International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 12, Number 1, pages 164–187, January 2022

Neural Architecture Search based on Genetic Algorithm and Deployed in a Bare-Metal Kubernetes Cluster

Andreas Klos
Chair of computer architecture, FernUniversität in Hagen
Hagen, North Rhine-Westphalia, 58097, Germany

Marius Rosenbaum
Chair of computer architecture, FernUniversität in Hagen
Hagen, North Rhine-Westphalia, 58097, Germany

Wolfram Schiffmann
Chair of computer architecture, FernUniversität in Hagen
Hagen, North Rhine-Westphalia, 58097, Germany

Received: July 25, 2021
Revised: October 25, 2021
Accepted: November 29, 2021
Communicated by Susumu Matsumae

Abstract

The interest in Deep Neural Networks has dramatically increased, especially e. g. in Computer Vision or Neural Language Processing tasks. Due to the heavy influence of the Neural Networks architecture on its predictive accuracy, Neural Architecture Search has gained much attention in recent years. Neural Architecture Search typically comes along with a high computational demand and thus, requires scalability as well as high availability to ensure no data loss or waste of computational power. Hence, we developed a scalable and highly available multi-objective Neural Architecture Search and adopted it to the modern thinking of developing applications by subdividing an already existing, monolithic approach – based on a Genetic Algorithm – into microservices. Moreover, we adjusted the initial population creation by mutating each individual 1,000 times, extended the approach by inception layers, implemented it as island model and achieved on MNIST, Fashion-MNIST and CIFAR-10 dataset 99.75%, 94.35% and 89.90% test accuracy, respectively. Furthermore, we analyzed nine different configurations of the Genetic Algorithm – with only one subpopulation – to identify well performing settings. Besides, our model is strongly focused on high availability empowered by the deployment in our bare-metal `Kubernetes` cluster. Our results show that the introduced Neural Architecture Search can easily handle and recover – without the necessity of human interaction – from the exceptional loss of `Kubernetes` pods within seconds and no loss of results or the algorithms state.

Keywords: neural architecture search, genetic algorithm, island model, kubernetes, microservices, high availability

1 Introduction

Due to the fact that the architecture of an Artificial Neural Network (ANN) heavily influence its predictive performance, numerous Neural Architecture Search (NAS) algorithms have been developed

in recent year. In the following we center our attention to NAS algorithms based on a genetic algorithm. To identify a well suited neural network architecture, usually a huge search space must be explored including e. g. the number of layers, the kind of each layer, activation functions, the quantity of neurons in each layer etc. Such a huge search space must be efficiently traversed and therefore, becomes restricted as e. g. in [1–3]. Furthermore, to reduce the computational demands early stopping [3, 4] as well as parameter sharing has already well proven to lessen the necessary number of epochs to provide sufficient results which leads to a minimized training time [3, 5]. Nevertheless, NAS depicts a high computational burden as e. g. the algorithm in [6] applied 200 GPUs for 1.5 days resulting in 300 GPU days. Thus, scalability as well as high availability are required to increase parallelism and reduce potential loss of results or the algorithms state which may force the user to run the algorithm from scratch again.

To guarantee the scalability, high availability and to adopt the software development to the changed thinking of developing microservices instead of monolithic applications, all components of the proposed NAS algorithm have been deployed in our bare-metal `Kubernetes` cluster. Our NAS algorithm is based on [3] which has been adopted to the Island Model Genetic Algorithm, as in [7], to facilitate scalability and avoid pre-mature convergence to one solution.

We subdivided the proposed NAS algorithm into the following microservices: `PostgreSQL` databases, Network File System (NFS) server and client for storing and exchanging results, `RabbitMQ` message brokers to facilitate Remote Procedure Calls (RPC), RPC clients as well as RPC workers. We used the message broker with the RPC pattern to assign task to our pods on the worker nodes.

We showed that our NAS deployment is scalable as well as highly available and capable to reach with compact architectures – low free parameter count – state of the art results regarding accuracy on MNIST [8], Fashion-MNIST [9] and CIFAR-10 [10] dataset. We achieved 99.75%, 94.35% and 89.90% accuracy on the test dataset after utilizing the NAS for 1,000 generations. Furthermore, we investigated several configurations of the proposed NAS algorithm to identify favorable settings.

This paper is the extended version of our previous investigations in [11] and enlarges the initial contribution by a more in-depth description of the proposed Genetic Algorithm. Furthermore, the key differences of the proposed NAS to related work are highlighted. Besides, we analyse the scalability of our approach as well as the utilized hardware resources more precisely. Moreover, the aspect of high availability is examined in a more systematic manner and results are reported by different configurations of the Genetic Algorithm.

The remaining paper is structured as follows: Section 2 summarizes related work and highlights the key differences to our proposed NAS algorithm. In Sec. 3 the developed NAS algorithm and its components are detailed. Section 4 is dedicated to the achieved results and its discussion. In Sec. 5 the paper ends in a conclusion and an outlook on our future works.

2 Related Work

The creation of a proper architecture of an ANN depicts a crucial task for its predictive performance. To develop a well suited architecture for the task at hand many decision e. g. about the kind of layer, number of neurons, activation function, number of feature maps and more have to be taken. In recent years, a vast amount of research has been done to identify appropriate architectures and to perform hyperparameter optimization. Many algorithms have been applied, as e. g. Evolutionary Algorithms, Bayesian Optimization, Reinforcement Learning [12] [13] [14] etc. In this paper we only consider related work based on Evolutionary Algorithms.

This paper is based on the Genetic Algorithm proposed in [3]. This algorithm represents the individuals internally by the architecture and their assigned fitness value. The architecture is depicted by a list of strings, each string describes the configuration of a layer. The possible layers are predefined comprising dense, convolutional, pooling, and residual layers. The selection takes place by roulette wheel. Every generation one or two individuals are selected, depending if mutation or crossover should be utilized. The initial population is created by mutating the starting architecture (only input and output layer) repeatedly, till the population reaches a certain size. Each generation, one new individual is created and evaluated regarding its fitness. The fitness values are the accuracy

reached on the test dataset and the number of free parameters of the corresponding architecture. Depending on some threshold, the fitness value for selection is the accuracy or the free parameter count. To facilitate a greater variety of our population, our approach differs by the creation of the initial population. Furthermore, we add inceptions layers and implemented the approach in a highly available and scalable fashion. The former property is achieved by deploying all components of our NAS in a `Kubernetes` cluster. The scalability of the proposed approach is enabled by implementing the algorithm as Island Model Genetic Algorithm.

In [15] another multi-population Genetic Algorithm is proposed. Each architecture is constrained to a stack of three cells and can be represented by a directed acyclic graph with maximum seven nodes and nine edges. Internally the individual is represented by two chromosomes. The first chromosome determines the layer type and position and the second chromosome the connections between the layers. The subpopulations can be evolved either by Regularized Evolution (RE) or Genetic Algorithm (GA). A group of subpopulations is called a tribe. Each Genetic Algorithm inside a tribe utilizes different mutation and crossover strategies. Thereby, two tribes exchange individuals. Mixed tribes (RE + GA) as well as only GA tribes are investigated. The stopping criteria is the maximum number of generations or the test error. New individuals are created by mutation or crossover. Two selection mechanisms are employed: 1) the best half of the ranked individuals in the current population is selected. Afterwards, the parents are randomly selected from those individuals. 2) The population is subdivided into two groups with the best and worst half of the individuals. Subsequently, they select individuals from the best group to perform mutation and one individual from both groups for crossover. Our proposed approach differs from selection, internal representation, migration schema and the fact, that we exclusively use Genetic Algorithm.

Moreover, in [16] a scalable NAS approach based on evolutionary search is presented. The approach identifies a module of convolutional layers and repeat it several times. The spatial dimension is reduced by max pooling layers. The number of layers inside a module is fixed to three. New individuals are created by mutation (adding layers) or crossover. Unfortunately, the internal representation, as well as selection etc. are left unconsidered. Nevertheless, the approach uses the remote procedure call pattern and facilitates a scalable NAS approach. In case of failure, the tasks survive, but human interaction is necessary to restart the failed workers which can be costly in the utilized AWS EC2 platform, if not sufficient monitored. Our approach differs that we use `RabbitMQ` as message broker, instead of developing our own. Additionally, we store our population, algorithm state and fitness information in a `PostgreSQL` database. Both, the message broker as well as the database are deployed highly available in a `Kubernetes` cluster. Furthermore, we perform selection and crossover in a different manner. Besides, our approach is deployed as Island Model Genetic Algorithm.

