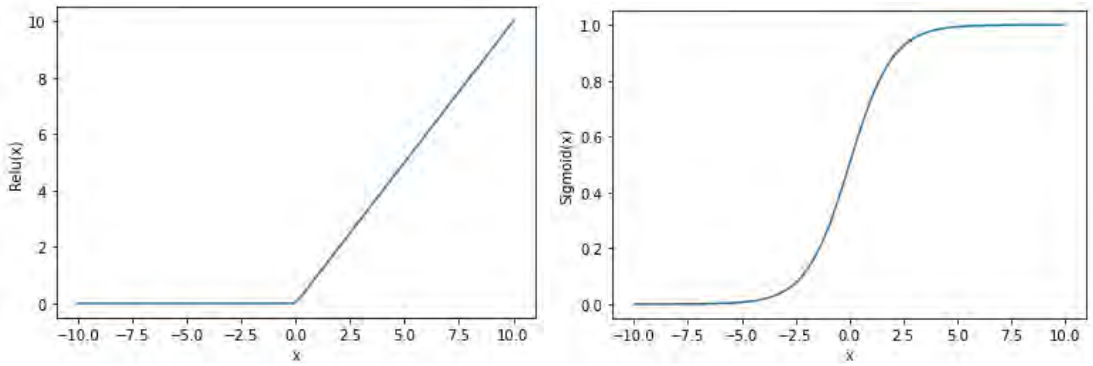
Activation functions Activation functions execute a nonlinear transformation on the input (Jian Zhou, 2015). The provided framework uses ReLU and sigmoid as activation functions, which are shortly introduced. The equation (2.3) uses a rectified linear function, which performs following computation:

$$ReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{2.3}$$

Another popular activation function is the sigmoid activation function. The sigmoid function squashes its inputs to values between 0 and 1 by applying the following function:

$$sigmoid(x) = \frac{1}{1+e^{-x}} \tag{2.4}$$

Since the output ranges between 0 and 1, the sigmoid activation function is especially used for the prediction of probabilities. Figure 2 depicts the sigmoid and ReLU activation function.



(a) ReLU activation (b) Sigmoid activation

Figure 2: Activation functions. The figure illustrates two activation functions, which are used within this work. The x-axes represent the input and the y-axes the transformed input.

Loss function A so-called loss function quantifies the ability of the model to generate accurate predictions (Z. Zhang and Sabuncu, 2018). A common loss function for multiclass classification is the cross-entropy loss, which is defined as follows:

$$CE = -\sum_{s}\sum_{t} y_t^s log(f_t(x^s, \theta)) \tag{2.5}$$

where s represents the index of a training sample and t represents the index of the target feature. $y_t^s \in \{0,1\}$ is the true label for a specific sample s, where the value 1 indicates that a specific target feature is true and 0 indicates that a specific target feature is false. For a given input x_s, $f_t(x^s, \theta) \in (0,1)$ with $\sum_t f_t(x^s, \theta) = 1 \ \forall s,t$ indicates the corresponding prediction of the deep learning model, which is the probability for a specific target feature t.
The binary cross-entropy loss is used by Jian Zhou (2015):

$$BCE = -\sum_{s} y^s log(f(x^s, \theta)) + (1 - y^s) log(1 - f(x^s, \theta)) \tag{2.6}$$

Back-propagation and Gradient Descent During back-propagation, the weights get adjusted via gradient descent, with the purpose to minimize the error terms of the output predictions (Lecun et al., 1998). The gradient descent algorithm iteratively uses the partial derivatives to modify the weights W. The partial derivatives of equation (2.5) and (2.6) are obtained by executing the chain rule:

$$\frac{\partial L_s}{\partial o_l} = F'(o_l)\frac{\partial L_s}{\partial x_l} \tag{2.7}$$

$$\frac{\partial L_s}{\partial W_l} = x_{l-1}\frac{\partial L_s}{\partial o_l} \tag{2.8}$$

$$\frac{\partial L_s}{\partial x_{l-1}} = W_l^T\frac{\partial L_s}{\partial o_l} \tag{2.9}$$

where L_s defines the loss of sample s. By executing the above equations for all layers $l = 1,2,...,L$, all partial derivatives of the loss function with respect to all the weights are obtained. The weights get then adjusted with the gradient descent formula:

$$W(p) = W(p-1) - \nu\frac{\partial L}{\partial W} \tag{2.10}$$

with p being the iteration index and ν being the learning rate. The learning rate ν determines the size of the training step.

Convolution network layers Convolutional neural networks are a subclass of deep neural networks, which are specially used for image or audio data (Goodfellow, Bengio, and Courville, 2016). Usually, the structure of a convolutional neural network consists of repetitive convolutional layers and pooling layers. While in fully connected layers all neurons are connected to each other and each neuron processes all neurons from the previous layer, in a convolutional operation each neuron only uses the information of a spatial receptive field of the previous layer.
A so-called kernel moves along the input, to extract features, which are then summarized in the created feature maps. The kernel moves along the input with a specific step size, the so-called stride. A convolution layer with ReLU activation function can be described by the following equation (Jian Zhou, 2015):

$$convolution(X)_{ik} = ReLU(\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} W_{mn}^k X_{i+m,n}) \tag{2.11}$$

where X defines the input, k the kernel index and i the index of the output. Each weight matrix W^k is a matrix of dimension $M \times N$, where M is the size of the receptive field which moves along the input and N defines the number of input channels. In the field of genomics, convolution filters aim to identify local, recurring patterns that are assumed to have a specific biological function.

Pooling layer A typical convolutional neural network uses a pooling layer after a convolution operation (Goodfellow, Bengio, and Courville, 2016). A popular choice might be a max pooling layer which computes the maximum value over a receptive field. The layer reduces the dimension, by removing unnecessary information. This leads to higher receptive fields for subsequent layers. Another approach would be average pooling, where the average of the receptive field is computed.

Recurrent network layers Another variant of neural networks is the so-called recurrent neural networks (RNNs), which processes input values $x_1, ..., x_T$ with $t = 1, ..., T$ being the time steps of a sequence (ibid.). Due to their ability to capture sequential dependencies in internal states, they are usually used for sequential data, such as time series or DNA sequences.

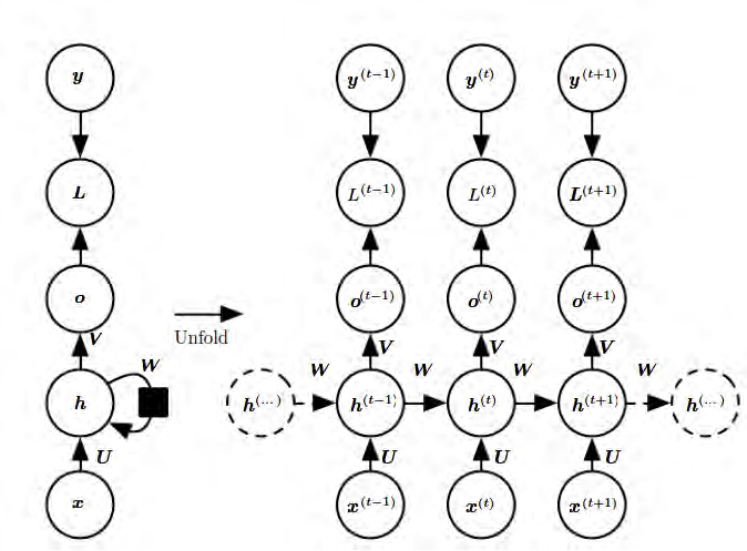


Figure 3: Forward propagation of an RNN (from Goodfellow, Bengio, and Courville, 2016). At time step t, hidden state h_t processes the output from previous hidden state h_{t-1} with weight matrix W and the current input x_t is processed with weight matrix U. Output o_t is calculated with the weight matrix V and hidden state h_t.

Figure 3 illustrates the forward propagation of an RNN that transforms an input sequence to an output sequence. More formally, at each time step following equations are executed:

$$h^{(t)} = tanh(b + Wh^{(t-1)} + Ux^{(t)}) \tag{2.12}$$

$$o^{(t)} = c + Vh^{(t)} \tag{2.13}$$

$$y^{(t)} = softmax(o^{(t)}) \tag{2.14}$$

with b and c being the bias vectors. The equations show that a hidden state $h^{(t)}$ contains information about all past time steps. This may lead to vanishing or exploding gradients

when the depth through time of recurrent neural network increases since long-term interactions lead to exponentially smaller weights and the multiplication of many Jacobians. Multiple multiplications of small values result in values close to 0. This leads to an inability to learn long-term dependencies. Long Short Term Memory (LSTM) addresses the exploding and vanishing gradient problem for recurrent networks by using gates, which memorize past information and control the gradient information which passes the gates. Instead of processing the previous state $h^{(t-1)}$ and input $x^{(t)}$ in a fixed manner, the gates control which information flows into the LSTM cell.

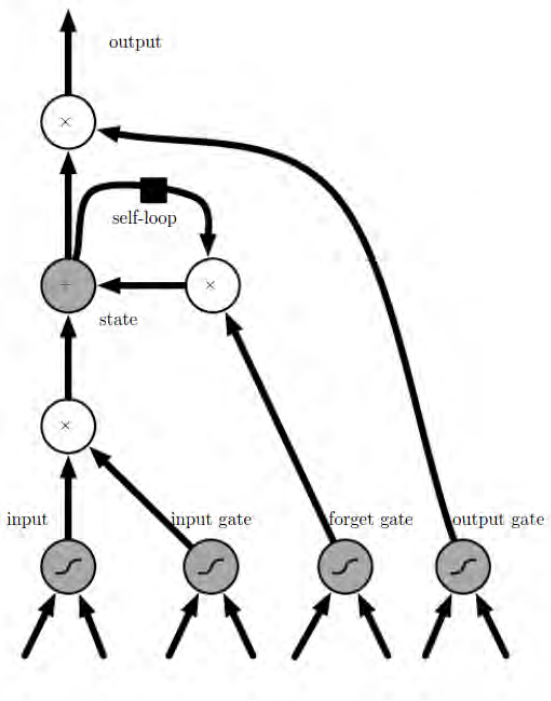


Figure 4: Visualization of an LSTM cell and its gates (from Goodfellow, Bengio, and Courville, 2016). In each LSTM cell an input gate controls the amount of input information entering the cell. The state unit $s_i(t)$ has a linear self-loop. The weight of the linear self-loop is controlled by the forget gate, allowing it to accumulate past information or to forget past information. The output gate controls the value which gets passed to the next cell.

Figure 4 shows a single LSTM cell. As in standard RNN, the cells are recurrently stacked together. In each LSTM cell, an input gate controls the amount of input information entering the cell.

2.2 Optimization

In our provided framework two optimization approaches are used - SMBO and differential optimization. Differentiable optimization is used for the inner optimization of neural network weights w, as well as for the outer optimization of the NAS algorithms. As the gradient descent method was already presented in section 2.1 in the backpropagation paragraph, this section gives only a brief introduction to the SMBO approach. In this work, SMBO is also called BO.

Sequential Model-Based Optimization Given a d-dimensional input domain $X = X_1 \times X_2 \times ... \times X_d$ and an output $y = f(x)$, SMBO searches input parameter x^*, which yields global maximum (or minimum): $x^* = argmax_{x \in X} f(x)$ (Bischl et al., 2018). Usually, Bayesian optimization uses so-called surrogate models to get cheap estimations of the black-box function f. A common choice for a surrogate model is the so-called Gaussian process (GP). Recently also random forests and deep feedforward neural networks have been widely used by researches. During the optimization procedure, new points are added through an acquisition function, which balances exploitation and exploration. In BO-based NAS, the algorithm proposes new architectures, which should be evaluated next. The selected architectures are then used to improve and update the surrogate $\hat{f}$.
The individual steps of the SMBO approach can be summarized as follows:

1. Sample n_{init} points $x^{(j)}$ $(j = 1, ..., n_{init})$ from X and evaluate these points on the black-box function to compute outputs $y^{(j)} = f(x^{(j)})$. Create an initial design with these points
2. Fit a surrogate model $\hat{f}$ on the tuples $(x^{(j)}, y^{(j)})$
3. The acquisition function selects m new points x^{j+k} $(k = 1, ..., m)$, which are assumed to be most promising for the optimization.
4. The selected points are evaluated using f and the new tuples $(x^{(j+1)}, y^{(j+1)})$ are added to the design
5. If the stopping criterion is not met, go to step 2
6. If the stopping criterion is met, return the proposed solution

2.3 NAS algorithms

In general, NAS algorithms attempt to automate the process of finding an appropriate deep learning model for a specific task (Elsken, J. H. Metzen, and Hutter, 2019). NAS remains a challenging task, since the search space can have billions of network architectures (Shi et al., 2020). Moreover, it is computationally expensive to obtain the performance of a specific architecture. Several recent approaches attempt to speed up NAS algorithms, such as weights or performance prediction for architectures (Baker et al., 2017; Brock et al., 2017) or weight sharing and inheritance among individual architectures (Elsken, J.-H. Metzen, and Hutter, 2017; Pham et al., 2018). There exist mainly two NAS approaches - sample-based algorithms and one-shot NAS algorithms (Shi et al., 2020).

Sample-based NAS Sample-based NAS approaches usually contain an optimization algorithm that proposes candidate architectures with a promising performance. The selected architectures are then evaluated to obtain their actual performance. The main advantage of sample-based approaches is good reproducibility. Furthermore, a sample-based approach may lead to a better exploration of candidate architectures, since the search space has usually fewer constraints. However, the main disadvantage is the need for heavy computation for training each candidate architecture from scratch.

One-shot NAS In one-shot NAS, all architectures are merged together, to build a large one-shot model architecture. The main advantage of one-shot NAS is, that it allows weight sharing among sub-architectures, which accelerates the training of those. The main disadvantage is the lack of reproducibility since the obtained results are sensitive to initialization. Moreover, one-shot NAS introduces constraints on the search space especially on the size of the one-shot model to fit in the memory. This leads to a smaller number of possible architectures.
To compare different NAS approaches, the provided genomeNAS framework uses sample-based and one-shot NAS. In this section the popular NAS algorithms DARTS, P-DARTS and BONAS are presented. While DARTS and P-DARTS use the one-shot method (H. Liu, Simonyan, and Yang, 2019; X. Chen et al., 2019), BONAS combines one-shot NAS with sample-based NAS to benefit from both approaches (Shi et al., 2020).

2.3.1 DARTS

The following section refers to H. Liu, Simonyan, and Yang (2019). An important source of computational inefficiency is that conventional NAS approaches optimize over a discrete search space, which leads to billions of possible architectures. Moreover, many sample-efficient optimization algorithms can not be used with a discrete search space such as BO with Gaussian processes or differentiable optimization. H. Liu, Simonyan, and Yang (ibid.) propose a continuous relaxation of the architecture representation, to transform a discrete search space into a continuous

search space. The continuous search space enables gradient-based optimization, which is presumed to be more data-efficient and less computationally demanding.