In [17] a NAS approach based on GA and Dynamic Structured Grammatical Evolution (DSGE) is introduced. Candidate solutions are represented by two independent levels: 1) GA level which encodes the macro structure of the architecture (which layer type can be used and their order) and depicts the sequence of the evolutionary units. Each unit in the sequence is the starting non-terminal symbol for expansion of the DSGE level genotype. 2) In the DSGE level, the parameters are stored in a backus-naur form grammar, represented as ranges, or closed sets of possibilities. New individuals are created by crossover and mutation operator. The predictive performance of an individual is used as fitness measure.

Besides, in [18] a multi-objective NAS which generates convolutional- and capsule-based architectures is proposed. The validation accuracy, energy consumption, latency and memory footprint are used as fitness measures. The algorithm is based on a specialized version of NSGA-II algorithm. The internal representation is performed by a layer descriptor (9-element position-based structure). To create a new individual mutation and crossover are applied. The search terminates after an execution duration has been exceeded.

In [19] a NAS algorithm based on a Genetic Algorithm is introduced. The population is randomly initialized. The architecture of an individual is represented by a gene map with the keys defining hyperparameter (e. g. activation, pooling, etc.) which are mapped to values sampled from a predefined value range. The elitism roulette wheel selection is used to pick individuals as parents. New individuals are created by the application of crossover or mutation on the selected parents.

None of the last three discussed publications focus on multi-population genetic algorithm, high

availability or scalability. Moreover, we implemented a different selection mechanism, target multi-objectives in a different way and have a simplified internal representation.

3 Neural Architecture Search Algorithm and Components

First the NAS algorithm and its adjustments are explained. Afterwards, each component of the proposed NAS algorithm is detailed.

3.1 Neural Architecture Search Algorithm

The proposed algorithm is based on [3] which utilizes the Genetic Algorithm. An individual of a population is represented by its architecture and the assigned fitness values. The architecture is represented by a list of strings. The first and the last element of the list are always the input and output layer. The input layer is depicted by `In_w_h` where `In` determines the kind of layer with the width `w` and height `h` of each input sample. The output layer is encoded by `D_n` where `n` specifies the number of output neurons. The hidden layers are inserted between the first and last element of the list and are represented either by `type_`k_w`_`k_h`_`s_w`_`s_h`_f` or `type.bt-e-fact_`k_w`_`k_h`_`s_w`_`s_h`_f.branch2.branch3.branch4` depending on the layer type. In [3], only convolution, max-pooling, residual and dense layers are used, which are encoded by the former string, while the inception layer is encoded by the latter string. The `type` can be one of the aforementioned layer types. The kernel width is indicated by k_w and the height by k_h. The stride is depicted by s_w as well as s_h and the number of feature maps is denoted by `f`. In case of the inception layer `bt` defines the branch type, `e` controls if the corresponding branch is employed and `fact` determines the factorisation applied to this branch. In sum, three more branches are described in the same fashion – indicated by `branch[2,4]`, each separated by a dot. The inception layer and its factorization types are based on [20]. The inception layer is implemented in three different variants: 1. Replacing convolution kernels ≥ 3 by smaller convolution kernels, 2. factorizing convolution kernels and 3. expanding the kernel outputs. If a branch is employed as well as the variant of inception layer is chosen randomly. Thereby, it is guaranteed, that at least one branch is evolved to avoid the creation of faulty neural network architectures.

The algorithmic procedure of the Genetic Algorithm is illustrated in Fig.1. Initially, each individual

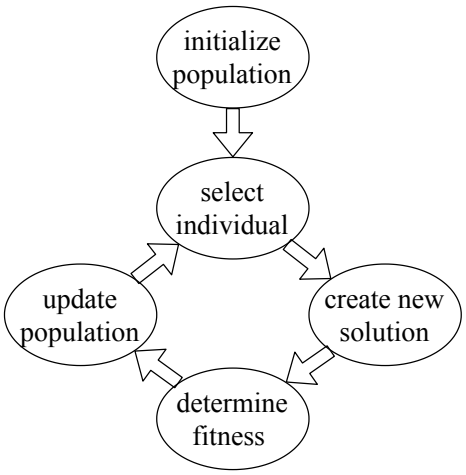


Figure 1: Genetic Algorithm

of the population is equal (only input and output layer are employed with a proper size according to the classification task). First, every individual of the population is mutated 1,000 times and the fitness is evaluated. Afterwards, an individual is selected from the population. In our case the selection is performed by roulette wheel selection (also known as fitness proportionate selection). During roulette wheel selection, each individual i of the population N has a probability p_i to become

selected as shown in Eq. 1

$$p_i = \frac{f_i}{\sum_j^N f_j} \tag{1}$$

where f_i and f_j are the assigned fitness values to the individuals i and j respectively. Depending on the kind of operation (mutation, migration or crossover) one or two parent individuals are selected. Thirdly, the new solution is created by applying one of the aforementioned operations to the selected parent(s). The operation selection takes place randomly, based on the pre-configured probability distribution. Subsequently, the fitness values of the newly created individual, namely accuracy and number of free parameters, are determined and assigned. The decision about which fitness value is picked for subsequent selections is based on the fitness of the worst individual as well as a threshold. The new individual is inserted into the population if its fitness value is better than the worst performing individual of the population, otherwise, the new individual is omitted. The fitness values of each individual determines which individual will be replaced by the newly created one. The described steps from selection to the update of the population with the new individual are repeated several times. Each circle leads to a new generation. The algorithm terminates if the number of max generations is reached.

Furthermore, the NAS algorithm has been changed to the Island Model Genetic Algorithm as illustrated in Fig. 2. The NAS algorithm comprises multiple – in the subsequent experiments five [1,5]

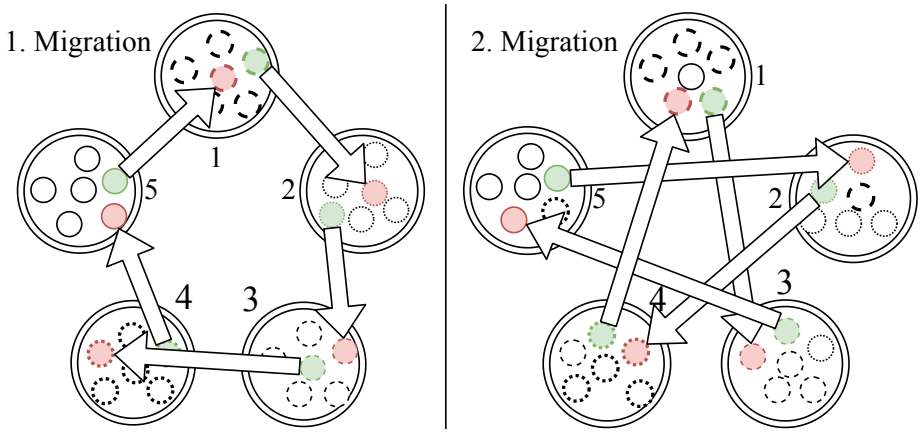


Figure 2: Two migration steps of the island model with five sub-populations. Selected individuals (green), replaced once (red).

– subpopulations per dataset as illustrated by double lined, big circles in Fig. 2. Therein, small circles depict an individual within a subpopulation. Green tagged circles highlight individuals migrated to another subpopulation and red circles denote the individuals which will be replaced in the destination subpopulation. Therefore, 30% of the best performing individuals are pre-selected as possible migrants. From those, the 5% of the subpopulation size are migrated to another subpopulation. The destination subpopulation is chosen as described by Alg. 1, where m_l is the previous migration subpopulation,

Algorithm 1: Determining migration destination subpopulation. Similar to [7]

```
Input: m_l := int, N_p := int, p := int, N_d := int
Output: m_c := int
m_c ← (m_l + N_d)%N_p
if m_c == p or m_c == 0 then
    m_c ← (p + N_d)
    if m_c! = N_p then
        m_c = m_c%N_p
m_l ← m_c
```

N_p denotes the population count, p depicts the current population number, N_d is the dataset count

and m_c is the current migration subpopulation of the latest generation. N_d is necessary, if more than one dataset should be utilized at once and if subpopulations, exclusively working on the same dataset, should exchange migrants.

The overall configuration of the proposed NAS is shown in Tab. 5 (see Appendix A). Our approach favours small architectures, as soon as the worst individual of a subpopulation has exceeded the fitness thresholds mentioned in Tab. 5.

3.2 Neural Architecture Search Components

Our bare-metal `Kubernetes` cluster consists of ten worker nodes, each with the following hardware components: AMD Ryzen 9 3900X, 1 TB SSD NVME XPG GAMMIX S50, 16 TB SATA WD40EFRX Red as software RAID 5, 32 GB DDR4 RAM, two worker nodes, each with one NVIDIA GeForce RTX 2080 Ti and eight worker nodes, each with one NVIDIA GeForce RTX 3090. The master comprises an Intel(R) Core(TM) i7-2600K, 1 TB ST1000DM003-1CH1_Z1D2A4DX, 12 GB DDR3 RAM, and NVIDIA GeForce GTX 460. We use: `Kubernetes` 1.19.13 initialized with `kubeadm` with Host OS `Ubuntu` 20.04.1 LTS, CNI `Weave Net` 2.7.0 and CRI `Docker` 19.3.13.

The monitoring of our cluster is done with `Prometheus`, `DCGM-exporter` and `Grafana`. In case e. g. of overheating the GPU or CPU, Grafana will send a notification to our `Discord` channel, so that in worst case the cluster administrator can take action.

Our deployment of the proposed NAS algorithm is composed of four microservices (`Genetic Algorithm`, `Fitness Evaluator`, `PostgreSQL` database and `RabbitMQ` message broker) as shown in Fig. 3. Each microservice is composed of various `Kubernetes` resources which are briefly explained in

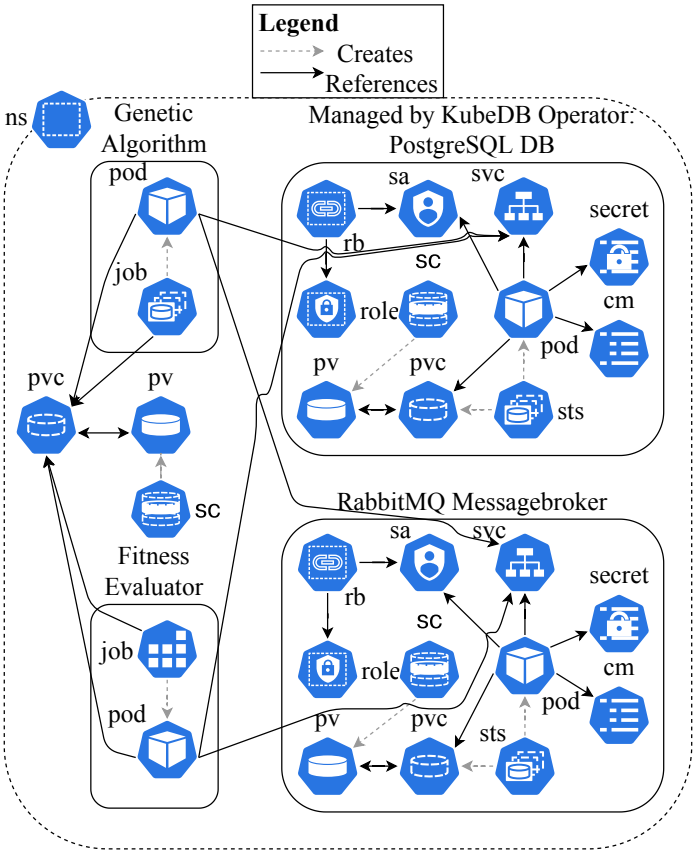


Figure 3: Decomposed Neural Architecture Search `Kubernetes` cluster setup. Gray dashed arrows indicate creation, whereas black solid arrows illustrate references between individual resources.

the following. A service (`svc`), in `Kubernetes`, is an abstraction defining the policy by which a logical

set of pods is accessible. `svc` facilitating pod intern communication of a microservice are omitted in Fig. 3. The namespace (`ns`) determines where the resources will be deployed. The `role`, role binding (`rb`) as well as the service account (`sa`) facilitate role-based access control. A `pod` represents the smallest deployable unit of computing composed of one or more containers with shared storage as well as network resources and specifies, how to run the container(s). Persistent volumes (`pv`) are pieces of storage in the cluster with their own lifecycle independent of any individual `pod` using it. A Persistent volume claim (`pvc`) denotes a request for storage (`pv`) by some user. If a proper `pv` is identified, the `pv` becomes bound to the `pvc`. The `pvc` can be used to mount the storage of the `pv` at a specific location in the container. The stateful set (`sts`) manages the deployment and scaling of a set of `pods` with guarantees about the ordering and uniqueness of those. Besides, the `sts` maintains a pinned identity for each of its pods. `ConfigMaps` (`cm`) are used to store non-confidential data in key-value pairs which can be consumed by `pods` as environment variables, command-line arguments, or as configuration files in a volume. `Secrets` contain sensitive data such as passwords, tokens, or keys to avoid the necessity to include confidential data in the application code. A `job` creates one or more `pods` and keeps track about successful completions. If the pre-defined number of successful completions is not met, the `job` will create new pods till the number of successful terminations is met [1].

The utilized `PostgreSQL` database is managed by the `KubeDB` operator. The database is deployed threefold, each on a different worker node. One database – further called DB1 – serves incoming queries in master mode, while the other databases – DB2 and DB3 – are in standby mode to replicate the contents stored in DB1 by streaming replication. Streaming replication means that the databases in standby mode will stay up-to-date by shipping and applying the Write-Ahead Logging (WAL) records continuously. The master DB streams the WAL records to the standby DBs as they are generated, without waiting for the WAL file to be filled. Due to the asynchronous behaviour of the streaming replication, small delays between committing a transaction in the master DB and the changes becoming visible in the standby DBs exist, which in the worse case can lead to data loss. In case that DB1 fails e. g. caused by a software bug or faulty hardware, DB2 or DB3 will take over in master mode through a leader election after a pre-configured amount of time.

The `RabbitMQ` message broker is deployed as statefulset. In sum three pods are created by the statefulset. If a new message becomes published to a queue it will be replicated to all mirrors. The primary replica – called master – is the one created first and all operations for a given queue are performed on the masters queue first. If a message have been acknowledged by the master, the secondary replicas – called mirrors – drop the corresponding message from the queue too. If the master fails, the oldest queue will be promoted to the new master as long as it is synchronised. The message broker is used for distributing the work (fitness evaluations) by the RPC pattern from the NAS algorithm to the `Fitness Evaluator` pods.

Each `Fitness Evaluator` pod utilizes `PyTorch` and depicts a RPC worker, whereas the NAS (genetic algorithm) pods are the RPC clients. If one of the RPC workers or clients fail, a new pod will be started as soon as the crash is realized by the `kubelet`. The RPC worker pods are stateless, thus a crash of those pods is uncritical, while the pods of the RPC clients have a state, stored in the aforementioned database. If a new pod is launched, the database is queried, in case that a state of the current pod is available, the NAS algorithm is continued from that state on. To assign every container in all NAS pods a certain population, a counter in a table of the database is incremented one by one, facilitated by querying locks and suitable unlocking. If a NAS pod fails, the pod verifies, which pods are not reachable anymore (by pinging all pod IPs stored in the state table of the corresponding database), or if the same IP address has already allocated a certain population and updates the state table properly.

Weight sharing is facilitated by a NFS share between the pods of the NAS and the `Fitness Evaluator`. The trained models are saved in the NFS and the results regarding training, validation, test, NAS state and information about every single population are stored in the database.

The RPC communication scheme between the n NAS populations, the message broker and n `Fitness Evaluator` pods is exemplified by Fig. 4.

[1] For further information about `Kubernetes` the authors refer to [21] or the documentation [22].

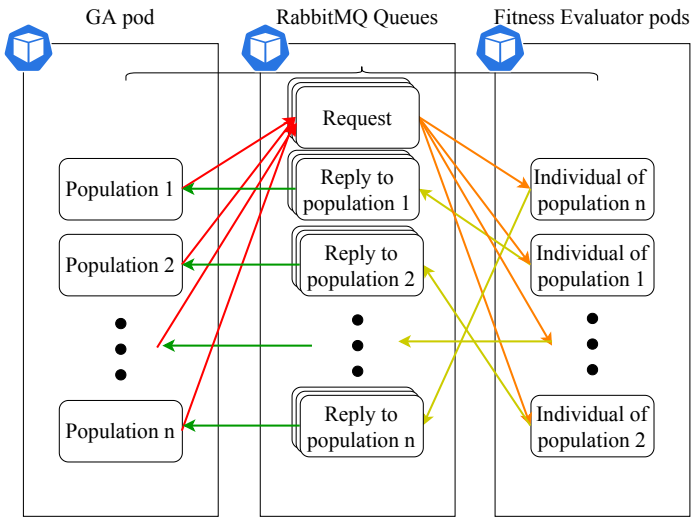


Figure 4: RPC setup. Red arrows: RPC client send request; Orange arrows: RPC worker receives request; Yellow arrows: RPC worker send reply; Green arrows: RPC client receive results.