Search space Many researchers propose to search for repetitive building blocks to build the final architecture (Zoph, Vasudevan, et al., 2018; Real et al., 2019; C. Liu et al., 2018). These building blocks are computation cells, which are stacked together to form a convolutional network or recursively connected to form a recurrent network. Each cell is a directed acyclic graph and consists of N ordered nodes that are connected to each other via directed edges (i, j). A node $x^{(i)}$ defines a latent representation, such as feature maps in convolutional networks. The edges are specific operations $o^{(i,j)}$, which transform an input node $x^{(i)}$, such as a dilated convolution layer or a separable convolution layer in a convolutional network. As visualized in figure 5 the edges connect each node to all its previous nodes. An intermediate node processes all previous nodes by taking the sum over all previous nodes, which were transformed by a set of operations:

$$x^{(j)} = \sum_{i<j} o^{(i,j)}(x^{(i)}) \tag{2.15}$$

.

The output node concatenates all previous nodes along the channel dimension. The so-called directed acyclic graph (DAG), enables weight sharing among individual architectures, as multiple architectures are trained simultaneously.

The discrete choice of a certain operation is transformed into a continuous search space by applying a softmax over all possible operations:

$$\tilde{o}^{(i,j)} = \sum_{o \in O} \frac{exp(\alpha_o^{(i,j)})}{\sum_{o' \in O} exp(\alpha_{o'}^{(i,j)})} o(x) \tag{2.16}$$

where O defines a set of operations (e.g., separable convolutions, dilated convolutions, average pooling) and each operation defines some function $o(.)$ which processes $x^{(i)}$. The vector $\alpha^{(i,j)}$ parameterizes the operation mixing weights for a pair of nodes (i, j) and is of dimension $|O|$. $\frac{exp(\alpha_o^{(i,j)})}{\sum_{o' \in O} exp(\alpha_{o'}^{(i,j)})} o(x)$ determines the strength of an operation, and to get the final architecture, each mixed operation $\tilde{o}^{(i,j)}$ is replaced with the operation which yields the highest α value:

$$o^{(i,j)} = arg\,max_{o \in O} \alpha_o^{(i,j)} \tag{2.17}$$

.

A node in a convolutional cell retains the top-2 operations from the previous node, and for the recurrent cells, each node uses the top-1 operation.

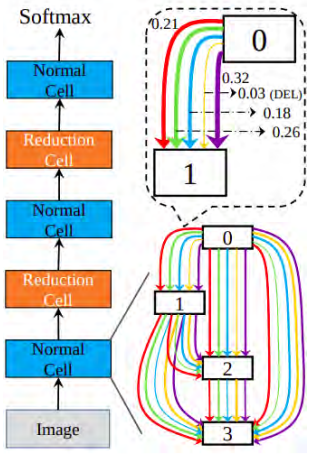


Figure 5: Visualitzation of DARTS (from X. Chen et al., 2019). The figure illustrates how a final architecture is built by stacking repetitive computation cells together. Each cell is a directed acyclic graph and consists of four ordered nodes that are connected to each other via directed edges (i,j). Each colored line defines a specific operation, which transforms an input node. As can be seen, each operation has a weight, which represents the operation strength.

Optimization As visualized in Figure 5 the task of the DARTS algorithm reduces to learning the encoding of the architecture. By optimizing the architecture encoding weights α and deep learning weights w jointly, DARTS aims to find architecture α^* that minimizes the validation loss $\mathscr{L}_{val}(w^*, \alpha^*)$ with corresponding weights w^*. The weights w^* are determined by minimizing the training loss $w^* = arg\,min_w \mathscr{L}_{train}(w, \alpha^*)$. This results in a bilevel optimization problem, which can be described as follows:

$$\min_{\alpha} \mathscr{L}_{val}(w^*(\alpha), \alpha) \tag{2.18}$$

$$s.t. \quad w^*(\alpha) = arg\,min_w \mathscr{L}_{train}(w, \alpha) \tag{2.19}$$

with α being the upper-level variable and w as the lower-level variable. Due to the expensive inner optimization, DARTS uses a simple approximation:

$$\nabla_\alpha \mathscr{L}_{val}(w^*(\alpha), \alpha) \approx \nabla_\alpha \mathscr{L}_{val}(w - \varepsilon \nabla_w \mathscr{L}_{train}(w, \alpha), \alpha) \tag{2.20}$$

Algorithm 1 describes the whole procedure of the DARTS algorithm, where the bilevel optimization is applied for a certain amount of epochs.

Algorithm 1 DARTS - Differentiable Architecture Search

1: create a mixed operation $\tilde{o}^{(i,j)}$ parametrized by $\alpha^{(i,j)}$ for each edge (i, j);
2: **for** `each epoch` **do**
3: Update architecture α by descending $\nabla\alpha \ \mathscr{L}_{val}(w - \varepsilon\nabla_w \mathscr{L}_{train}(w, \alpha), \alpha)$;
4: Update weights w by descending $\nabla w \ \mathscr{L}_{train}(w, \alpha)$;
5: **end for**
6: Derive the final architecture based on the learned α

2.3.2 P-DARTS

In this framework, P-DARTS is used, because the approach achieved significant improvements over the original DARTS algorithm (X. Chen et al., 2019). P-DARTS converges much faster while additionally achieving a better test error. While DARTS requires four GPU days for the training of the CIFAR10 dataset, P-DARTS only requires 0.3 GPU days. Furthermore, the test error improved from 2.76 percent to 2.5 percent.

As described in the previous section, the DARTS algorithm trains the whole one-shot model in the search phase for a certain amount of epochs. The one-shot model is also called super-network and consists of all operations and sub-architectures. Afterwards, the final architecture is derived by taking the k-best operations from the super-network and increasing the number of stacked cells (H. Liu, Simonyan, and Yang, 2019). According to X. Chen et al. (2019), this leads to the so-called depth gap because there is a large difference between the behaviour of a shallower one-shot model and the deeper final architecture. Normal cells of the final architectures tend to keep shallow connections because their errors decay faster compared to the errors of deeper connections. To address this problem, P-DARTS divides the DARTS procedure into multiple stages and in each stage the number of stacked cells is increased. Therefore, in each stage of the P-DARTS search, the super-network gets gradually closer to the final architecture and resolves the large gap between the search phase and the re-training of the final architecture. Figure 6 depicts the differences between DARTS and P-DARTS.

Training deeper networks causes longer runtime of the algorithms and a larger GPU memory usage. Moreover, the one-shot model is trained with all parameters, and therefore the parameters of the final architecture over-fits the one-shot model. This results in poor performance, especially when the architecture is transferred to a different task. Therefore, search space approximation is applied, where bad performing operations are gradually removed. As illustrated in figure 7 P-DARTS does not prune the super-network in a single step to obtain the sub-network, instead, P-DARTS gradually converge to the sub-network during the search process. While DARTS uses a constant operation space O, constant architecture encoding α, and constant L stacked cells, P-DARTS has operation space $O_k^{(i,j)}$, $\alpha_k^{(i,j)}$, and L_k for each stage Ψ_k. After each stage, the learned architecture $\alpha_{k-1}^{(i,j)}$ is used to rank the operations. Operations with lower weights in the previous stage are assigned with lower scores and are dropped. Based on the remaining

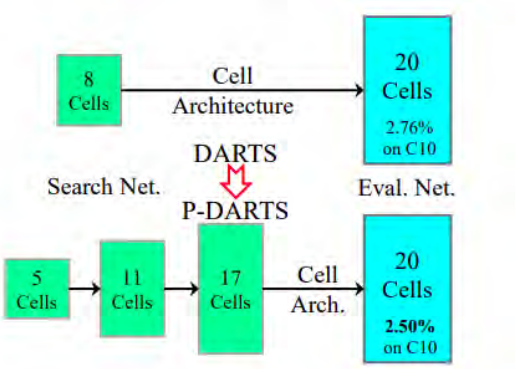


Figure 6: P-DARTS vs. DARTS (from X. Chen et al., 2019). While the original DARTS approach uses eight cells in the search phase, and 20 for evaluation of the final architecture, P-DARTS gradually increases the number of cells during the search phase.

operations, new operation space $O_k^{(i,j)}$ is build, i.e $O_k^{(i,j)} < O_{k-1}^{(i,j)}$. Additionally, the depth of the super-network is increased, i.e. $L_k > L_{k-1}$. As several operations are dropped in each stage, the super-network is trained from scratch and all network weights and architecture weights are initialized.

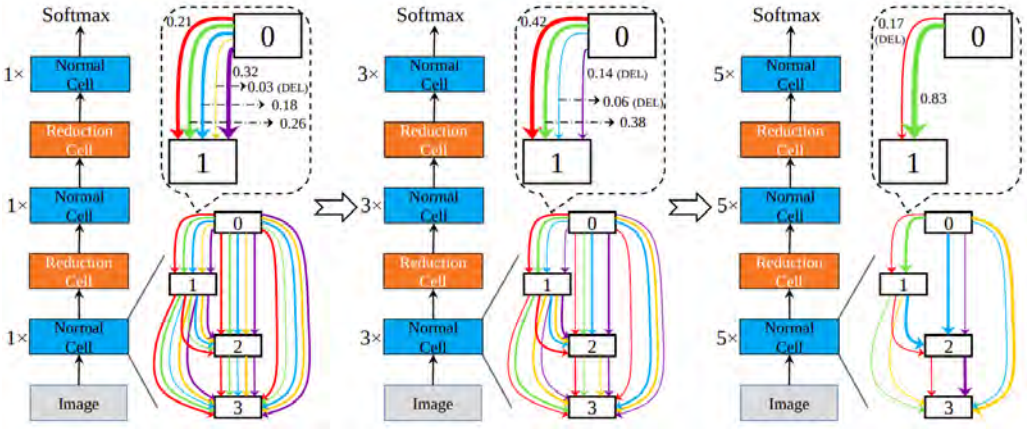


Figure 7: Visualitzation of P-DARTS (from X. Chen et al., 2019). DARTS search is split into multiple stages, and in each stage some operations with low α values are discarded. Additionally, the number of normal cells are increased in each stage.

Furthermore, training a deep one-shot model may lead to unstable gradients, and the super-network would be biased towards skip-connect operations because their corresponding errors decay faster than the error rates of convolutional layers. However, parametrized operations would have a stronger ability to learn representations when trained for longer epochs in the evaluation stage. Therefore a search space regularization is proposed, which uses operation-level dropout (Srivastava et al., 2014). Each skip-connect operation is enhanced by a dropout layer, where a specific number of neurons are set to 0. This leads to lower weights for skip-connect operations and ensures the better exploration of other operations, as the straightforward path of

skip-connection operation is disturbed. Each stage has a different dropout rate and after each epoch the dropout rate decreases.
When determining the final architecture, the appearance of skip-connect is further regularized by simply restricting the final architecture to only keep a certain amount M of skip-connect operations after the final stage.

2.3.3 BONAS

Main contribution of BONAS The main contribution of the BONAS algorithm is the combination of sample-based and one-shot NAS (Shi et al., 2020). The main advantage over other sample-based NAS algorithms such as BANANAS is that BONAS can train multiple related sub-architectures at the same time using weight-sharing. This accelerates the training of architectures. Standard one-shot approaches use weight sharing among the whole search space, including very different ones, which may be misleading. To solve this problem, BONAS performs the weight sharing approach only among similarly-performing sub-architectures. Figure 8 illustrates the BONAS procedure.

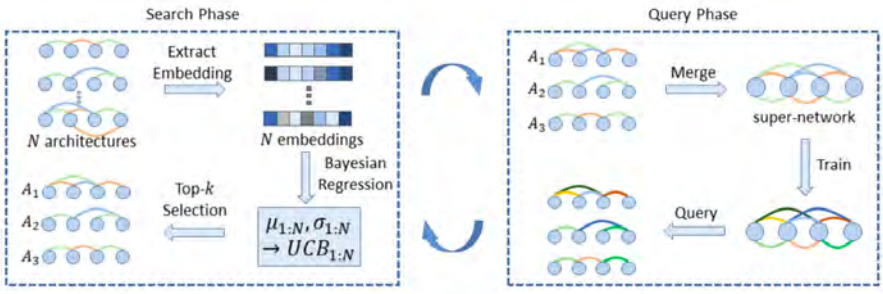


Figure 8: BONAS algorithm (from Shi et al., 2020). BONAS starts with a search phase, where N architectures are selected. Graph Convolutional Network (GCN) embedding extractor and Bayesian sigmoid regressor (BSR) are used as a surrogate model. The acquisition function selects the most promising architectures and merges them to a one-shot model, the so-called super-network. While training the super-network weights, the weights of all sub-networks are trained simultaneously by weight sharing. These trained weights are then used in the query phase to obtain the validation accuracy of the individual sub-networks.

Representation of architectures A main challenge of BO for NAS is the representation of the architectures. As shown in section 2.3.1 neural networks can be represented as directed acyclic graphs. An adjacency matrix is naturally able to encode the graph connectivity and is therefore used to represent the edges between the nodes of the search space (ibid.). The operations of each edge are represented as one-hot vectors, which are merged together to build the feature matrix X. Figure 9 illustrates how an adjacency matrix A and feature matrix X gets pre-processed by

connecting all nodes of the directed acyclic graph to an additional "global" node to enable the GCN to produce embeddings for the whole graph.

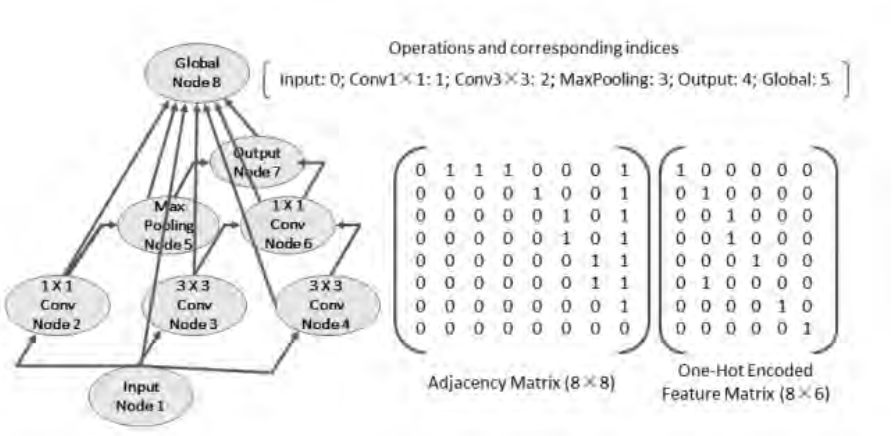


Figure 9: Representation of an example architecture (from Shi et al., 2020). On the left side of the figure, an example architecture with its nodes and connections is illustrated. The right side shows the corresponding adjacency matrix A and feature matrix X.

Surrogate model Using Gaussian process (GP) as a surrogate model is a popular choice for Bayesian optimization. However, graph convolutional networks are naturally able to preserve the structural information of graphs (Kipf and Welling, 2017) and are therefore a proper choice as a surrogate model. The surrogate model consists of several parts, including a GCN predictor and a BSR.

The GCN predictor aims to produce embeddings $\phi(A,X)$, which are then used by the BSR to determine the predictive mean and predictive variance. The learned embeddings $\phi(A,X)$ of the GCN are feed to a regressor, which computes the accuracy prediction. The regressor is a single-hidden-layer network. In order to ensure that the predictions lie in ranges between [0,1] a sigmoid function is used. Thus, the GCN predictor consists of multiple graph convolutional layers and a fully connected layer, which is followed by a sigmoid activation function. The GCN predictor is trained with the exponentially weighted loss:

$$L_{exp} = \frac{1}{N}\sum_{i=1}^{N}(exp(t_i)-1)(t_i-\tilde{t_i}) \tag{2.21}$$

with t_i being the ground truth and $\tilde{t_i}$ being the prediction of the GCN predictor (Shi et al., 2020).