Obviously, each NAS pod sends its task to the `Request` queue. Each RPC worker polls one task at once, process it and publish the achieved results to the dedicated queue as well as acknowledge the task to be processed successfully. The RPC client receives the message on the corresponding response queue, updates the population and creates the next generation of the population. Each task is formatted as the subsequent string:

`population~generation~input*kfolds*loi~architecture!checkpointFilename`

where the `population` is the population number, `generation` is the current generation, `input` is in our case `MNIST`, `FMNIST` or `CIFAR-10`, `kfolds` determines if k-fold cross validation should be performed and how many folds are used, in the subsequent analysis this value was one and `loi` defines which layer requires the computation of the gradient during fitness calculation. The `architecture` is the concatenation of the internal reprentation of the individuals architecture (a list of strings, see Sec. 3.1) separated by a double dot:

`input_layer:hidden_layer:...:output_layer`

The response of the RPC worker is a string as follows:

`hash,architecture,checkpointFilename,generation,accuracy,loss,executionTime,freeParameters,age`

where the `hash` is generated with Pythons `hashlib` based on the current time, queried by the `datetime` package.

4 Results and Discussion

First the results achieved with the Island Model Genetic Algorithm are detailed and discussed. Afterwards, the results obtained with nine different configurations of the Genetic Algorithm – with only one subpopulation – are presented and discussed. Thirdly, the high availability, scalability as well as utilized hardware is investigated.

4.1 Island Model Genetic Algorithm

After 1,000 generations of each subpopulation, the NAS algorithm terminates. The achieved test accuracies as well as those reached by related work are summarized in Tab. 1.

Table 1: Accuracy overview achieved on test dataset. In column p are the results reached by applying some pre-processing of the input data in addition to normalization or standardization. $\bar{p}$ shows the results obtained with only normalization or standardization as pre-processing of the input data.

NAS	**MNIST [%]**		**F.-MNIST [%]**		**CIFAR-10 [%]**	
	$\bar{p}$	p	$\bar{p}$	p	$\bar{p}$	p
Hajewski et al. [16]	–	–	–	–	76.8	
Yotchon et al. [15]	–	–	–	–	88.82	–
Assunção et al. [17]	99.65	99.7	94.23	**94.7**	88.41	**92.51**
Ma et al. [19]	–	**99.72**	–	94.6	–	89.32
Marchisio et al. [18]	99.72	–	93.34	–	85.99	–
Litzinger et al. [3]	99.69	–	93.58	–	85.16	–
Proposed approach	**99.75**	–	**94.35**	–	**89.90**	–

In [3] – the paper of the base algorithm – results are stated neither for the Fashion-MNIST nor the CIFAR-10 dataset. For a better comparability, we utilize our algorithm with only one subpopulation and without the inception layer for three runs. The aforementioned configuration is assumed to be similar to the one in [3]. The highest reached accuracy on both datasets for this configuration is shown in Tab. 1 in the row dedicated to the base algorithm ([3]).

The best performing, evolved architecture reached an accuracy of 99.75% and hence, outperformed the base algorithm proposed in [3] with an accuracy of 99.69% by having less than half the number of free parameters – in sum 966,215. The evolved architecture is shown in Fig. 9 (see Appendix B) and composed of two residual, two convolutional, one pooling and two inception layers.

The developed architecture has a lower error rate (0.25%) than all single predictive models stated in [23]. Compared with the results achieved by other NAS algorithms, the proposed one was capable to evolve state of the art results. E. g. in [18] the framework `NASCaps` applied to the MNIST dataset achieved a test accuracy of 99.72%. Besides, in [19] the same test accuracy on MNIST dataset of 99.72% has been reported. For the DENSER NAS algorithm proposed in [17] a test accuracy of 99.7% was achieved. Those mentioned results are comparable to the 99.75% accuracy achieved by our NAS algorithm.

The highest accuracy achieved on the Fashion-MNIST test dataset by the proposed NAS approach is 94.35% with 3,774,376 free parameters. Compared with the base algorithm, ours surpass the accuracy by 0.77% with about 1.85 times more free parameters. The overall architecture is shown in Fig. 10 (see Appendix B).

The evolved neural network consists of one residual, one pooling, one dense and one inception layer. This architecture is capable to outperform all models – without data augmentation – listed in [24] except the model generated by DENSER which is stated with an accuracy of 95.3%. In [17] the test accuracy reached by DENSER is reported as 95.26%. This accuracy was achieved by the utilization of an ensemble of the best performing two network architectures. The accuracy reached by the best single architecture evolved by DENSER is stated as 94.23% and 94.7%, not using and using data augmentation on the test dataset respectively. In [18] a test accuracy of 93.34% and in [19] of 94.6% was reached. Note that, in [19] data augmentation was used to increase the accuracy.

On the CIFAR-10 dataset the evolved architecture – shown in Fig. 11 (see Appendix B) – reached an test dataset accuracy of 89.90% with 2,055,914 free parameters. Against the results achieved by the base algorithm, the accuracy differs by 4.74% by having a comparable amount of free parameters. The developed architecture compromises two inception, three residual, three convolutional, five pooling and one dense layer. As shown in Tab. 1, the introduced NAS approach achieved a 13.1% higher accuracy than in [16]. Besides, our NAS algorithm outperforms the other mentioned multi population approach in [15], which is reported with 88.82%. In [19] the NAS algorithm reached an accuracy of 89.32%, quite similar to the proposed approach. In [18] an accuracy of 85.99% has been reported. This approach tackles multiple objectives at once, namely accuracy, energy consumption, memory footprint and latency, which might affect the resulting accuracy. The accuracy reported by DENSER in [17] is 92.51% and is achieved after tweaking the hyperparameter for training the network. Without this hyperparameter adjustment, the best performing architecture summed up

to an average accuracy of 88.41% over several different runs. The maximum accuracy under this circumstances are not mentioned in [17].

The above stated results for the MNIST, Fashion-MNIST and the CIFAR-10 dataset might be further improvable by applying data augmentation or optimizing hyperparameters. Besides adjusting the depth, the layer types and its configuration, scaling the input resolution could lead to enhanced results. In [25] the simultaneous scaling of the neural networks depth, width and resolution resulted in high accuracy on various datasets, including CIFAR-10. Adjusting the Genetic Algorithm more drastically, could lead to superior results, as e. g. replacing the selection method as well as the underlying fitness measures (see NSGA-II [26], NSGA-III [27], SEPA-2 [28], SMS-EMOA [29]) or the whole island model as surveyed in [30].

4.2 Genetic Algorithm with only one Subpopulation

In the following the results achieved by only one subpopulation are presented. Due to the computational demand, each configuration has been executed only three times for 1,000 generations. For that reason, statistical measures as mean and standard deviation might lack statistical relevance. The configurations are summarized in Tab. 2. Obviously, the kind of available layers, the minimum

Table 2: Configurations of the Genetic Algorithm

Config.	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**	**#9**
Add. layers	R	R, I	I	R	R	IF1	R, IF1	IF1	IF1
Min. age	1	1	1	5	1	1	1	1	1
K-folds	`False`	`False`	`False`	`False`	5	`False`	`False`	`False`	`False`
Pop. size	20	20	20	20	20	20	20	100	20
Seed	`False`	`False`	`False`	`False`	`False`	`False`	`False`	`False`	`True`

required age, till an individual can serve as parent, if k-folds is utilized, the population size and if seed architectures are preliminary created is adjusted. Only one adjustment has taken place per configuration to determine the impact of the alteration on the results. By investigating and constraining some layers, an ablation study is performed to see the impact of the residual and inception layer on the achieved results. Besides, the minimum age for being a parent is investigated to reduce premature convergence till only one root parent is left in the population. k-folds cross-validation is applied to enlarge the number of training samples and evaluate the effects on the predictive performance. Moreover, the population size is adjusted to verify the impact of selecting individuals from a greater variety across the population. Finally, the consequences of seed architectures are investigated so that the initial population only consists of promising network individuals.

In the row denoted to layers, only the residual (R), inception (I) and inception with fixed factorization 1 (IF1) layer are mentioned. The dense, pooling and convolutional layers are present in each configuration. To speed up the analyses, the activation function has been adjusted to ReLU. Because the CIFAR-10 dataset depicted the most challenging dataset in the previous experiments and variation by different configurations of the Genetic Algorithm was most significant, the subsequent investigations are exclusively performed on that dataset.

The achieved results during the three runs with config. #1 and #2 on the CIFAR-10 dataset are summarized in Tab. 3. Both configurations accomplish similar results. However, config. #1 outperformed config. #2 regarding $\max_{\text{acc}}^{\text{test}}$ (diff: 0.36%) as well as $\mu_{\text{acc}}^{\text{test}}$ (diff: 0.01%) by having a bigger $\sigma_{\text{acc}}^{\text{test}}$ (diff: 0.15%). The $\text{loss}^{\text{test}}$ (diff: 0.2×10^{-3}), $\mu_{\text{loss}}^{\text{test}}$ (diff: 0.05×10^{-3})) and $\sigma_{\text{loss}}^{\text{test}}$ (diff: 0.05×10^{-3}) are quite comparable. N of the best performing architecture identified with config. #1 is about 116,000 free parameters bigger than the one created with config. #2. μ_{N} for the best architectures created with config #1 as well as the σ_{N} are bigger (diff: 38,000 $\pm$ 327,840).