The purpose of the BSR is to determine the mean and variance of the architecture's accuracy. Since the GCN predictor uses a sigmoid function to predict $\tilde{t}$, the target y of the bayesian sigmoid regressor is modeled as follows

$$y = logit(t) = log(\frac{t}{(1-t)}) \tag{2.22}$$

in order to use a bayesian linear regression model. In linear regression, model parameters w are used together with a feature matrix to compute a prediction. From a bayesian perspective, w is treated as a random variable with some distribution (Bishop, 2006). The prior is described as follows:

$$p(w) = \mathcal{N}(w|m_0, S_0) \tag{2.23}$$

with mean m_0 and covariance S_0. The posterior will also be Gaussian since a conjugate Gaussian prior distribution is used. The posterior distribution is then defined as:

$$p(w|t) = \mathcal{N}(w|m_N, S_N) \tag{2.24}$$

and following Bishop (ibid.), leads to the predictive mean μ of logit(t):

$$\mu(A, X; \mathcal{D}, \alpha, \beta) = m_N^T \phi(A, X) \tag{2.25}$$

with

$$m_N^T = \beta S_N \Phi^T y \tag{2.26}$$

$$S_N = (\alpha I + \beta \Phi^T \Phi)^{-1} \tag{2.27}$$

The precision parameters (α, β) are estimated by maximizing the marginal likelihood (Snoek and Adams, 2012), I is the identity matrix, and Φ defines the design matrix. The equation for the predictive variance of logit(t) is:

$$\sigma^2(A, X; \mathcal{D}, \alpha, \beta) = \phi(A, X)^T S_N \phi(A, X) + \frac{1}{\beta} \tag{2.28}$$

Afterwards, the predictive mean and variance of $logit(t)$ is converted to the predictive mean and variance of t, which results in the following equations (Shi et al., 2020):

$$\mathbb{E}[t] \simeq (\frac{\mu}{\sqrt{1 + \lambda^2 \sigma^2}}), \tag{2.29}$$

and

$$var[t] \simeq sigmoid(\frac{\alpha(\mu + \beta)}{\sqrt{1 + \lambda^2 \alpha^2 \sigma^2}}) - sigmoid(\frac{\mu}{\sqrt{1 + \lambda^2 \sigma^2}})^2 \tag{2.30}$$

Aquisition function An acquisition function selects new points by balancing the exploration and exploitation (Shahriari et al., 2016). Common acquisition functions are the maximum probability of improvement (MPI) (Kushner, 1964), expected improvement (EI) (Mockus, Tiesis, and Zilinskas, 1978), or upper confidence bound (UCB) (Srinivas et al., 2012). Shi et al. (2020) uses the UCB, whose exploitation exploration tradeoff is straightforward to implement. With Θ

being the hyperparameters of BO's surrogate model, and $\mathscr{D}$ being the observed data, the UCB for a new sample x is calculated by the following equation:

$$a_{UCB}(x;\mathscr{D},\Theta) = \mu(x;\mathscr{D},\Theta) + \gamma\sigma(x;\mathscr{D},\Theta) \tag{2.31}$$

$\mu(x;\mathscr{D},\Theta)$ represents the predictive mean of the output of the surrogate model which was defined in equation (2.25) and $\sigma^2(x;\mathscr{D},\Theta)$ defines the corresponding predictive variance defined in equation (2.28). $\gamma > 0$ is a hyperparameter, which balances the exploitation exploration tradeoff. Larger γ values result in higher exploration and vice versa. The final BONAS framework is depicted in algorithm 2.

Algorithm 2 BONAS

randomly select m_0 architectures $\mathscr{D}$ from search space $\mathscr{A}$ for weight-sharing training;
initialize GCN and BSR usng $\mathscr{D}$;
for `each BONAS iteration` **do**
    sample candidate pool $\mathscr{C}$ from $\mathscr{A}$ by EA;
    for `each candidate` m `in` $\mathscr{C}$ **do**
        embed m using GCN;
        compute mean and variance using BSR;
        compute UCB;
    end for
    $M \leftarrow$ candidates with the top-k scored;
    (query); train M with weight-sharing;
    add M and their performances to $\mathscr{D}$;
    update GCN and BSR with the enlarged $\mathscr{D}$;
end for

3 Method

3.1 Baseline models

In this chapter, a brief introduction to some popular state-of-the-art architectures for genomic sequence data is presented.

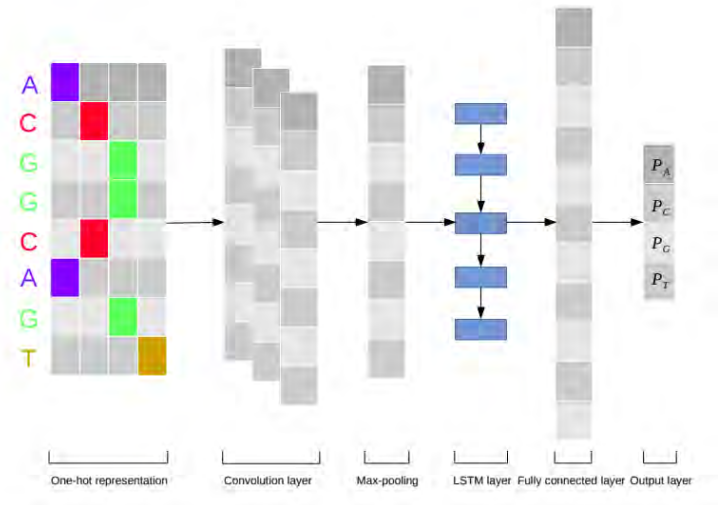


Figure 10: Typical deep learning model for genomic sequence data (from Wang et al., 2018). A typical DNA sequence consists of nucleobases adenine (A), cytosine (C), guanine (G), and thymine (T). The DNA sequences are usually one-hot encoded which leads to a sequence-length $\times$ 4 binary matrix. The figure shows how a kernel moves along an input matrix to create several feature maps. The feature maps are then fed to a recurrent layer. The recurrent layer is then followed by a fully connected layer and an output layer.

Figure 10 illustrates a typical deep learning model for genomic sequence data. In our framework, all architectures are designed for the DeepSEA task and attempt to predict the effects of non-coding variants on chromatin profiles, including transcription factor binding, DNA accessibility, and histone marks of sequences. The presented architectures are used within the provided genomeNAS framework and are compared with the architectures found by genomeNAS algorithms. All baseline models were re-implemented with the Pytorch library.

3.1.1 DeepSEA

The deep learning architecture of DeepSEA consists of three convolutional network layers with max pooling layers in between each convolution, followed by two fully connected layers (Jian Zhou, 2015). After each convolutional layer, a ReLU activation is applied. To regularize the model, the first two layers use a dropout rate of 0.2 and the third layer has a dropout rate of 0.5. Figure 11 illustrates the DeepSEA architecture and its hyperparameters.

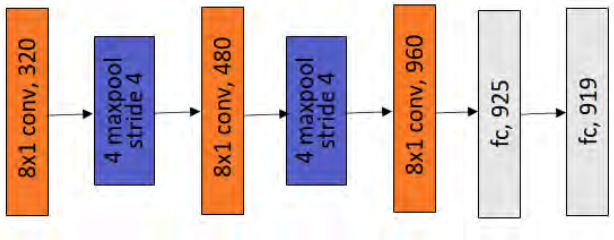


Figure 11: DeepSEA architecture (Jian Zhou, 2015). The architecture consists of three convolutional layers with a kernel size of eight. The first convolutional layer has channel size 320, the second 480 and the third has channel size 960. Between the convolutional layers, the max pooling layers downsample the sequences. The pooling layers have a kernel size four and stride four. The first fully connected layer has 925 neurons, and the last fully connected layer has 919 neurons.

3.1.2 DanQ

Quang and Xie (2015) presented a novel hybrid convolutional and bi-directional long short-term memory recurrent neural (BLSTM) network framework to predict non-coding functions from genomic sequences. The convolution layer aims to learn regulatory motifs, while the recurrent layer learns long-term dependencies between the motifs. The DanQ architecture consists of a single convolutional layer, followed by a max pooling layer, a BLSTM layer, and two fully connected layers. The output of the CNN part is processed by a forward and backward LSTM. The output is flattened and passed to a fully connected layer. After max pooling a dropout layer with dropout rate 0.2 is applied, and after the BLSTM layer, 50 percent of neurons are set to 0 to regularize the model. The fully connected layers and the convolutional layer are followed by a ReLU activation function. Figure 12 gives a detailed description of the DanQ architecture and its hyperparameters.

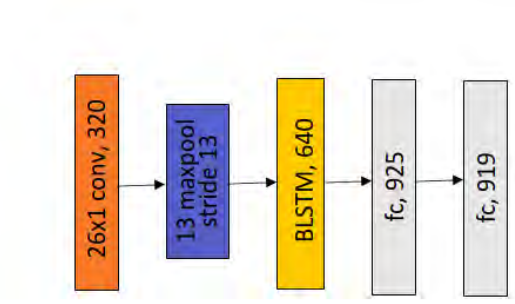


Figure 12: DanQ architecture (Quang and Xie, 2015). The single convolutional layer is of dimension 320 and has a kernel size of 26 with stride one. The max pooling layer has kernel size 13 and stride 13. Each directional LSTM has 320 channels, which results in 640 total channels. The first fully connected layers has 925 units and the second fully connected layer has 919 units.

3.1.3 NCNet

H. Zhang et al. (2019) propose several extensions for the convolutional network part of the DanQ model by using deep residual CNNs, that were already applied in many domains, such as image classification and object detections. The new convolution layers attempt to improve the learning of the local patterns from the sequences. These recurring patterns in DNA are presumed to have a biological function.

NCNet-RR Model (Residual Then Recurrent Network Model) Deeper networks tend to learn more local spatial information (ibid.). However, this comes with an increased risk of the so-called degradation (accuracy saturation) problem, where deeper models tend to have higher training errors (He et al., 2015). This problem can be overcome with an identity shortcut connection. Figure 13 illustrates a residual block. The residual block is used to enhance the convolutional part of the DanQ architecture by replacing the single convolutional layer with two of such blocks. Each residual block consists of a 1D convolution layer, a batch normalization layer, a ReLU activation layer, and then another 1D convolution layer with batch normalization layer.

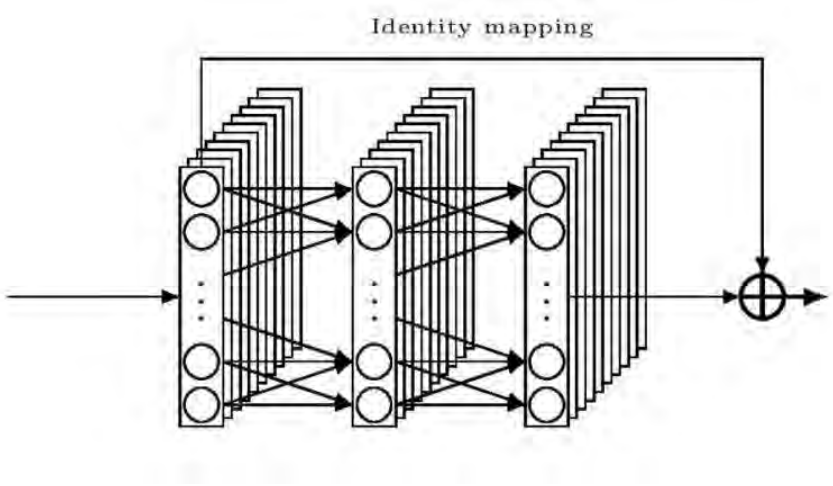


Figure 13: Skip-connection (from H. Zhang et al., 2019). The beginning of the block is directly linked to the end of the block with an identity mapping. Afterwards, both information flows are concatenated and flow to the next block.

NCNet-bRR Model (Bottleneck Residual then Recurrent Network Model) In order to use more layers while keeping the training time and number of weights low, H. Zhang et al. (2019) propose so-called "bottleneck" blocks. These blocks aim to cut down weights in order to use more layers. Figure 14 shows the bottleneck design, where the first and the last layers have a kernel of size one, in order to reduce and restore the channels for the middle convolution. H. Zhang et al. (ibid.) propose a deeper network with eight bottleneck residual blocks for the CNN part, which is connected to the recurrent part.

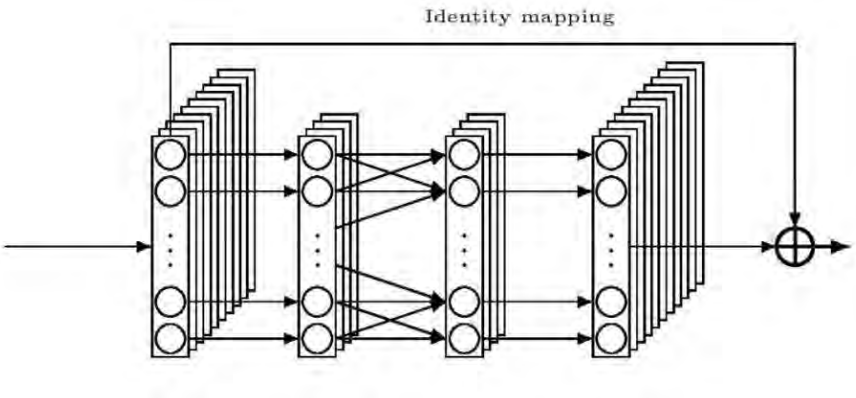


Figure 14: Bottleneck block (from H. Zhang et al., 2019). As can be seen, the middle layers have smaller channel sizes than the first and the last layer. The beginning of the block is directly connected to the end of the block with an identity mapping.

3.2 Search space for genomeNAS Algorithms

In research, NAS algorithms are usually applied on image classification tasks (H. Liu, Simonyan, and Yang, 2019; Pham et al., 2018), and therefore their search space only consists of convolutional layers. However, DARTS and ENAS also perform NAS on language modeling tasks in order to search for recurrent architectures (H. Liu, Simonyan, and Yang, 2019; Pham et al., 2018). So far, Amber is the only framework that also focuses on the application of NAS for genome data (Z. Zhang, Park, et al., 2020). However, Amber only uses a convolutional search space, although a lot of deep learning models for genomic sequence tasks consist of convolutional layers and recurrent layers (Quang and Xie, 2015; H. Zhang et al., 2019; Lanchantin et al., 2016; Wang et al., 2018). Therefore, we claim that our framework provides a few advantages over the concurrent work Amber. First, we implemented more genomeNAS algorithms in order to compare a wider range of NAS algorithms as it is still unclear how NAS can be applied in the field of genomics and how different NAS approaches perform with genomic sequence data. But the outstanding feature of our provided work is the unique combination of the convolutional DAG with recurrent DAG. We combine convolutional DAG with recurrent DAG because hybrid models showed superior performance compared to pure convolution neural network architectures (Quang and Xie, 2015; H. Zhang et al., 2019). Briefly, unlike the search space of Amber, which only consists of CNN operations, our search space combines convolutional and recurrent layers.