In Fig. 5 a) and b) the populations maximum training, validation and test accuracy of each generation, achieved with config. #1 and config. #2, are shown. Moreover, the mean as well as the standard deviation are shown in Fig. 5. As expected, the gap between the training and the test accuracy becomes smaller during the progress of the NAS (with config. #1 diff: 14.13% and

Table 3: Max., mean and standard deviation of the accuracy and the corresponding loss achieved on the test dataset. Number of free parameters of the best performing model as well as the mean and standard deviation of the number of free parameters of each run.

	CIFAR-10					
Config.	$\mathbf{max_{acc}^{test}}$ %	$\boldsymbol{\mu_{acc}^{test} \pm \sigma_{acc}^{test}}$ %	$\mathbf{loss^{test}}$ $\times 10^{-3}$	$\boldsymbol{\mu_{loss}^{test} \pm \sigma_{loss}^{test}}$ $\times 10^{-3}$	**N** $\times 10^3$	$\boldsymbol{\mu_N \pm \sigma_N}$ $\times 10^3$
#1	85.16	84.05 ± 1.22	2.37	2.66 ± 0.25	2089	2154 ± 651.93
#2	84.8	84.04 ± 1.07	2.57	2.71 ± 0.30	1973	2116 ± 324.09
#3	84.73	83.56 ± 1.74	2.39	2.73 ± 0.45	1638	2035.33 ± 355.18
#4	84.38	82.92 ± 1.32	2.56	2.82 ± 0.27	**1465**	2059.67 ± 726.90
#5	84.5	83.60 ± 1.27	2.74	2.86 ± 0.09	2278	1879.33 ± 345.78
#6	86.22	**85.40 ± 0.74**	2.25	**2.35** ± 0.13	1982	1830 ± 155.82
#7	**86.92**	85.21 ± 1.63	**2.12**	2.43 ± 0.28	2447	2118.67 ± 826.92
#8	82.8	81.48 ± 1.09	2.65	2.89 ± 0.24	1700	**1796.67 ± 100.17**
#9	85.54	85.06 ± **0.74**	2.39	2.43 ± **0.06**	1792	1945.33 ± 855.37

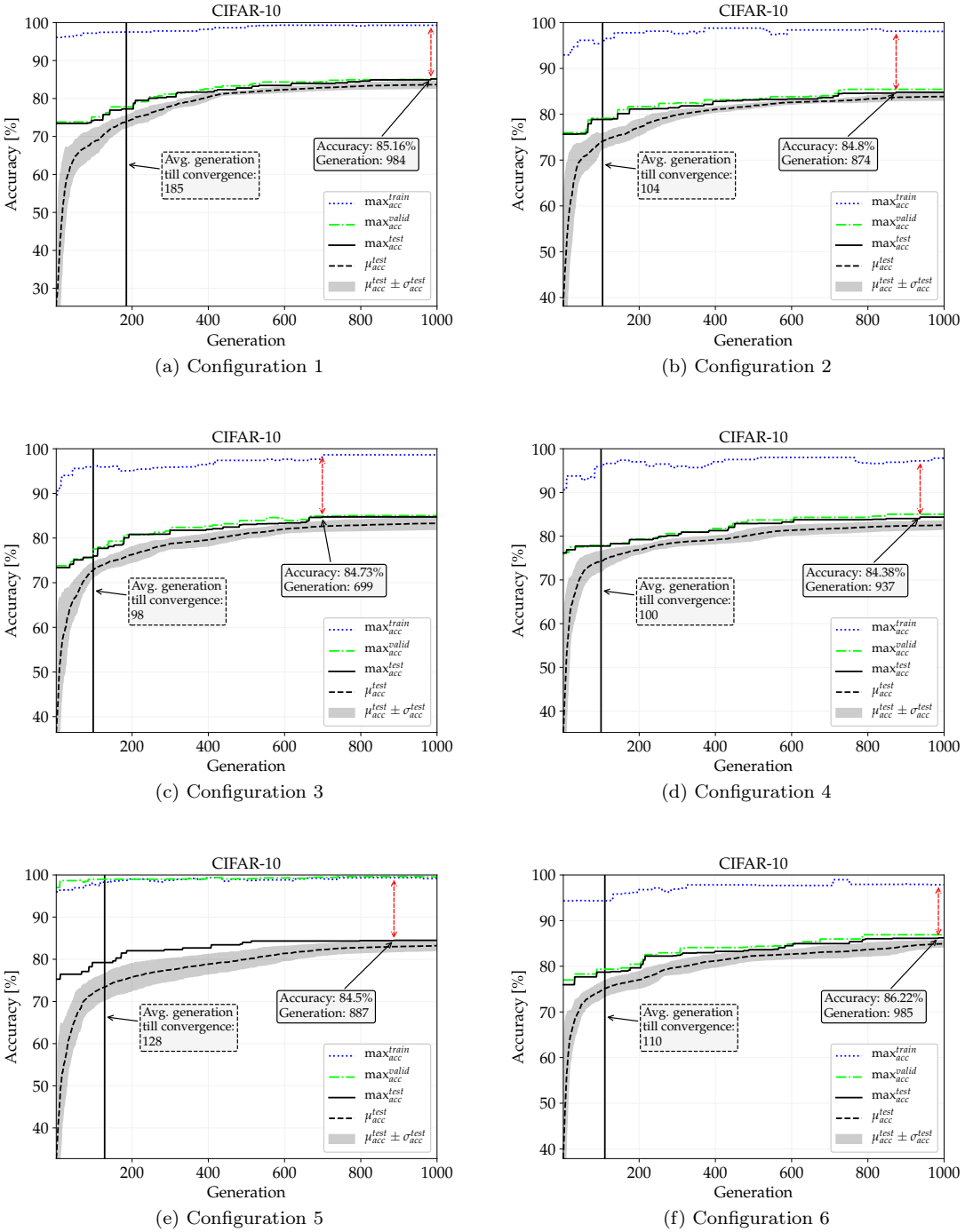
config. #2 diff: 13.31% at the generation in which the max. accuracy on the test dataset has been encountered the first time) – this is also the case for all subsequent analyses. With config. #1 the best result on the test dataset has been achieved in generation 984, whereas the Genetic Algorithm with config. #2 found the best performing individual in generation 874. This could imply that there might be still the opportunity to improve with both configurations if the Genetic Algorithm would be executed more than 1,000 generations. Moreover, averaged over the three runs with config. #1 the population of the Genetic Algorithm exists only of siblings of one root parent in generation 185 and with config. #2 in generation 104. In summary, config. #1 led to better results regarding the max. and mean accuracy on the test dataset, while the emerged best performing individual with config. #1 is a little bigger considering the number of free parameters, μ_N and σ_N.

Against our expectation, the introduction of the inception layer lead to worse results regarding the accuracy reached on the CIFAR-10 test dataset. To investigate, if the combination of residual and inception layer directs to unfavourable results, we removed the residual layer in config. #3. The achieved results during the three runs with config. #3 on the CIFAR-10 dataset are presented in Tab. 3.

Compared to the results stated for config. #1, the removal of the residual layer leads to poorer accuracies on the test dataset. Configuration #1 outperformed config. #3 regarding $\max_{acc}^{test}$ (diff: 0.43%) as well as μ_{acc}^{test} (diff: 0.49%) with a smaller σ_{acc}^{test} (diff: 0.52%). The $loss^{test}$ (diff: 0.02×10^{-3}), μ_{loss}^{test} (diff: 0.07×10^{-3})) and σ_{loss}^{test} (diff: 0.2×10^{-3}) are quite comparable. N of the best performing architecture identified with config. #3 is about 451,000 free parameters smaller than the one created with config. #1. μ_N for the best architectures created with config #3 as well as the σ_N are smaller (diff: 118,670 ± 296,750).

In Fig. 5 c) the populations maximum training, validation and test accuracy of each generation achieved with config. #3, are presented. Besides, the mean as well as the standard deviation are shown in Fig. 5 c). As expected, the gap between the training and the test accuracy becomes smaller during the progress of the NAS (with config. #3 diff: 13.91% at the generation in which the max. accuracy on the test dataset has been encountered the first time). With config. #3 the best result on the test dataset has been achieved in generation 699. Due to the fact, that many consecutive generations an improvement regarding test accuracy is missing, even a longer execution (more generations) of the NAS might not result in an improved accuracy. Averaged over the three runs, with config. #3 the population of the Genetic Algorithm exists only of siblings of one root parent in generation 98. In summary, config. #3 led to compacter architectures accompanied by a worse accuracy on the test dataset.

To avoid premature convergence to the siblings of only one root parent, we investigate if a minimal necessary age for being a parent directs to better results and a higher diversity in the population during the course of the NAS (see config. #4).



(a) Configuration 1 (b) Configuration 2

(c) Configuration 3 (d) Configuration 4

(e) Configuration 5 (f) Configuration 6

Figure 5: Maximum accuracy achieved each generation on training, validation and test dataset (blue dotted, green dash-dotted, black solid line respectively). Black dashed line: mean maximum accuracy per individual in each generation, Gray area: standard deviation, Vertical line: average generation, from which only the gens of one root parent are present, Red arrow: Difference between the max. test and training accuracy in the corresponding generation.

In Tab. 3 the achieved results during the three runs with config. #4 are summarized. Compared to the results stated for config. #1, the minimum required age for being a parent leads to worse

accuracies on the test dataset. Configuration #1 outperformed config. #4 regarding $\max_{\text{acc}}^{\text{test}}$ (diff: 0.78%) as well as $\mu_{\text{acc}}^{\text{test}}$ (diff: 1.13%) with a smaller $\sigma_{\text{acc}}^{\text{test}}$ (diff: 0.1%). The $\text{loss}^{\text{test}}$ (diff: 0.19×10^{-3}), $\mu_{\text{loss}}^{\text{test}}$ (diff: 0.16×10^{-3})) and $\sigma_{\text{loss}}^{\text{test}}$ (diff: 0.02×10^{-3}) are quite comparable. N of the best performing architecture identified is about 624,000 free parameters smaller than the one created with config. #1. μ_{N} for the best architectures created with config #4 is smaller (diff: 94,330) by having a larger σ_{N} (diff: 74,970).

In Fig. 5 d) the populations maximum training, validation and test accuracy of each generation achieved with config. #4, are presented. The mean as well as the standard deviation are shown in Fig. 5 d). As expected, the gap between the training and the test accuracy becomes smaller during the progress of the NAS (with config. #4 diff: 12.87% at the generation in which the max. accuracy on the test dataset has been encountered the first time). With config. #4 the best result on the test dataset has been achieved in generation 937. This could imply, that a longer execution (more generations) of the NAS might result in a better accuracy on the test dataset. Averaged over the three runs, the population of the Genetic Algorithm exists only of siblings of one root parent in generation 100. In summary, config. #4 led to compacter architectures accompanied by a worse accuracy on the test dataset. Unfortunately, the reason for introducing a minimum age still directs to a fast convergence, in the sense that only siblings of one root parent are present in the population.

Afterwards, we investigated, if the well known k-fold cross-validation might improve the accuracy achieved on the test dataset. The obtained results are shown in Tab. 3 as config. #5. Compared to the other already mentioned configurations, superior results are neither for the accuracy, the loss nor the number of free parameters reported.

In Fig. 5 e) the populations maximum training, validation and test accuracy as well as the mean and the standard deviation of the test accuracy are shown. The gap between the training and the test accuracy at the generation in which the max. accuracy on the test dataset has been encountered the first time is 14.87%. With config. #5 the best result on the test dataset has been achieved in generation 887. This could imply, that a longer execution (more generations) of the NAS might result in a better accuracy on the test dataset. In average, the siblings of only one root parent exist in the population in generation 128. Recapitulatory can be said, that for config. #5 superior results were missing.

To further analyze the poor results achieved by the introduction of the inception layer, we fix the kind of factorization to variant 1 – Replacing convolution kernels ≥ 3 by smaller convolution kernels – and thus, constrain the search space. The obtained results on the CIFAR-10 dataset are shown in Tab. 3 for config. #6.

Compared to the previously stated results for the other configurations, fixing the kind of factorization led to superior results. By comparing the results with config. #6 to the best config. – till now config. #1 – the $\max_{\text{acc}}^{\text{test}}$ differ 1.06%, $\mu_{\text{acc}}^{\text{test}}$ differ 1.35% with a 0.48% smaller $\sigma_{\text{acc}}^{\text{test}}$. The $\text{loss}^{\text{test}}$ (diff: 0.12×10^{-3}), $\mu_{\text{loss}}^{\text{test}}$ (diff: 0.31×10^{-3}) and $\sigma_{\text{loss}}^{\text{test}}$ (diff: 0.12×10^{-3}) are comparable. N of the best performing architecture identified with config. #6 is about 107,000 free parameters smaller than the one created with config. #1. μ_{N} for the best architecture created with config #6 as well as the σ_{N} are smaller (diff: 324,000 $\pm$ 496,110).

In Fig. 5 f) the populations maximum training, validation and test accuracy as well as the mean and the standard deviation of the test accuracy are shown. The gap between the training and the test accuracy at the generation in which the max. accuracy on the test dataset has been encountered the first time is 11.64%. With config. #6 the best result on the test dataset has been achieved in generation 985. This could imply, that a longer execution (more generations) of the NAS might result in a higher test accuracy. Besides, after 110 generations only the siblings of one root parent are present in the population. In summary led config. #6 to compacter architectures accompanied by a higher maximum as well as mean accuracy on the test dataset compared to config. #1.

The last three configurations of the Genetic Algorithm have been analyzed in parallel. Configuration #7 comprises residual and inception layers with fixed factorization variant 1, to verify, if a combination of those layers may results in a favourable setting. With config. #8 the premature convergence to the siblings of only one root parent is investigated by increasing the population size to 100. Lastly, with config. #9 the utilization of seed architectures is analyzed. In config. #9 all initial individuals are 50 times mutated. After each mutation, the fitness is approximated by training the

resulting neural network only a few iterations (5) with few samples of the dataset (256). Afterwards, the validation accuracy is determined with 1/10 of the training dataset size.

The reached results with config. #7, #8 and #9 are shown in Tab. 3. The max. accuracy has been achieved with config. #7. The results obtained by config. #8 are poor, compared to the other ones in Tab. 3 with the lowest maximum as well as mean accuracy achieved the test dataset. The number of free parameters for the best performing architecture, μ_{N} and its σ_{N} are small. With config. #9 a higher max. and mean accuracy as with config. #1 have been reached. Nevertheless, the values for the max. and mean accuracy are smaller than those obtained with config. #6 and #7. For that reason, in the following only the results achieved with config. #7 are compared to those of config. #6.

In comparison, the $\max_{\mathrm{acc}}^{\mathrm{test}}$ obtained with config. #7 is 0.7% higher than the one with config. #6. For $\mu_{\mathrm{acc}}^{\mathrm{test}}$ as well as $\sigma_{\mathrm{acc}}^{\mathrm{test}}$ better results are achieved with config. #6 (diff: $0.2\% \pm 0.89\%$). The $\mathrm{loss}^{\mathrm{test}}$ of the best performing architecture created with config. #7 is smaller than the one with config. #6 (diff: 0.13×10^{-3}), whereas $\mu_{\mathrm{loss}}^{\mathrm{test}}$ and $\sigma_{\mathrm{loss}}^{\mathrm{test}}$ are smaller with config. #6 ($0.08 \times 10^{-3} \pm 0.15 \times 10^{-3}$). N of the best performing architecture identified with config. #7 is about 465,000 free parameters bigger than the one generated with config. #6. μ_{N} for the best architectures created with config #7 as well as the σ_{N} are bigger (diff: 288,670 ± 671,100).

In Fig. 6 a), b) and c) the populations maximum training, validation and test accuracy as well as the mean and the standard deviation of the test accuracy for config. #7, #8 and #9 are shown, respectively. The gap between the training and the test accuracy at the generation in which the max. accuracy on the test dataset has been encountered the first time is 11.03%, 16.27% and 12.4% for config. #7, #8 and #9, in that order. With config. #7, #8 and #9 the best result on the test dataset has been achieved in generation 991, 965 and 951. This could imply, that a longer execution (more generations) of the NAS might result in a higher test accuracy for all three configurations. After 90 (config. #7), 925 (config. #8) and 113 (config. #9) generations only the siblings of one root parent are present in the population. In summary, led that config. #7 to bigger architectures (regarding the free parameters) accompanied by a higher maximum accuracy on the test dataset compared to config. #6. Nevertheless, the mean accuracy and its standard deviation obtained with config. #6 are superior to those reached with config. #7. With config. #8 it was possible to avoid the premature convergence to siblings of only one root parent. However, poor max. and mean accuracies have been obtained. Furthermore, improvements with config. #9 were missing. Maybe, the initial seed architectures must be trained for more epochs or a larger training set.