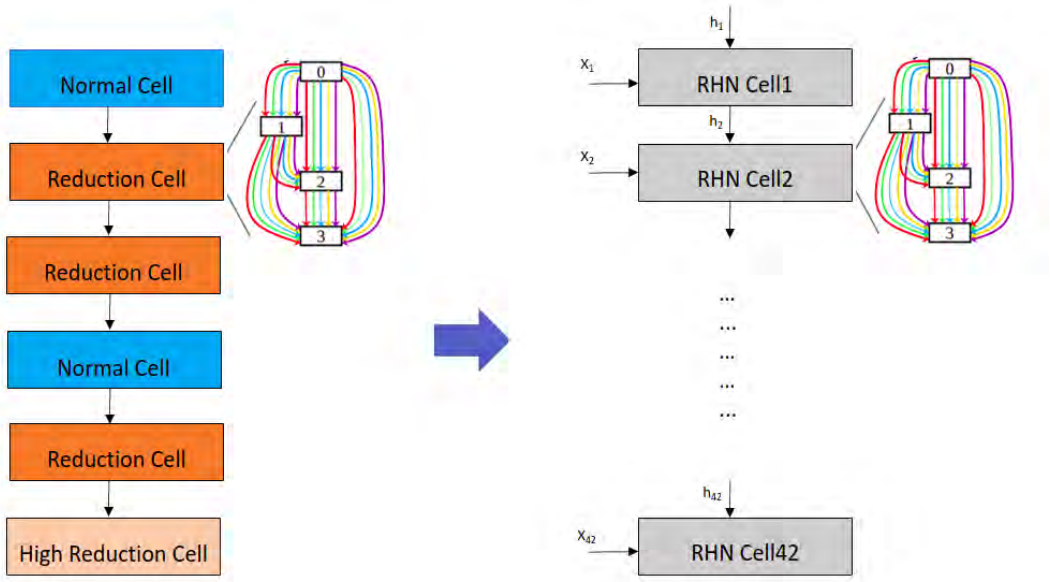


Figure 15: Search-space of genomeNAS algorithms. The input sequence of size 1000 is processed by the convolutional DAG, which consists of three reduction cells, one high reduction cell, and two normal cells. While the reduction cells downsample the input sequences by using a stride of size two, the "high reduction cell" further downsample the input by using a stride of size three. As can be seen, this results in a sequence of length 42, which is then processed by 42 recurrent cells.

3.2.1 Convolutional part

As explained in section 2.3, a promising approach in NAS is the one-shot model approach, where multiple sub-architectures are trained all together through weight sharing (H. Liu, Simonyan, and Yang, 2019; X. Chen et al., 2019; Pham et al., 2018). The one-shot model is usually defined through multiple cells that are stacked together (Zoph, Vasudevan, et al., 2018). For each convolutional cell, the cell outputs of the previous two layers are used as input. We follow H. Liu, Simonyan, and Yang (2019) and also use four nodes to represent a convolutional cell. While the original DARTS algorithm stacks eight convolutions cells together, P-DARTS gradually increases the number of stacked cells from five cells up to 17 cells (X. Chen et al., 2019). Both approaches only use two reduction cells, while the rest of the cells are so-called normal cells. Normal cells always use stride one, in order to keep the same dimension between the cells. Reduction cells on the other side, use stride two for the two cell inputs to halve the dimension. Using two reduction cells, as proposed by multiple NAS approaches, would result in halving the input dimension two times. Given an input sequence of length 1000 for the DeepSEA task, the output of the convolutional part would be a sequence of size 250 which is then fed to the recurrent cells. As illustrated in figure 15 each of the 250 time steps would be processed by a

recurrent DAG, which would lead to computational overhead. Moreover, popular deep learning architectures, such as the DanQ model, reduce the input sequence from length 1000 to 75, which is then fed to a LSTM layer (Quang and Xie, 2015). To avoid computational overhead and to stay close to state-of-the-art genomic deep learning architectures, our convolutional search space consists of four reduction cells and only two normal cells. Moreover, we introduce a so-called "high reduction cell", where a stride of size three is used to further downsample the input. As depicted in figure 15 this results in a sequence of length 42, which is then processed by the recurrent DAG, in order to learn dependencies between the features. In total, we only stack six cells together in the search phase and evaluation phase in order to avoid the so-called depth gap, which was explained in section 2.3.2 (X. Chen et al., 2019). Moreover, we attempt to stay comparable to genomic deep learning architectures such as NCNet in terms of model complexity.
As described in previous sections, the search space consists of N ordered nodes that are connected to each other via directed edges (i, j) (H. Liu, Simonyan, and Yang, 2019). The edges are specific operations $o^{(i,j)}$ which transforms an input node $x^{(i)}$. As genomic sequences are usually represented through one dimensional sequences, all convolutional layers are defined through 1D convolutional layers. We consider the same operations as in DARTS, namely zero, dilated convolutions, separable convolutions, skip-connections, max pooling, and average pooling. Both pooling operations have filter size five, while dilated convolutions and separable convolutions can have filter sizes nine or 15. Since most of the genomic deep learning models use normal convolutional layers, we add normal convolution operation to our operation space, which results in nine possible operations. The operation-space was chosen based on previous works (Jian Zhou, 2015; Quang and Xie, 2015; H. Zhang et al., 2019). The feature maps are padded to ensure that the output dimensions of all operations are the same and can be concatenated in each node. Each convolutional operation uses the ReLU-Conv-BN order, and each separable convolution and normal convolution is always repeated two times (H. Liu, Simonyan, and Yang, 2019). A convolutional cell includes two input nodes and four intermediate nodes which process the inputs. The last node is an output node, which concatenates all the intermediate nodes. This results in seven nodes for the convolutional part.
The convolutional network is then formed by stacking six convolutional cells together. The first and fourth cell are normal cells and the rest are reduction cells. Moreover, the last cell is a high reduction cell, because stride three is applied to the inputs of this cell. This results in a sequence of size 42, which is then processed by 42 recurrent cells.

3.2.2 Recurrent part

Recurrent Highway Networks (RHN) can be described as an extension of classical LSTM by allowing deeper step-to-step transitions (Zilly et al., 2017). While an LSTM cell has only a transition depth of one, RHN can be defined with deeper transitions.
In our NAS framework, each time step of an RHN cell is represented as a DAG with N nodes,

where N defines the transition depth. We followed the approach from H. Liu, Simonyan, and Yang (2019) and Pham et al. (2018) using a transition depth of $N = 9$. Following Pham et al. (2018), at time step t the computations for the first node are defined as follows:

$$c_1^{(t)} \leftarrow sigmoid(x^{(t)} \cdot W^{(x,c)} + h_N^{(t-1)} \cdot W_0^{(c)}) \tag{3.1}$$

$$h_1^{(t)} \leftarrow c_1^{(t)} \otimes f_1(x^{(t)} \cdot W^{(x,h)} + h_N^{(t-1)} \cdot W_1^{(h)}) + (1 - c_1^{(t)}) \otimes h_N^{(t-1)} \tag{3.2}$$

h_N^{t-1} describes the last node from the previous time step and $x^{(t)}$ the input at time step t. Thus, each recurrent cell processes the input at the current step and the state from the previous step (H. Liu, Simonyan, and Yang, 2019). The rest of the nodes $l = 2, 3, .., 9$ are specified as:

$$c_l^{(t)} \leftarrow sigmoid(h_{j_l}^{(t)} \cdot W_{l,j_l}^{(c)}) \tag{3.3}$$

$$h_l^{(t)} \leftarrow c_l^{(t)} \otimes f_l(h_{j_l}^{(t)} \cdot W_{l,j_l}^{(h)}) + (1 - c_l^{(t)}) \otimes h_{j_l}^{(t)} \tag{3.4}$$

with f_1 being an activation function, which the NAS algorithm searches for. In the framework provided by DARTS and ENAS the weight matrix w is shared among the recurrent cells and consists of $W^{(x,c)}$, $W^{(x,h)}$ and $W_{l,j}^{(c)}$ (Pham et al., 2018). The operation-space for the recurrent part follows previous works (Pham et al., 2018; Zoph and Le, 2017) and consist of tanh, relu, sigmoid and identity activations, as well as the zero operation. Each recurrent cell includes 12 nodes. The first two nodes are the inputs nodes and the first intermediate node processes the input nodes as described in equations (3.1) and (3.2). For the next eight intermediate nodes, f_l is learned and the output node of a cell is defined as the average over all intermediate nodes.

3.2.3 Further settings

Identical to DanQ, the output of the recurrent part is then flattened and processed by a fully connected layer with output size 925 (Quang and Xie, 2015). Another fully connected layer ensures that the output size corresponds to the number of classes and finally a sigmoid layer is applied to the output of the last layer. All genomeNAS algorithms start with an initial channel size of eight in the first convolutional cell, which gets gradually increased in each convolutional cell until channel size 512 is reached. The channel size of the recurrent highway networks is then set to 512. The numbers were chosen to ensure that our model size is comparable to state-of-the-art genomic deep learning models. While DanQ and NCNet have a channel size of 320, DeepSEA increases its channel dimension up to 960 channels (Quang and Xie, 2015; H. Zhang et al., 2019; Jian Zhou, 2015). As in DARTS, batches of size 64 are used during the search phase (training and validation sets). To ensure comparability between all genomeNAS algorithms, the batch size of all genomeNAS algorithms is set to 64.

3.3 NAS algorithms for Genomic Sequence Data

3.3.1 Random Search

To analyze if the superior performance of genomeNAS algorithms (over baseline-models) results from the well-designed search space, or the optimization, we also include random sampling in our framework. Therefore, we compare the genomeNAS algorithms against state-of-the-art deep learning models and also against randomly sampled architectures. We randomly sample several architectures from the search space described in section 3.2 and train them for a specific amount of epochs. The architecture that achieves the lowest validation loss defines the final architecture.

3.3.2 Hyperband-NAS

Moreover, we also include Hyperband-NAS in our framework. The Hyperband-NAS algorithm starts by randomly sampling a specific number of final architectures. Then, each architecture is trained for some epochs. The algorithm consists of several iterations. In each iteration, the number of architectures is halved by keeping only the best performing architectures. After halving the number of architectures, the remaining architectures are trained for some budget. These steps are repeated for a pre-defined number of iterations. This procedure is presumed to achieve good results, as we are directly optimizing with the final architectures instead of using a super-network as in the one-shot approach. Moreover, the optimization uses the F1-Score instead of validation loss, which may better handle the class imbalance.

3.3.3 genomeDARTS

Unlike the original DARTS algorithm, the operation-space of genomeDARTS now consists of $O = (O_{cr}, O_{cn}, O_r)$, with O_{cr} being the operation-space of the reduction and high reduction cells, O_{cn} being the operation-space of the normal cells, and O_r being the operation-space of the recurrent cells. We use the search space described in section 3.2, which results in a super-network that consists of six convolutional cells, and the output of the convolutional cells is then fed to the 42 recurrent cells. (O_{cr}, O_{cn}) includes nine convolutional operations, which were introduced in chapter 3.2.1 and O_r includes the five activation functions which were introduced in chapter 3.2.2. During the whole search process, the operation space $O = (O_{cr}, O_{cn}, O_r)$ is constant, which means that the operations of the super-network stay the same during the architecture search process. Moreover, architecture parameter α now includes $(\alpha_{cr}, \alpha_{cn}, \alpha_r)$. Zero initialization for architecture parameters α (for normal, reduction, and recurrent cells) ensures equal strength of all possible operations and leads to more exploration during the beginning of the search (H. Liu, Simonyan, and Yang, 2019). Other settings remain the same as the ones used for DARTS, where parameters w and architecture paramters α are trained jointly during architecture search. After training the super-network (including all operations) for a specific amount of epochs, the final architecture is obtained by choosing the operations and edges with high alpha values.

3.3.4 genomeP-DARTS

As in genomeDARTS, the operation-space O is defined through (O_{cr}, O_{cn}, O_r). Identical to the original P-DARTS algorithm, the search process of genomeP-DARTS consists of $K = 3$ stages (X. Chen et al., 2019). As we add one operation to the operation space of the convolutional part, the size of operation spaces $(O_{cr,0}^{(i,j)}, O_{cn,0}^{(i,j)})$ is set to be nine in the initial stage. After the initial stage three operations are removed from $(O_{cr,0}^{(i,j)}, O_{cn,0}^{(i,j)})$ and after the second stage another three operations are removed from $(O_{cr,1}^{(i,j)}, O_{cn,1}^{(i,j)})$. Unlike original P-DARTS, we additionally need to remove the operations from the recurrent highway network cells. In the first stage $O_{r,0}^{(i,j)}$ consists of five operations, which gets then decreased to three operations for $O_{r,1}$, and two operations for $O_{r,2}$. To ensure the same layer size during the search and the evaluation of the final architecture and to additionally reduce computational overhead, we use a constant layer size of six across all stages. Other settings are similar to P-DARTS, where dropout on skip-connect operations is applied. In each stage, the remaining one-shot model is trained for a specific amount of epochs. While in the first epochs, only network weights w_{k-1} are trained to ensure well-learned operations, in the last epochs of a stage the network weights w_{k-1} and architecture weights α_{k-1} are trained jointly as previously described. The genomeP-DARTS procedure can be described by algorithm 3.

Algorithm 3 GenomeP-DARTS

initialize w_0 and a mixed operation $\tilde{o_0}^{(i,j)}$ parametrized by $\alpha_0^{(i,j)}$ for each edge (i, j);
for `each stage` Ψ_k **do**
    for `each epoch of this stage` **do**
        update α_{k-1} by descending $\nabla\alpha_{k-1}\, \mathscr{L}_{val}(w_{k-1} - \varepsilon\nabla_{w_{k-1}}\mathscr{L}_{train}(w_{k-1}, \alpha_{k-1}), \alpha_{k-1})$;
        update weights w_{k-1} by descending $\nabla w_{k-1}\, \mathscr{L}_{train}(w_{k-1}, \alpha_{k-1})$;
    end for
    discard operations from $O_{k-1}^{(i,j)}$ based on the learned $\alpha_{k-1}^{(i,j)}$ to build new $O_k^{(i,j)}$
    initialize w_k and a mixed operation $\tilde{o_k}^{(i,j)}$ parametrized by $\alpha_k^{(i,j)}$, based on the new $O_k^{(i,j)}$;
end for
derive the final architecture based on the learned $\alpha_K^{(i,j)}$

3.3.5 genomeBONAS

We follow Shi et al. (2020) and reduce the operation-space (O_{cr}, O_{cn}) to four convolutional operations, namely identity, max pooling, separable convolutions, and dilated convolutions. While the kernel size for max pooling is five, separable and dilated convolutions have a kernel size 15. Having a smaller operation-space result in similar child models and ease training the child models jointly. For the recurrent part, we include tanh, relu, sigmoid activations, as well as the identity mapping and the zero operation as choices. As presented in section 2.3.3, BONAS

uses adjacency matrices and feature matrices to represent architectures. Since genomeBONAS now consists of a convolutional and a recurrent part, an architecture is now represented through a convolutional adjacency matrix A_c, a recurrent adjacency matrix A_r, a convolutional feature matrix X_c and a recurrent feature matrix X_r.

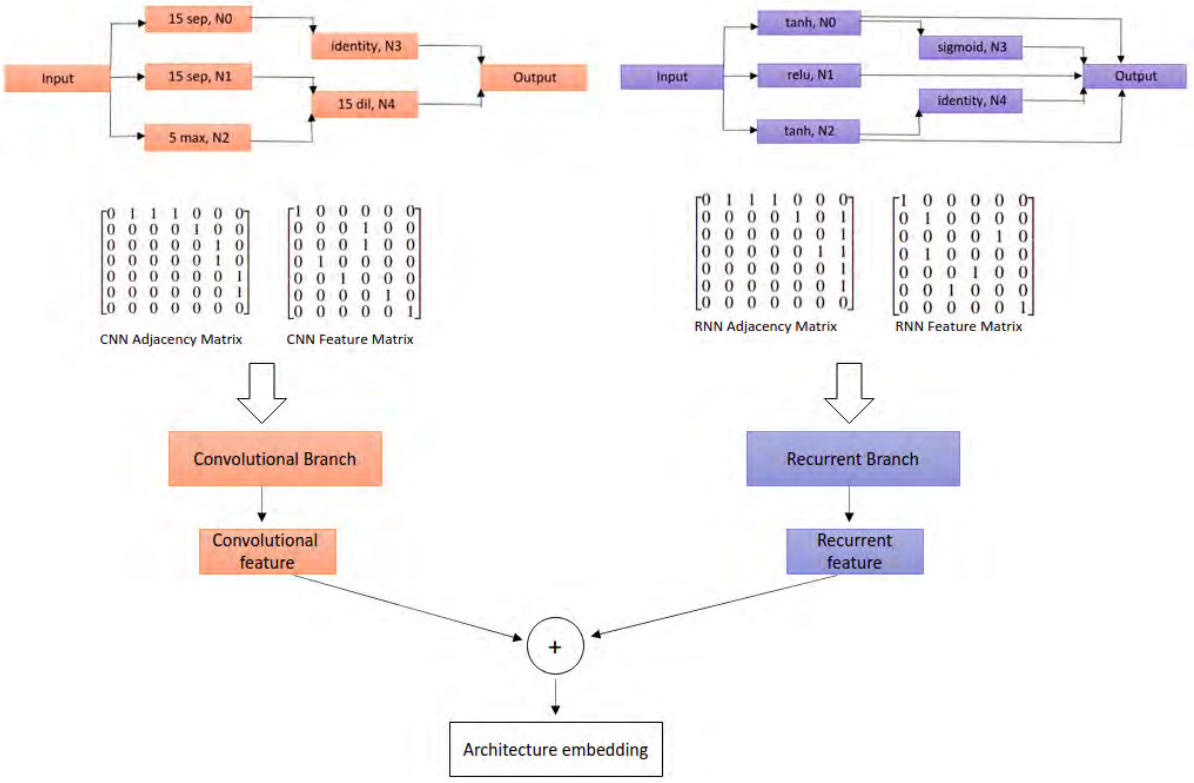


Figure 16: GenomeBONAS GCN. On the upper left part of the figure, the convolutional architecture is shown, and on the upper right side, the recurrent architecture is illustrated. Both architectures are encoded by an adjacency matrix and a feature matrix. While the first branch of genomBONAS GCN processes the CNN adjacency matrix and the CNN feature matrix, the second branch of the genomeBONAS GCN processes the RNN adjacency matrix and the RNN feature matrix. Both feature maps are then concatenated to build the architecture embedding.

Figure 16 illustrates how an architecture is represented through four matrices. genomeBONAS starts with an initial design $\mathscr{D}$ of m_0 randomly sampled architectures, which are merged together, to form a super-network $(\hat{A}_c, \hat{A}_r, \hat{X}_c, \hat{X}_r)$ with adjacency matrices $\hat{A}_c = A_{c,1}||A_{c,2}||...||A_{c,m_0}$ and $\hat{A}_r = A_{r,1}||A_{r,2}||...||A_{r,m_0}$ and feature matrices $\hat{X}_c = X_{c,1}||X_{c,2}||...||X_{c,m_0}$ and $\hat{X}_r = X_{r,1}||X_{r,2}||...||X_{r,m_0}$. Instead of training all architectures until convergence, we follow the BONAS approach and jointly train the architectures. In each step of an epoch, one architecture is randomly sampled from $\mathscr{D}$, and only the corresponding paths in the super-network are activated (Shi et al., 2020). After training the super-network for some epochs, each child model of the super-network is evaluated individually with the weights of the super-network. The GCN embedding extractor and BSR are then initialized and trained with $\mathscr{D}$ and the corresponding performance values t_i.

While BONAS uses a GCN with four graph convolutional layers to produce embeddings for an architecture, the GCN in genomeBONAS includes two branches and each of them consists of four graph convolutional layers. One branch processes the convolutional part and the other the recurrent part. Both parts are then concatenated to produce embeddings for the whole architecture. Unlike BONAS which uses accuracy as metric for its predictor, genomeBONAS uses the F1-Score to address the class imbalance of DeepSEA data.
In each genomeBONAS iteration, new architectures are generated based on local permutations of the current best architecture. Local permutations are applied by changing one value of the adjacency matrix or feature matrix. This ensures better exploitation of promising architectures and that the generated architectures are similar to each other, which eases the training of the super-network. A sampled architecture is then fed to the GCN to produce the embedding, which is used by the BSR to determine the predictive mean and variance of the predicted F1-Score. After that, the UCB scores for the architectures are calculated. We then select the top-k architectures with the best UCB scores to build new $\mathscr{D}$ and train them together by weight-sharing to obtain their actual performance t_i. The surrogate is then updated, using $\mathscr{D}$ and t_i.

3.4 Novel Neural Architecture Search algorithms for Genomic Sequence Data

3.4.1 genomeOSP-NAS

As explained in section 2.3.1, DARTS jointly optimizes architecture weights α and neural network weights w with gradient descent (H. Liu, Simonyan, and Yang, 2019). Various extensions of the original DARTS propose to gradually prune the search space of the super-network (Li et al., 2020; C. Liu et al., 2018; X. Chen et al., 2019). OSP-NAS is based on the same idea, as the super-network is also pruned. Unlike P-DARTS where the architecture weights α reflect the strength of an operation, we propose to measure the strength of an operation by its impact on the validation performance, when being removed from a super-network. Briefly, OSP-NAS iteratively prunes the super-network by removing operations with low operation strength in terms of its contribution to the super-network's validation loss. In each iteration, child models are built by removing an operation-set from the previous super-network. As the child models only differ in some operations from the super-network, the child models are called "supersubnetwork" in the following work.

Figure 17 illustrates an example iteration of the genomeOSP-NAS algorithm, where three different supersubnets are built from a super-network. These steps are executed multiple times until the final architecture is reached.

Algorithm 4 describes the OSP-NAS procedure in more detail. The algorithm starts with an initialized super-network S_0. To build S_0, we randomly sample operations from O_0 to form a super-network. At iteration 0, O_0 includes all convolution and recurrent operations and edges. S_0 is then trained for a specific amount of epochs, the so-called budget. Given a pre-trained super-network S_{k-1} with an operations space O_{k-1}, we randomly sample n operation-sets $o_l \in O_{k-1}$. Each operation-set includes one operation of the normal cell, one of the reduction cell, and one of the recurrent cell. These operation-sets are removed from S_{k-1} to form supersubnets $s_1, ..., s_n$. The super-network S_{k-1} is then updated by retaining the operation-set which was discarded in worst performing supersubnet s_l^*. Therefore, only the corresponding operation-set o_l^* is retained in the operation-space and all other operations-sets are discarded. The supersubnet stays close to the previous supersubnet since only some operation-sets are discarded. Therefore, the weights from the previous super-network can be used, which enables weight sharing across the OSP-NAS iterations. However, to recover from the perturbation of the super-network caused by removing the operation-sets, the updated S_k is trained for a specific budget. These steps are applied iteratively until the final architecture is reached. While the convolution part of the final architecture consists of four nodes and each of them has two edges, the recurrent part has eight nodes with one edge per node. The edges of the convolutional and the recurrent part consist of one operation per edge.

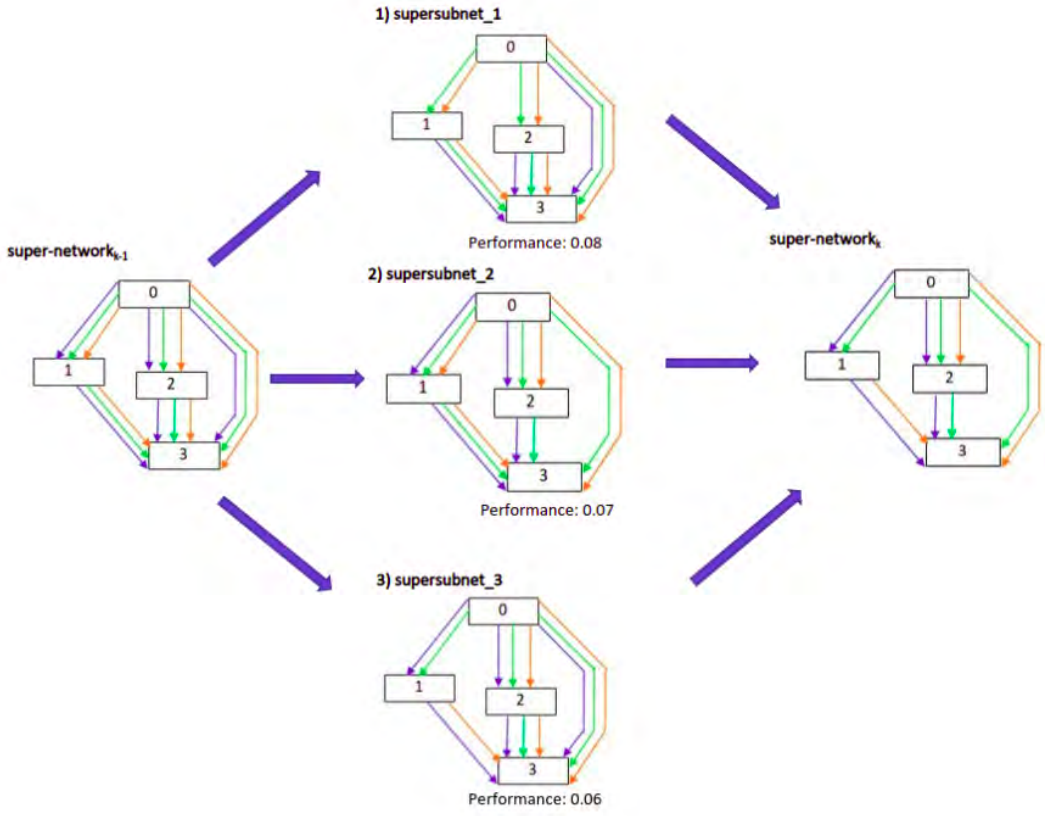


Figure 17: GenomeOSP-NAS example. Three supersubnetworks are build by removing three different operations-sets from the previous super-network. As supersubnet 1 achieves the worst validation performance, it is presumed to exclude important operations and therefore its operation-set is remained for the next iteration. The operation-sets from supersubnet 2 and 3 are discarded to build the new super-network.

Algorithm 4 OSP-NAS

sample m random operations from O_0 to build supernet S_0;
pretrain supernet S_0 with some budget (epochs);
while `not converged` **do**
    sample randomly n operation-sets from O_{k-1};
    remove these operation-sets from supernet S_{k-1} to build supersubnets $s_1, ..., s_n$;
    for `each candidate` l **do**
        validate the supersubnet s_l using weights from pretrained supernet S_{k-1};
    end for
    rank all candidates $s_1, ..., s_n$, based on the validation loss;
    keep the operation-set o_l^* of the worst performing supersubnet to build S_k;
    update O_k, by removing all other operation-sets;
    train supernet S_k for some budget (epochs) with weight sharing;
end while

The computation time of the algorithm is mainly affected by the budget, which is defined as the number of epochs. Also parameter n plays an important role, because a high n value, leads to a better exploration of the search space of the remaining super-network, as more operation-sets are evaluated. On the other side, this would also result in a high number of discarded operations per iteration, and the algorithm would also need a higher budget to recover.

3.4.2 genomeCWP-DARTS

To speed up the genomeP-DARTS algorithm, we designed Genome Continuous Weight Sharing Progressive Differentiable Architecture Search (genomeCWP-DARTS), where the weights of the inner optimization w_k as well as the architecture weights $\alpha_k^{(i,j)}$ are shared between stages. Identical to the already presented genomeP-DARTS algorithm, the search process of genomeCWP-DARTS consists of several stages. But instead of reinitializing the weights $(w_k, \alpha_k^{(i,j)})$ at each stage, the weights from the previous stage $(w_{k-1}, \alpha_{k-1}^{(i,j)})$ are transferred to the current stage. Same as in genomeP-DARTS, genomeCWP-DARTS starts with nine convolutional operations and five recurrent operations. To enable a fluent transition from one stage to the next stage, we use again $K = 3$ number of stages. By only discarding a few operations, the remaining super-network S_k stays close to the super-network S_{k-1} from the previous stage. This procedure is presumed to ease weight sharing across stages because the small perturbation on the super-network results in a small performance drop between stages. Therefore the algorithm also needs fewer epochs to recover from the perturbation of the super-network.
The number of operations, which are discarded in each stage, remains the same as in genomeP-DARTS. The new super-network S_k is then built with the remaining convolutional and recurrent operations and is initialized with the weights $(w_{k-1}, \alpha_{k-1}^{(i,j)})$ that matches the operations from $(O_{cr,k}, O_{cn,k}, O_{r,k})$. With the weight sharing approach, we attempt to reduce the number of epochs and to speed up the genomeP-DARTS algorithm. Other settings are similar to genomeP-DARTS.

3.4.3 genomeDEP-DARTS

To obtain the final architecture, we first have to determine the best operation from each edge (i, j), and then, the top edges are selected (H. Liu, Simonyan, and Yang, 2019; X. Chen et al., 2019; Shi et al., 2020; Pham et al., 2018). In a final architecture, each node of a convolutional cell keeps the top-2 edges, while the recurrent cells only retain the top-1 edge. As the recurrent cells include nine intermediate nodes, the last node always processes all eight previous nodes during architecture search. After retaining only the best edge of a node, the final architecture is only trained with one edge per node. Therefore we claim, that there is a large gap between the search and the evaluation, as already described in previous works (X. Chen et al., 2019). To overcome this issue, we propose a P-DARTS approach, where not only bad performing operations are discarded, but also bad performing edges.
In the following, we describe how an optimal edge is determined. As described in the previous

section, the best operation of an edge (i, j) is obtained by choosing the operation with the highest α value:

$$o^{(i,j)} = arg\,max_{o \in O} \alpha_o^{(i,j)} \tag{3.5}$$

Now, the maximum $\alpha^{(i,j)}$ value of each edge (i, j) can be determined with

$$\alpha^{*(i,j)} = arg\,max_{\alpha \in \alpha^{(i,j)}} \alpha_\alpha^{(i,j)}. \tag{3.6}$$

The best edge of a node i can be determined, by selecting $o^{*(i,j)}$, which has the highest $\alpha^{*(i,j)}$ value of a given node i:

$$o^{*(i,j)} = arg\,max_{o \in O} \alpha_o^{*(i,j)} \tag{3.7}$$

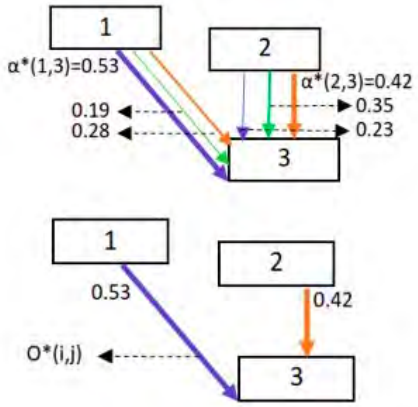


Figure 18: Process of discarding edges from a node. Node three receives two edges - one from node one and one from node two. In the first step, the best α value of each edge has to be determined, which is $\alpha^{*(1,3)} = 0.53$ and $\alpha^{*(2,3)} = 0.42$. In the second step, the operation with the highest $\alpha^{*(i,j)}$ is selected, which is the corresponding operation to $\alpha^{*(1,3)}$.