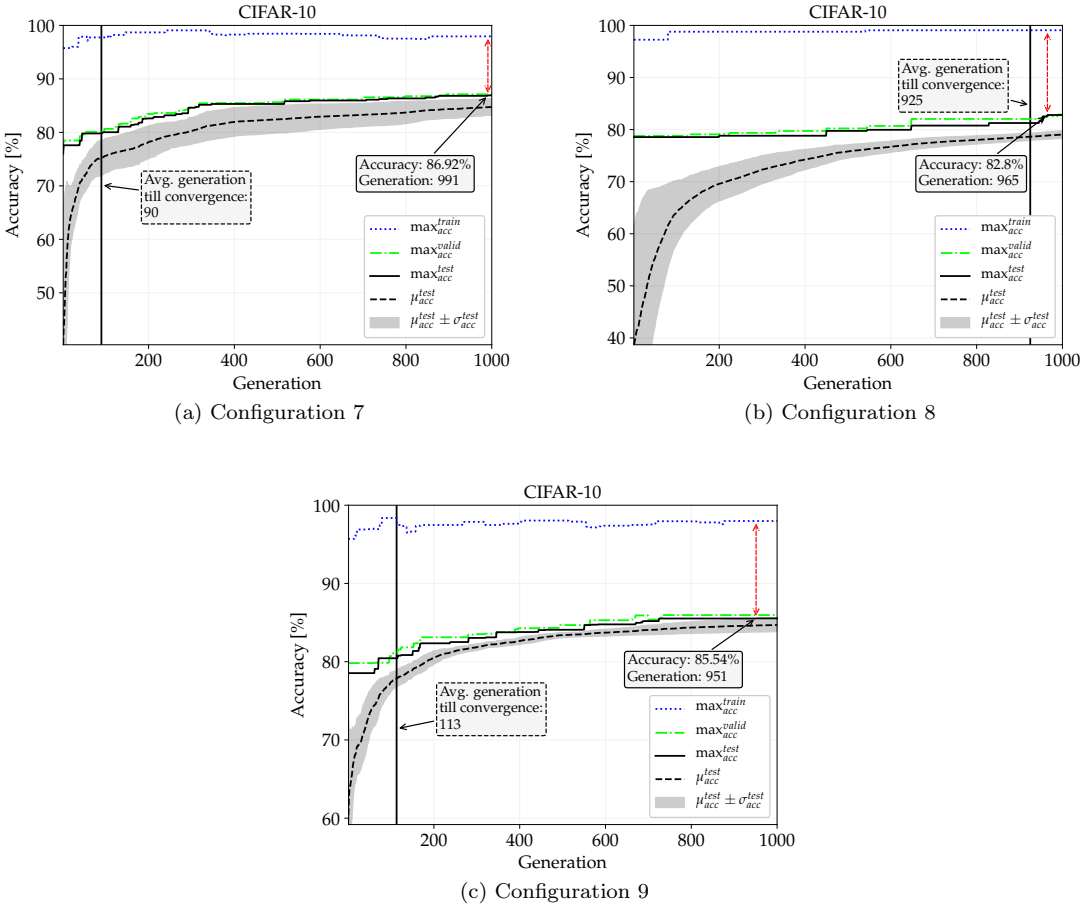
4.3 Recovery from Failure Injection

During the NAS execution one `Fitness Evaluator` `pod` crashed once. It took 43.222 s that `kubelet` discovered that the `pod` failed, a new one became launched and the first log messages of the application appear. Afterwards, we injected failures into the pods of the four microservices explained in Sec. 3.2. For each microservice we simulated 1,000 times a crash of its pods – one at a time. We measured the duration between failure injection and till a new pod is launched, runnning in the state `Running` and all containers in the `pod` have the ready state. The results are summarized in Tab. 4.

Table 4: Mean (μ) duration and its standard deviation (σ) to recover from simulated pod failure. Statistics are calculated by 1,000 pod failure simulations per microservice.

Microservice	$\boldsymbol{\mu}$ **[s]**	$\boldsymbol{\sigma}$ **[s]**
Fitness Evaluator	3.272	2.011
Genetic Algorithm	1.749	0.339
Database	55.744	27.31
Message broker	43.473	15.342

Due to the fact, that the Genetic Algorithm microservice did not reach any resource limitations and the higher level `Kubernetes` resource – the `job` – does not pin a fixed identity to the corresponding `pod`, it was able to recover the fastest from the simulated failures with about 1.749 ± 0.339 s. Furthermore,



(a) Configuration 7

(b) Configuration 8

(c) Configuration 9

Figure 6: Maximum accuracy achieved each generation on training, validation and test dataset (blue dotted, green dash-dotted, black solid line respectively). Black dashed line: mean maximum accuracy per individual in each generation, Gray area: standard deviation, Vertical line: average generation, from which only the gens of one root parent are present, Red arrow: Difference between the max. test and training accuracy in the corresponding generation.

the container images were already downloaded on each worker node in the cluster. If the image is not present on the selected new host node, the recovery time might increase depending on the image size as well as the network infrastructure. The `Fitness Evaluator` microservice recovered from failure in about 3.272 ± 2.011 s. Compared to the recovery time stated earlier, the failure detection of `kubelet` and the creation of a new `pod` took about 40 s longer. When the `pod` failed first, the other compute nodes were busy and creation of the new `pod` needed to wait till the failed pod is successfully terminated and the consumed resources are freed. During our experiment, other nodes have been available so that the creation of the `pod` is undelayed by resource limitations. In another experiment, we performed 200 failure injections and limited the number of available nodes to two. The recovery time increased to 20.725 ± 13.22 s. The `RabbitMQ` message broker needed about 43.473 ± 15.342 s to detect `pod` failure and to create a new pod. Due to that a `sts` waits for a successful termination of the failed `pod` till it starts the new `pod`, the recovery times are much higher. The PostgreSQL `database` demanded about 55.744 ± 27.31 s to recover from failure. This might have the same reason as mentioned for the message broker. Besides, the database microservice comprises two containers, as a consequence it takes longer till both containers are ready, especially because of there internal dependencies. It must be mentioned, that other configurations of the microservices as well as the

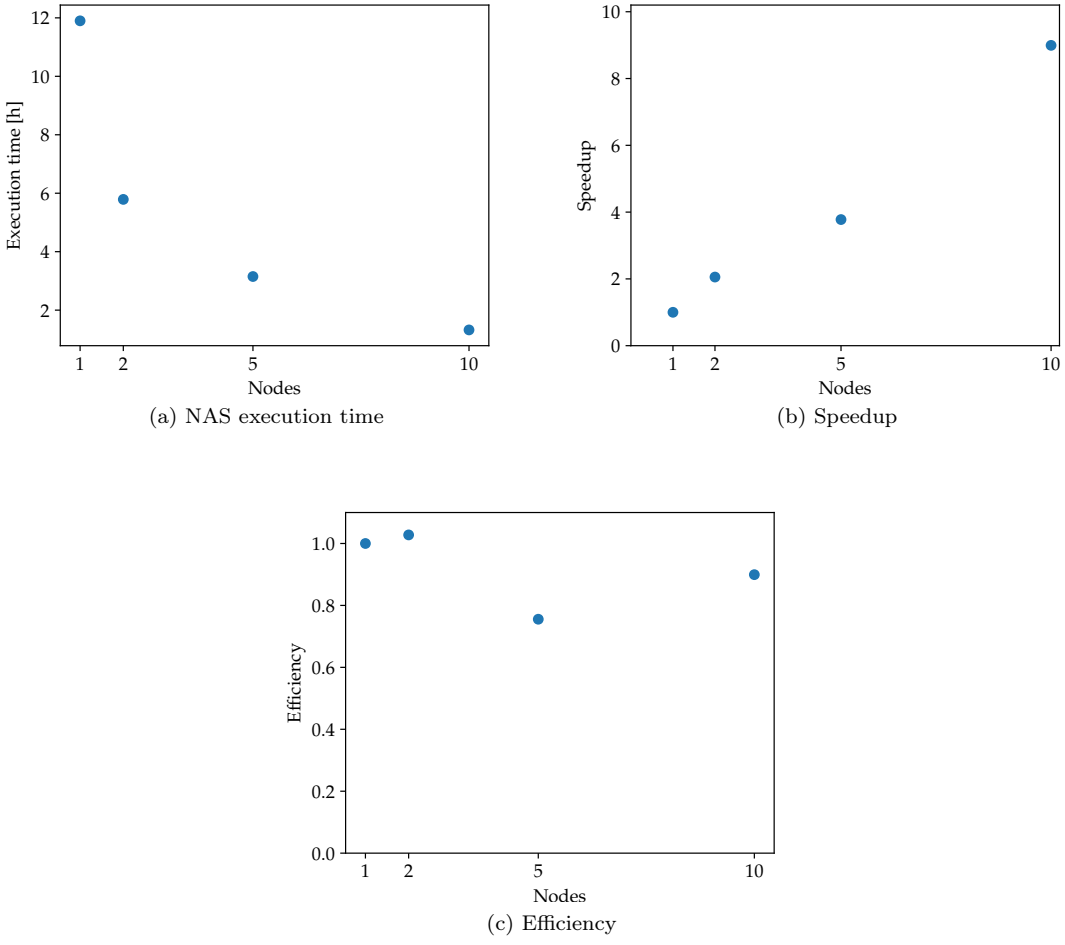
cluster could lead to a faster recovery, perhaps with the trade-off unnecessary overhead regarding healthy checks etc.