Figure 18 illustrates, how an edge from node three is discarded. After each stage, the Genome Discarding Edges Progressive Differentiable Architecture Search (genomeDEP-DARTS) procedure includes two steps. In the first step, the candidate operations of $O_{k-1}^{(i,j)}$ for each edge (i, j) are ranked based on the learned $\alpha_{k-1}^{(i,j)}$. $O_k^{(i,j)}$ is then built by retaining only the top-candidate operations. In the next step, all edges (i, j) of a node i are ranked according to the size of $\alpha^{*(i,j)}$ for an edge (i, j). After ranking all edges, the edges (i, j) with the lowest $\alpha^{*(i,j)}$ scores are removed from the super-network S_{k-1}. The detailed procedure of genomeDEP-DARTS is illustrated in algorithm 5.

Algorithm 5 GenomeDEP-DARTS

initialize w_0 and a mixed operation $\tilde{o_0}^{(i,j)}$ parametrized by $\alpha_0^{(i,j)}$ for each edge (i,j);
for `each stage` Ψ_k **do**
  for `each epoch of this stage` **do**
    update α_{k-1} by descending $\nabla\alpha_{k-1}\ \mathscr{L}_{val}(w_{k-1} - \varepsilon\nabla_{w_{k-1}}\mathscr{L}_{train}(w_{k-1},\alpha_{k-1}),\alpha_{k-1})$;
    update weights w_{k-1} by descending $\nabla w_{k-1}\ \mathscr{L}_{train}(w_{k-1},\alpha_{k-1})$;
  end for
  discard operations from $O_{k-1}^{(i,j)}$ based on the learned $\alpha_{k-1}^{(i,j)}$ to build new $O_k^{(i,j)}$
  determine $\alpha^{*(i,j)}$, which is highest α value of an edge (i,j)
  rank each edge (i,j) based on the magnitude of $\alpha^{*(i,j)}$
  discard bad performing edges (i,j) according to the ranks
  initialize w_k and a mixed operation $\tilde{o_k}^{(i,j)}$ parametrized by $\alpha_k^{(i,j)}$ for each edge;
end for
derive the final architecture based on the learned $\alpha_K^{(i,j)}$

Identical to P-DARTS and genomeP-DARTS, genomeDEP-DARTS includes $K = 3$ stages. After the first stage the top-6 candidate operations of the convolutional cells, and top-3 candidate operations of the recurrent cells are retained. Additionally, we limit each node of a convolutional cell to receive a maximum of four connections. As the convolutional cells have four nodes, the 4th and last node can receive an edge from the two previous convolutional cells as well as from the three previous nodes, which results in five connections in the initial stage. Forcing the nodes to have a maximum of four connections, results in discarding the worst-performing edge of the last node of the convolutional cells. For the recurrent cells, we choose a maximum size of five connections after the initial stage. Therefore the 8th node of a recurrent cell removes three edges, the 7th node two edges, and so on. After the second stage, we keep the top-3 candidate operations of the convolutional cells, and top-2 candidate operations of the recurrent cells to build a new operation-space O_k. To further decrease the number of edges, the upper limit for connections is set to be three for the convolutional cells and the recurrent cells. To obtain the final architecture after the last stage K, we follow the same procedure as all previous NAS algorithms by determining the best operation for each edge and then using the top-2 edges for the convolutional cell and the top-1 edge for the recurrent cell.

4 Experiments

4.1 Data and Application

The main functionalities of the genomic NAS algorithms are demonstrated with the DeepSEA genome task.

Non-coding DNA sequences play an important role in the human genome, as the majority of disease-associated variants lie in these sequences (H. Zhang et al., 2019). While 98.5 percent of the human genome consists of non-coding DNA sequences, most of them have an unknown function. Thus, the prediction of the function of non-coding DNA has gained increasing interest in the bioinformatics research community (H. Zhang et al., 2019; Quang and Xie, 2015; Jian Zhou, 2015; Z. Zhang, Park, et al., 2020). The DeepSEA task is a popular non-coding DNA sequence dataset. The labels of DeepSEA data were determined through high-throughput sequencing of public available ChIP-seq data and quantifies genome-wide molecular profiles, such as protein bindings or chemical modifications (Jian Zhou, 2015). DeepSEA, DanQ, and NCNet are popular deep learning architectures for the prediction of non-coding DNA. They provide a framework to predict the effects of non-coding variants on chromatin profiles, including transcription factor (TF) binding, DNA accessibility, and histone marks of sequences. Both - the DanQ and NCNet framework - use the same features and data provided by Jian Zhou (ibid.). Furthermore, Amber attempts to perform NAS on the DeepSEA data.

Our provided framework also uses the training, validation, and testing data sets from the DeepSEA website to investigate how NAS algorithms can be applied for the prediction of non-coding functions. The data can be downloaded from http://deepsea.princeton.edu/help/. The input is represented by a one-hot encoded 1000-bp DNA sequence, which leads to a 1000×4 binary matrix, where the columns correspond to A, G, C, and T. Each input sequence has a corresponding target which is a vector of length 919, where each element stands for a chromatin feature which is labeled as 1, if it is active, or 0 otherwise. The 919 chromatin features consist of 125 DNase features, 690 TF features, and 104 histone features. The DeepSEA task attempts to solve a multi-label classification task, as multiple chromatin features can be active at the same time. Each of them executes a binary classification for the corresponding target. The dataset is split into three non-overlapping datasets. The training data has 4,400,000 samples, the validation data 8,000 samples, and the testing data 455,024 samples. The genomeNAS algorithms use the training and validation set in the training phase. While the training data is used for the optimization of the neural network weights, the validation data is needed to update the alpha weights. The test data is only used for the evaluation of the final architectures.

4.2 Preliminary Study

DARTS and ENAS run experiments on image classification and language modeling tasks (H. Liu, Simonyan, and Yang, 2019; Pham et al., 2018). For image classification, an initial learning rate of 0.025 is used and for language modeling tasks an initial learning rate of 20 is used. Due to the high difference between the learning rate for the image classification tasks and the language modeling tasks, we decide to use different learning rates for the convolutional and recurrent part of the search space. Obviously, the learning rate of the recurrent highway network has to be larger than the learning rate of the convolutional part. As first experiments showed high sensitivity to the learning rate of the recurrent part, we decide to run a preliminary study, which is presented in this section. Moreover, DARTS applies variational dropout (Gal and Ghahramani, 2016) of 0.2 to word embeddings, 0.75 to the cell input, and 0.25 to all the hidden nodes (H. Liu, Simonyan, and Yang, 2019). As first experiments indicated poor performance with these settings, we also include different variational dropout rates for the cell input and the hidden nodes in the preliminary study section. GenomeDARTS is used as NAS algorithm to compare the different configurations. In order to reduce computational runtime, genomeDARTS is only run for 10 epochs. The batches are of size 64 and the rest of the settings are the same as described in section 2.3. For each configuration, genomeDARTS is run four times, which results in four final architectures per configuration. We then run each of these final architectures for 20 epochs and report its validation performance. As our genomeNAS algorithms make decisions based on validation loss (DARTS algorithms) and F1-Score (genomeBONAS), we report the results for both metrics. Other settings are the same as described in section 4.3.1.

Figure 19 summarizes the different learning rates and variational dropout rates for the recurrent cells. The figure outlines that there is a big impact of the chosen initial learning rate on the final result for the DeepSEA task. Learning rate two leads to a bad validation loss (and F1-Score) for all three dropout rates, which indicates poor performance of the learning rate two. But one can observe that there is no big difference between the configurations of learning rate eight and learning rate 12, as the medians for the configurations of both learning rates are close to each other. However, it can be observed, that using a dropout rate of 0.1 for the cell input and 0.05 for hidden nodes shows the best results for all three learning rates (blue boxes). Unsurprisingly, configurations with a low validation loss tend to yield a high F1-score. As the median of the configuration with learning rate eight and variational dropout 0.1 for the cell input and 0.05 for the hidden nodes yield the lowest validation loss, as well as the highest F1-Score, we decided to use this hyperparameter configuration in this work.

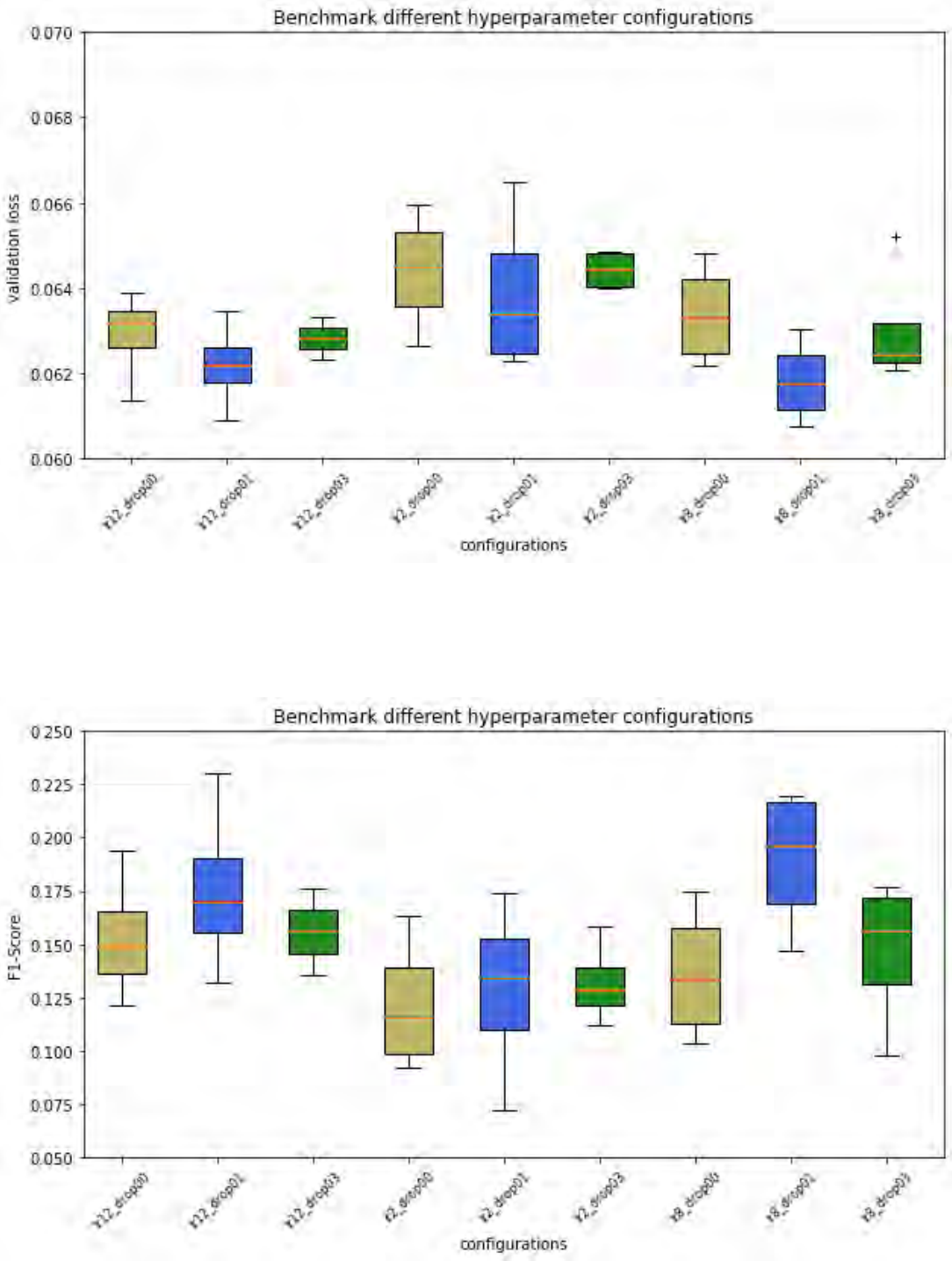


Figure 19: Benchmark of different hyperparameter configurations. Brown boxes are configurations with zero dropout, blue boxes are always with variational dropout rate 0.1 for cell input and 0.05 for hidden node and green boxes show the result of variational dropout rate 0.3 for cell input and 0.1 for hidden nodes. While the figure on top shows the validation loss of the configurations, the figure on the bottom shows the F1-Score of the configurations.

4.3 Benchmark NAS Algorithms on DeepSEA task

4.3.1 Experimental Design

We follow the NAS procedure suggested by H. Liu, Simonyan, and Yang (2019), where the NAS procedure is defined through three steps, namely architecture search, architecture selection, and architecture evaluation. In architecture search, we aim to learn the optimal architecture (encoding α in DARTS approaches) by training the genomeNAS algorithms for a pre-defined number of epochs. After the search phase is completed, we obtain the final architecture. Each model is then run for a defined amount of replications on the test set and the results are averaged. The genomeNAS procedure can be explained with the following steps:

1. Search phase: Run each NAS algorithm four times to obtain four different final architectures.
2. Selection phase: Run each of those four final architectures for 50 epochs and report the validation performance of each algorithm. The architecture that achieves the highest F1-Score on the validation data is chosen as the final architecture of the NAS algorithm.
3. Evaluation phase: To obtain the final performance of the final architecture of a NAS algorithm, we train it from scratch for 50 epochs and report its F1-Score on the test data. It is important to note that the test set is only used for the evaluation phase and not in the search phase or the selection phase. Since our results show some variance, even with the same setting, we do four replications of the final architecture run and average over the obtained results.

genomeDARTS As explained in the previous section, we use two different learning rates for the inner optimization of the weights w. In the convolutional part, the inner optimization of the weights is done with momentum SGD and an initial learning rate $\nu_w = 0.025$, which is annealed down to zero based on a cosine schedule without restart (Loshchilov and Hutter, 2017). We use an SGD optimizer to ensure that our optimizer is similar to DARTS. Momentum is set to 0.9, and weight decay to 3×10^{-4} (H. Liu, Simonyan, and Yang, 2019). For the recurrent part we use the same SGD optimizer but without momentum, and weight decay 5×10^{-7}. Based on the preliminary study results, the initial learning rate of the recurrent part is set to eight. Except the initial learning rate of the recurrent part and the fact that our SGD optimizer consists of two parts, the hyperparameters are identical to those in DARTS.
Also for the optimization of the architecture parameters α the settings remain the same as the ones used for DARTS. Similarly to DARTS, the architecture variables α are optimized with Adam (Kingma and Ba, 2017) with an initial learning rate of $\nu_\alpha = 3 \times 10^{-3}$, weight decay 10^{-3} and momentum $\beta = (0.9, 0.999)$. The stability of the network training can be ensured through a gradient clipping of size 0.25. Through this specification the gradients are not able to leave a pre-defined range. As in DARTS, the search is running for 50 epochs.

Random search and Hyperband For the Random search, 20 architectures are randomly sampled and trained for 7 epochs. This procedure ensures a similar runtime to other genomeNAS algorithms. In our experiments, the Hyperband-NAS algorithm consists of four iterations. In the first iteration, 20 architectures are randomly sampled and trained for three epochs. In the second iteration, the remaining 12 architectures continue their training with epochs three to six. As described in section 3.2, the final architectures always include three reduction cells, two normal cells, one high reduction cell, and 42 recurrent cells. The convolutional cells consist of four intermediate nodes, where each node receives two connections with one operation. The recurrent cells are defined through eight intermediate nodes, and each node receives one connection. The connections and the corresponding operations or activation functions are then randomly sampled. Other settings remain the same as for the other search algorithms, where a final channel-size of size 512, and a batch size of 64 are used. The hyperparameters for the inner optimization are the same as already described in the genomeDARTS paragraph.

genomeP-DARTS Most of the settings are similar to P-DARTS (X. Chen et al., 2019), where the initial dropout probability on skip-connect is set to 0.1 for the first stage, 0.2 for the second stage, and 0.3 for the third and last stage. In each stage, the one-shot model is trained for 25 epochs. To ensure well-learned inner weights, in the first 10 epochs, only network weights are trained. In the last 15 epochs of a stage, the network weights and architecture weights α are trained jointly. As in P-DARTS, we use Adam optimizer with learning rate $\nu = 0.0006$, weight decay 0.001 and momentum $\beta = (0.5, 0.999)$ for optimizing architecture parameters α. The optimizer for network parameters w_k is the same as for genomeDARTS.

genomeCWP-DARTS The initial stage of the genomeCWP-DARTS is the same as of P-DARTS, where the super-network is trained for 25 epochs, and in the first 10 epochs, only the super-network weights are trained. In stage two and three, the super-network is trained for 15 epochs and in the first three epochs, only the super-network weights are updated. Therefore the number of epochs is reduced from 75 to 55. The hyperparameters for the inner and outer optimization are the same as in genomeP-DARTS.

genomeDEP-DARTS As in P-DARTS, the super-network S_k is always trained for 25 epochs. While in the first 10 epochs only network weights w_k are trained, in epoch 10 to 25 the network weights w_k and architecture weights α_k are trained in a joint fashion. The rest of the settings, such as the learning rates for the inner optimization as well as the learning rate for updating the architecture parameters are the same as already used for genomeP-DARTS and genomeCWP-DARTS.

genomeOSP-NAS GenomeOSP-NAS starts with 108 randomly sampled operations for the normal cells, reduction cells and recurrent cells to build the initial super-network. The super-

network is then pre-trained for 10 epochs to ensure well-learned weights before discarding the first operations. The number of sampled operations n is set to seven for each iteration. As we only retain the best performing supersubnet, six operations are discarded in each iteration. To recover from the perturbation of the super-network, we use a budget of size five, which means that the remaining super-network is trained for five epochs in each iteration. The inner optimization of network weights w follows the same procedure as already described in the genomeDARTS paragraph.

genomeBONAS GenomeBONAS starts with an initial design by randomly sampling 60 architectures, which are trained for 60 epochs. Afterwards, the genomeBONAS procedure described in section 3.3.5 is repeated for two additional iterations. In each iteration, 1000 candidate architectures are sampled and the candidates with top-60 UCB scores are trained for 60 epochs. We follow the BONAS method and use a batch size of 128 to update neural network weights. The hyperparameters for the inner optimization of neural network weights are the same as previously described.

Selection and evaluation phase For the evaluation of the final architectures, we use batch size 100, which is the same as already used in DanQ, DeepSEA and NCNet (Quang and Xie, 2015; Jian Zhou, 2015; H. Zhang et al., 2019). The final architectures of the NAS algorithms include six convolutional cells and 42 recurrent cells. The channel-size and the settings for the optimization of w remain the same as previously described. The convolutional cells consist of four intermediate nodes, and each of them has two connections. The recurrent cells include eight intermediate nodes, and each of the nodes has one connection.

4.3.2 Experimental Results

In this section, we study the search efficiency of the proposed genomeNAS algorithms and compare them against state-of-the-art baseline models. To compare the deep learning models different performance metrics, such as the F1-Score, the macro-averaged PR-AUC score, and the macro-averaged ROC-AUC score are used. We use the same settings as previously described, where the final architectures are obtained by running each NAS algorithm four times and only using the best performing architecture. Then, the best architecture is running four times.

Selection phase First experiments indicated big differences between different NAS runs. Some NAS runs result in a good performing final architecture and some do not. As we are interested in genomeNAS algorithms which provide stable results and good performing architectures across different runs, the results of all genomeNAS runs will be compared. Instead of only using the best performing architecture, which was chosen in the selection stage as described in section 4.3.1, the boxplots now summarize the results of all four genomeNAS runs from the selection phase.

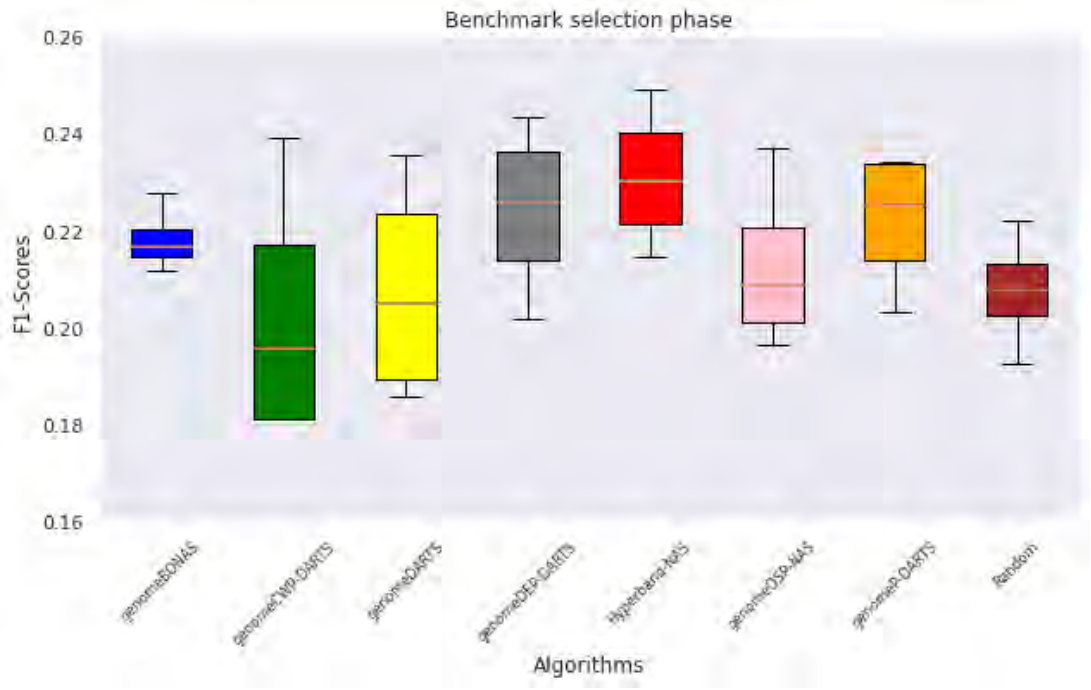


Figure 20: Benchmark of the selection phase. A box summarizes all four runs of the corresponding genomeNAS algorithm by training the final architecture of each run for 50 epochs and reporting its F1-Score on the validation data.

Figure 20 depicts the performance of the genomeNAS algorithms. It can be seen, that Hyperband and genomeDEP-DARTS provide the best results, as the boxplots have the highest medians and the highest upper whiskers. The upper whisker is important because only the best performing architecture is used for the evaluation phase. Moreover, figure 20 illustrates that the Hyperband search yields strong performing architectures across various runs because the algorithm achieves the highest median, the highest lower whisker, and the highest upper whisker. The median of the genomeBONAS algorithm is higher than the medians of genomeCWP-DARTS, genomeDARTS, genomeOSP-NAS, and Random search. In addition, genomeBONAS also has the lowest variance, which results in equally well performing architectures. Nevertheless, despite Random search, genomeBONAS has the lowest upper whisker, hence the algorithm is not able to find a very good performing final architecture for the evaluation phase. The corresponding boxes of the genomeCWP-DARTS and genomeDARTS algorithms show the highest variance, which indicates that both algorithms yield more inconsistent final results.

Evaluation phase In the following paragraph, the results of the evaluation phase are summarized. Table 1 and figure 21 illustrate the results of the final architectures, which were found by NAS algorithms. The optimization approaches of the genomeNAS algorithms seem to work well because except genomeBONAS, all genomeNAS algorithms perform better than the baseline Random search. Moreover, except genomeBONAS, all genomeNAS algorithms also consistently outperform the baseline algorithms. Even Random search achieves an average PR-AUC score

of 21.90 and an average ROC-AUC score of 84.56, which is comparable to the results from NCNet-RR and NCNet-bRR and better than the results of the DeepSEA and the DanQ algorithm. This indicates a strong performance of our unique designed search space, where a convolutional DAG is combined with a recurrent DAG.

Table 1: #params defines the number of parameters of a final architecture or a baseline model. The GPU days column depicts the average amount of GPU days of the genomeNAS search process, averaged over all four runs of the genomeNAS algorithm. For baseline models, there is no architecture search process, and the number of GPU days is marked "-"

Algorithms	#params	PR-AUC	ROC-AUC	GPU days
DeepSEA	52.84 M	6.44	71.22	-
DanQ	46.93 M	17.27	81.5	-
NCNet-RR	57.58 M	22.15	84.01	-
NCNet-bRR	47.69 M	22.05	84.31	-
Random search	27.01 M	21.90	84.56	10.22
Hyperband-NAS	26.86 M	23.44	85.23	9.91
genomeDARTS	29.22 M	23.90	85.47	10.21
genomeP-DARTS	27.08 M	22.24	84.68	11.61
genomeBONAS	26.25 M	21.20	84.19	24.03
genomeOSP-NAS	26.91 M	23.40	85.28	9.93
genomeCWP-DARTS	26.54 M	22.97	84.98	8.62
genomeDEP-DARTS	26.73 M	22.87	84.95	10.45

Hyperband-NAS, genomeDARTS, and genomeOSP-NAS achieve the best results. Hyperband-NAS has an average PR-AUC score of 23.44 and an average ROC-AUC score of 85.23, genomeDARTS has an average PR-AUC score of 23.90 and an average ROC-AUC score of 85.47, and genomeOSP-NAS has an average PR-AUC score of 23.40 and an average ROC-AUC score of 85.28. The performance of these three NAS algorithms are mostly comparable, perhaps with a slight advantage for the genomeDARTS due to the slightly higher PR-AUC and ROC-AUC score. The own designed algorithm genomeDEP-DARTS performs better than the original genomeP-DARTS, which indicates that we achieved to further decrease the gap between search and evaluation (X. Chen et al., 2019).

Hyperband-NAS shows strong performance because the algorithm has some advantages over the NAS algorithms. We assume that the good performance of Hyperband results from directly using the final architecture performance. The DARTS algorithms (DARTS, P-DARTS, CWP-DARTS and DEP-DARTS) for example use a one-shot model, where all sub-architectures are trained and evaluated in a joint fashion to learn the operation strength α (H. Liu, Simonyan, and Yang, 2019). As the learned α values only reflect the operation strength of an operation of the super-network, it does not directly quantify the performance of a sub-architecture. With an average PR-AUC score

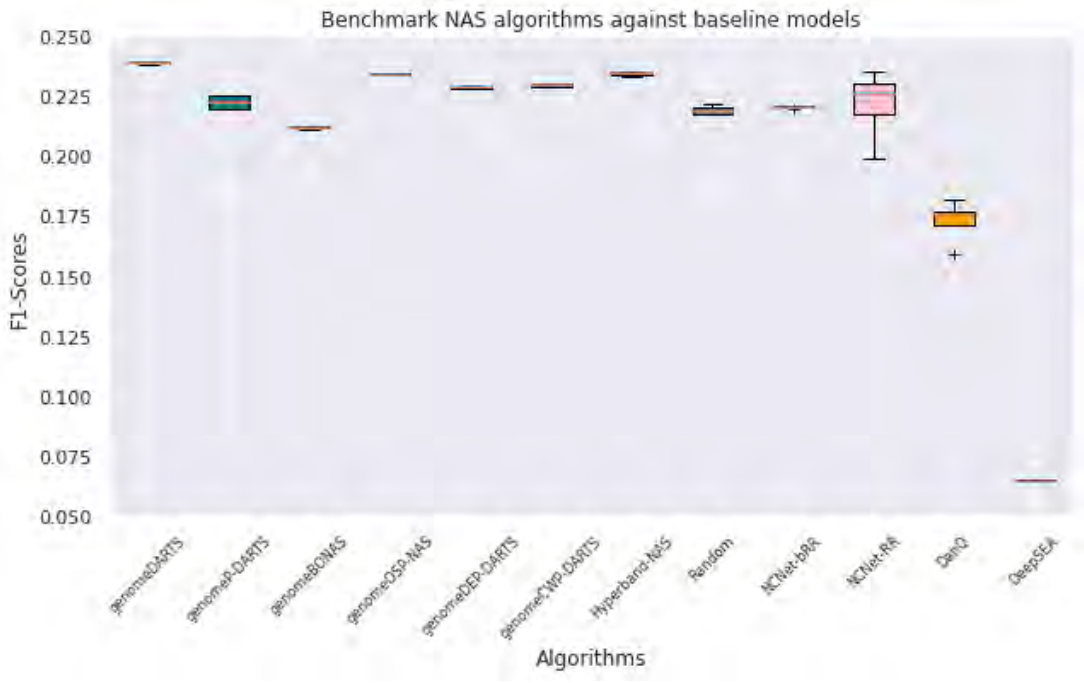
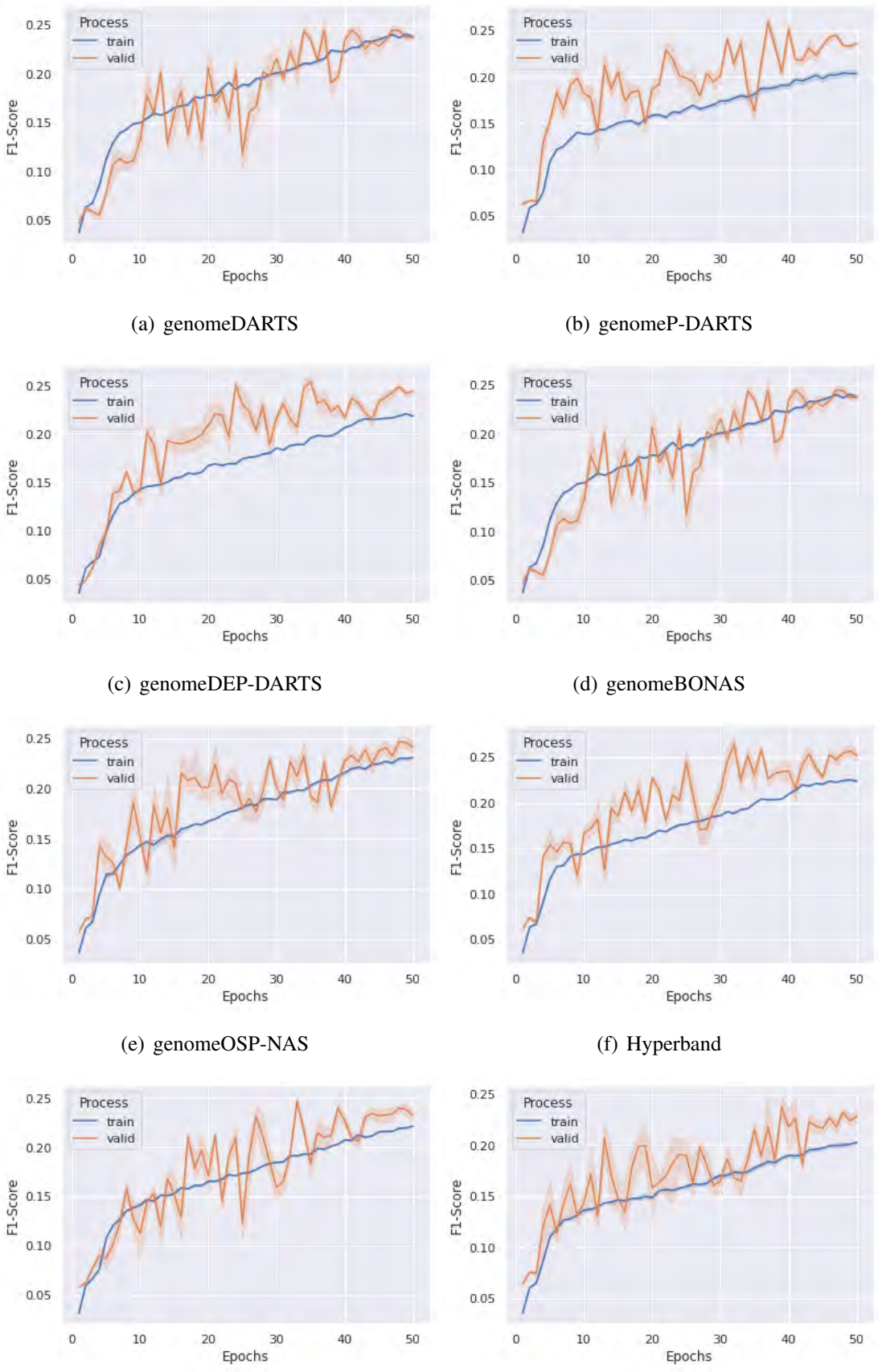


Figure 21: Benchmark NAS algorithms against baseline models. The figure shows the test F1-Scores of all genomeNAS algorithms and baseline models. For the genomeNAS algorithms, the boxes summarize the results of the best performing run, which is replicated four times.

of 21.20 and an average ROC-AUC score of 84.19, the final architecture of the genomeBONAS algorithm is the worst performing architecture. Although BONAS uses the validation performance of individual architectures, the weights of the individual architectures are determined by training multiple sub-architectures jointly. Therefore we claim that the validation performance of an individual architecture may be misleading because of the so-called gap between training and evaluation (X. Chen et al., 2019). For Hyperband on the other side, there is not such a gap between training and evaluation. The architecture which is used for training is also used for validation. As the boxplots from figure 21 indicate little variance in the results, obviously, a good performing architecture usually performs equally well over various runs. Figure 21 shows that the genomeDARTS algorithm produces a very good performing final architecture, but as can be seen in figure 20, it turns out that the genomeDARTS runs have some variance. GenomeDARTS produces some very good performing architectures, but also some bad performing architectures. We also include the run-time and number of parameters in our analysis. As can be seen in table 1, the final architectures of the genomeNAS algorithms have fewer parameters than the baseline models. While the genomeNAS architectures include 26.25-29.22 million parameters, the baseline models include 46.69-57.58 million parameters. The baseline models consist of more parameters because the convolutional part of the baseline models downsamples the input to a sequence of size 75, which is then fed to the recurrent layer and then to the fully connected layer. On the other hand, a final architecture of a genomeNAS algorithm downsamples the input to a sequence of size 42, which results in fewer units for the fully connected layer. Among

the genomeNAS algorithms, genomeDARTS produces the most complex architectures with on average 29.22 million parameters. On the other side, genomeBONAS produces the less complex models with on average only 26.25 million parameters.
The run-time of Random search, genomeDARTS, Hyperband, genomeOSP-NAS, and genomeDEP-DARTS are comparable and the algorithms approximately last 10 GPU days. The own designed algorithm genomeCWP-DARTS is the fastest algorithm and only needs 8.62 GPU days to complete the search process. GenomeBONAS is the slowest algorithm and lasts on average 24.03 GPU days.

Training process Figure 22 illustrates the training and validation process of the final architectures of the genomeNAS algorithms. The lines represent the average validation F1-Scores for the individual epochs and the shadows represent the 95 percent confidence interval over the four replications of the final architecture. In contrast to the validation process, the training process does not have a lot of variance. The plots show that all architectures tend to converge at the same point. Between the first and the 40th epoch, the validation F1-Scores are not smooth and the lines have a lot of spikes. After approximately 40 epochs, the model starts to converge and the validation F1-Scores stabilize. DeepSEA takes 7.18 minutes, DanQ 13.7, NCNet-RR 51.83 minutes, and NCNet-bRR 33.58 minutes to complete one epoch. The final architectures of the genomeNAS algorithms take on average 156 minutes (2.6 hours) to finish an epoch.



(a) genomeDARTS

(b) genomeP-DARTS

(c) genomeDEP-DARTS

(d) genomeBONAS

(e) genomeOSP-NAS

(f) Hyperband

(g) genomeCWP-DARTS

(h) Random search

Figure 22: Training and validation process. The plots show the training and validation F1-Scores of all genomeNAS Algorithms.

5 Conclusion and future work

It can be summarized that our unique designed search space works very well, as all genomeNAS algorithms showed strong performance on the DeepSEA task and outperformed current state-of-the-art baseline models as well as randomly sampled models. Even randomly sampled architectures from the search space outperform state-of-the-art models, such as DanQ. Therefore we conclude, that the unique combination of two directed acyclic graphs, as well as the usage of two different learning rates for the convolutional and the recurrent part, has great potential for further research.
However, it still remains a challenging task to adopt state-of-the-art NAS algorithms for genomic sequence data because algorithms such as P-DARTS consist of a lot of hyperparameters, and it is unclear which one works best for genomic sequence data. For instance, the appropriate number of stages or the best dropout rate of skip-connections can vary across different tasks or applications. The preliminary study results indicated high sensitivity to the hyperparameters of the genomeNAS algorithms. Especially the learning rate and the variational dropout rates of the recurrent highway network seem to have a big impact on model performance. While DARTS and ENAS use a learning rate of 20, we achieve good results with a learning rate of eight. It still remains unclear if there exists a general hyperparameter configuration for genomeNAS algorithms that works well over several genome tasks. Referring to the preliminary study results, a learning rate of eight seems to be a good choice as the initial learning rate for the recurrent part. We showed that the common NAS approach, where only the best performing architecture from the selection phase is selected, may be misleading. While genomeDARTS achieved the best performance in the evaluation phase, we showed that the selection phase of the genomeDARTS algorithm is affected by some variance. Another drawback is the computational demanding search process, as we have to search over two directed acyclic graphs instead of one as in common NAS algorithms. This leads to much longer run-times of the genomeNAS algorithms compared to common NAS approaches. In this work, we mainly focused on the DeepSEA task due to computational constraints.

Acknowledgement

I would like to thank my supervisors Dr. Mina Rezaei and Martin Binder for the inspiring weekly discussions. Also, the assistance provided by Prof. Dr. Bernd Bischl, Philipp C. Münch and Anil Gunduz was greatly appreciated. In addition, I would like to thank the Munich Center for Machine Learning and Helmholtz Centre for Infection Research for providing computational resources.

References

Baker, B. et al. *Accelerating Neural Architecture Search using Performance Prediction*. 2017. arXiv: 1705.10823 [cs.LG].

Bischl, B. et al. *mlrMBO: A Modular Framework for Model-Based Optimization of Expensive Black-Box Functions*. 2018. arXiv: 1703.03373 [stat.ML].

Bishop, C. M. *Pattern Recognition and Machine Learning*. Springer New York, 2006.

Brock, A. et al. *SMASH: One-Shot Model Architecture Search through HyperNetworks*. 2017. arXiv: 1708.05344 [cs.LG].

Chen, L.-C. et al. *Searching for Efficient Multi-Scale Architectures for Dense Image Prediction*. 2018. arXiv: 1809.04184 [cs.CV].

Chen, X. et al. *Progressive Differentiable Architecture Search: Bridging the Depth Gap between Search and Evaluation*. 2019. arXiv: 1904.12760 [cs.CV].

Elsken, T., J. H. Metzen, and F. Hutter. *Neural Architecture Search: A Survey*. 2019. arXiv: 1808.05377 [stat.ML].

Elsken, T., J.-H. Metzen, and F. Hutter. *Simple And Efficient Architecture Search for Convolutional Neural Networks*. 2017. arXiv: 1711.04528 [stat.ML].

Eraslan, G. et al. "Deep learning: new computational modelling techniques for genomics". In: *Nat Rev Genet* 20 (2019). eprint: 389âĂŞ403. URL: https://doi.org/10.1038/s41576-019-0122-6.

Gal, Y. and Z. Ghahramani. *A Theoretically Grounded Application of Dropout in Recurrent Neural Networks*. 2016. arXiv: 1512.05287 [stat.ML].

Goodfellow, I., Y. Bengio, and A. Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

He, K. et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

Huang, Y. and Y. Chen. *Autonomous Driving with Deep Learning: A Survey of State-of-Art Technologies*. 2020. arXiv: 2006.06091 [cs.CV].

Hutter, F., L. Kotthoff, and J. Vanschoren. *Automatic Machine Learning: Methods, Systems, Challenges*. http://automl.org/book. Springer, 2019.

Jian Zhou, O. T. "Predicting effects of noncoding variants with deep learning–based sequence model". In: *Nat Methods* 12 (2015). eprint: 931-934. URL: https://doi.org/10.1038/nmeth.3547.

Kingma, D. P. and J. Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

Kipf, T. N. and M. Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. 2017. arXiv: 1609.02907 [cs.LG].

Kushner, H. J. "A New Method of Locating the Maximum Point of an Arbitrary Multipeak Curve in the Presence of Noise". In: 86.1 (Mar. 1964), pp. 97–106. DOI: 10.1115/1.3653121. URL: https://doi.org/10.1115%2F1.3653121.

Lanchantin, J. et al. *Deep Motif Dashboard: Visualizing and Understanding Genomic Sequences Using Deep Neural Networks*. 2016. arXiv: 1608.03644 [cs.LG].

Lecun, Y. et al. *Efficient BackProp*. 1998.

Li, G. et al. *SGAS: Sequential Greedy Architecture Search*. 2020. arXiv: 1912.00195 [cs.LG].

Liu, C. et al. *Progressive Neural Architecture Search*. 2018. arXiv: 1712.00559 [cs.CV].

Liu, H., K. Simonyan, and Y. Yang. *DARTS: Differentiable Architecture Search*. 2019. arXiv: 1806.09055 [cs.LG].

Loshchilov, I. and F. Hutter. *SGDR: Stochastic Gradient Descent with Warm Restarts*. 2017. arXiv: 1608.03983 [cs.LG].

Mockus, J., V. Tiesis, and A. Zilinskas. "The Application of Bayesian Methods for Seeking the Extremum". In: *Towards Global Optimization* 2.117-129 (1978), p. 2.

Pham, H. et al. *Efficient Neural Architecture Search via Parameter Sharing*. 2018. arXiv: 1802.03268 [cs.LG].

Quang, D. and X. Xie. "DanQ: a hybrid convolutional and recurrent deep neural network for quantifying the function of DNA sequences". In: *bioRxiv* (2015). DOI: 10.1101/032821. eprint: https://www.biorxiv.org/content/early/2015/12/20/032821.full.pdf. URL: https://www.biorxiv.org/content/early/2015/12/20/032821.

Real, E. et al. *Regularized Evolution for Image Classifier Architecture Search*. 2019. arXiv: 1802.01548 [cs.NE].

Shahriari, B. et al. "Taking the Human Out of the Loop: A Review of Bayesian Optimization". In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175. DOI: 10.1109/JPROC.2015.2494218.

Shi, H. et al. *Bridging the Gap between Sample-based and One-shot Neural Architecture Search with BONAS*. 2020. arXiv: 1911.09336 [cs.LG].

Snoek Jasper, H. L. and R. P. Adams. *Practical Bayesian optimization of machine learning algorithms*. 2012. arXiv: 1206.2944 [cs.LG].

Srinivas, N. et al. "Information-Theoretic Regret Bounds for Gaussian Process Optimization in the Bandit Setting". In: *IEEE Transactions on Information Theory* 58.5 (May 2012), pp. 3250–3265. ISSN: 1557-9654. DOI: 10.1109/tit.2011.2182033. URL: http://dx.doi.org/10.1109/TIT.2011.2182033.

Srivastava, N. et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

Sun, C.-Y., A. C. .-H. Wu, and T. Hwang. *A Novel Privacy-Preserving Deep Learning Scheme without Using Cryptography Component*. 2020. arXiv: 1908.07701 [cs.CR].

Wang, R. et al. "DeepDNA: a hybrid convolutional and recurrent neural network for compressing human mitochondrial genomes". In: *2018 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. 2018, pp. 270–274. DOI: 10.1109/BIBM.2018.8621140.

Young, T. et al. *Recent Trends in Deep Learning Based Natural Language Processing*. 2018. arXiv: 1708.02709 [cs.CL].

Zhang, H. et al. “NCNet: Deep Learning Network Models for Predicting Function of Non-coding DNA”. In: *Frontiers in Genetics* 10 (2019), p. 432. ISSN: 1664-8021. DOI: 10.3389/fgene.2019.00432. URL: https://www.frontiersin.org/article/10.3389/fgene.2019.00432.

Zhang, Z. and M. R. Sabuncu. *Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels*. 2018. arXiv: 1805.07836 [cs.LG].

Zhang, Z., C. Y. Park, et al. “An automated framework for efficiently designing deep convolutional neural networks in genomics”. In: *bioRxiv* (2020). DOI: 10.1101/2020.08.18.251561. eprint: https://www.biorxiv.org/content/early/2020/08/19/2020.08.18.251561.full.pdf. URL: https://www.biorxiv.org/content/early/2020/08/19/2020.08.18.251561.

Zhao, Z.-Q. et al. *Object Detection with Deep Learning: A Review*. 2019. arXiv: 1807.05511 [cs.CV].

Zilly, J. G. et al. *Recurrent Highway Networks*. 2017. arXiv: 1607.03474 [cs.LG].

Zoph, B. and Q. V. Le. *Neural Architecture Search with Reinforcement Learning*. 2017. arXiv: 1611.01578 [cs.LG].

Zoph, B., V. Vasudevan, et al. *Learning Transferable Architectures for Scalable Image Recognition*. 2018. arXiv: 1707.07012 [cs.CV].

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und Zitate und gedankliche Übernahmen kenntlich gemacht habe.