4.4 Scalability and Hardware Utilization

In this subsection, first the scalability of the proposed NAS is analyzed. Subsequently, the utilized hardware during NAS execution is described and discussed.

4.4.1 Scalability

We employed $n \in [1, 2, 5, 10]$ subpopulations. Each run of the NAS with n subpopulations is performed five times. The whole population of each NAS execution sums up to 100 individuals. Furthermore, the whole population is evolved by 500 generations, i. e. a run consisting of five subpopulations, each covering 20 individuals and evolved for 100 generations. Therefore, we investigate strong scalability. Figure 7 shows the metrics measured and calculated.



(a) NAS execution time

(b) Speedup

(c) Efficiency

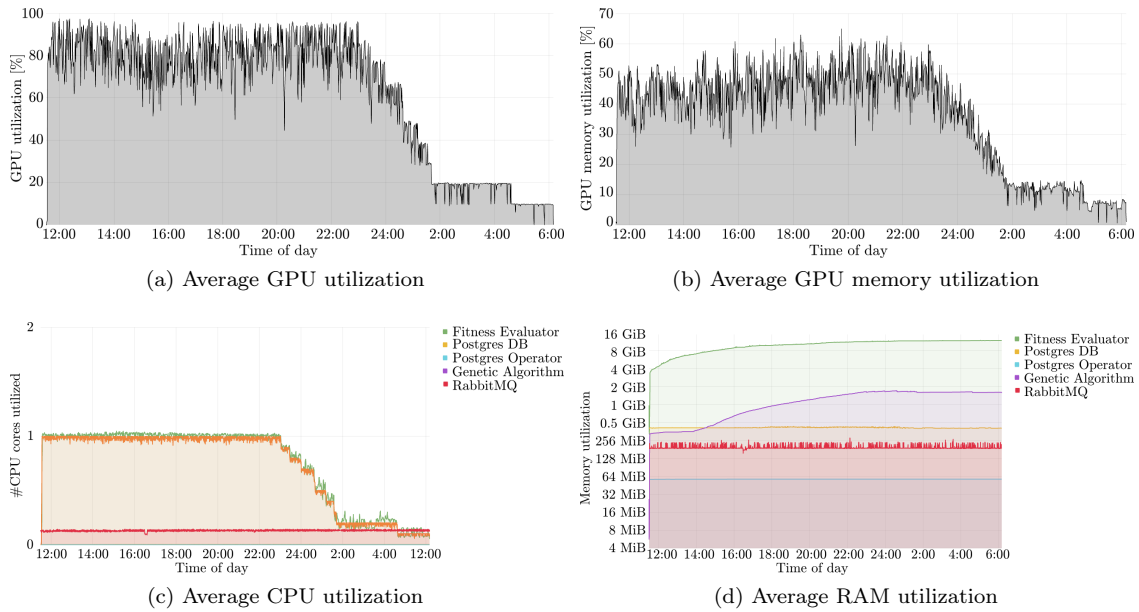
Figure 7: Scalability metrics

In Fig. 7 (a) is the execution time of the fastest NAS run over the number of compute nodes (represents also the number of subpopulations) illustrated. Figure 7 (b) presents the speedup over utilized node count. The speedup increases nearly linear by adding more nodes. Note that the speedup over the number of compute nodes does not proportional increase. Furthermore, in Fig. 7

(c) we observe that the efficiency can not hold constant over the number of nodes. This might lead to the conclusion that the proposed approach is scalable, but not strong scalable. Nevertheless, the utilized hardware, especially the GPU are not on all worker machines the same. This might effect the speedup as well as efficiency calculation. Due to the random nature of the Genetic Algorithm it is not guaranteed that the problem size remained fixed during our investigations. In the above consideration we create 600 individuals and evaluate their fitness. These individuals might be from run to run more complex and their fitness evaluation are computational more demanding.

4.4.2 Hardware Utilization

For the investigation of the utilized hardware during the NAS execution, we performed ten runs of our NAS in parallel so that each compute node of our cluster was occupied. Each NAS has been executed with a population size of 100 individuals and 500 generations. The averaged consumed hardware is illustrated in Fig. 8.



(a) Average GPU utilization

(b) Average GPU memory utilization

(c) Average CPU utilization

(d) Average RAM utilization

Figure 8: Hardware utilization

Figure 8 (a) shows the average GPU utilization. The average GPU utilization in this figure sums up to 61.79%. This value is low, because of the last few hours of the algorithm, when a decremented number of nodes were busy, caused by already finished NAS runs. By only considering the period when all nodes were busy, the average GPU utilization increases to 81.46%. In Fig. 8 (b) the average GPU memory utilization is presented. The average GPU memory utilization in this time period equals 35.68%. This value is also influenced by the last few hours of the algorithm, when some NAS runs already completed their execution. If exclusively time is analyzed, when all nodes were busy the average GPU memory utilization elevates to 46.19%. The average CPU utilization of each microservice (`Fitness Evaluator`, `Postgres` database + `Postgres operator`, `Genetic Algorithm`, `RabbitMQ` message broker) is shown in Fig. 8 (c). Due to the small dataset size, the whole dataset is stored in GPU memory so that multithreading for loading the data is not necessary and only one core is utilized in average by the `Fitness Evaluator` and the `Genetic Algorithm`, as long as we only consider the time period till no NAS run has finished. Afterwards, the average CPU utilization drops to 0.753 and 0.731 cores respectively. Independent of the number of NAS executions, the average CPU consumption by the `PostgreSQL` database and its operator sums up to 0.001 cores and by the `RabbitMQ` message broker to 0.130 cores. The average RAM utilization is illustrated by Fig. 8 (d). Because of the configuration of the NAS, that nether the

`Fitness Evaluator` nor the `Genetic Algorithm` finish automatically, the average RAM consume of both microservices did not drop till the end. During the considered period of time, the average RAM consume of the `Fitness Evaluator` is 9.972 GiB, the `Genetic Algorithm` is 1.149 GiB, the `PostgreSQL` database and its operator sum up to 475.843 MiB and the `RabbitMQ` message broker is 194.086 MiB.

5 Conclusion and Future Works

First, we highlighted the importance of the proposed research and gave an overview on related work. Subsequently, the scalable and highly available multi-objective neural architecture search algorithm has been introduced. Thereby, the hard- and software setup of the cluster as well as each micro service have been detailed. Furthermore, we reported our achieved test accuracy on MNIST, Fashion-MNIST and CIFAR-10 dataset (99.75%, 94.35% and 89.90%) and compared the results with other models and outcomes of various NAS algorithms. The evolved, best performing architectures can be downloaded from [31]. Besides, we considered the aspect of high availability of each component regarding recovery time from failure. Furthermore, we have proven that our approach is scalable. The scalability could be improved by generating more siblings during one generation, as e. g. in case of NSGA-II [26], NSGA-III [27], SEPA-2 [28] etc. Subsequently, we investigated nine different configurations of the Genetic Algorithm. Each setting is tested three times. The maximum test accuracies as well as the mean accuracies, loss and number of free parameters are investigated. The results showed, that constraining the kind of factorization of the inception layer seems to be favourable. An improvement by techniques like a minimal age for being a parent of an individual and k-fold cross-validation were missing. Too large populations should be avoided with our approach. The approach with the seed architectures for the initial population might be interesting to further analyze. The adjustment of the training epochs, the fraction of training dataset etc. could be adjusted.

Left for our future works is the application of the NAS algorithm on other datasets and to improve the results. Furthermore, we want to reduce the computational demands of the proposed NAS algorithm.

References

[1] A. Baldominos, Y. Saez, and P. Isasi, "Evolutionary convolutional neural networks: An application to handwriting recognition," *Neurocomputing*, vol. 283, pp. 38–52, 2018.

[2] N. Mitschke, M. Heizmann, K. Noffz, and R. Wittmann, "Gradient based evolution to optimize the structure of convolutional neural networks," in *25th IEEE International Conference on Image Processing (ICIP)*, 2018, pp. 3438–3442.

[3] S. Litzinger, A. Klos, and W. Schiffmann, *Compute-Efficient Neural Network Architecture Optimization by a Genetic Algorithm*, 2019, vol. 11728 LNCS.

[4] Z. Zhong, J. Yan, and C. Liu, "Practical network blocks design with q-learning," *CoRR*, vol. abs/1708.05552, 2017. [Online]. Available: http://arxiv.org/abs/1708.05552

[5] E. Real et al., "Large-scale evolution of image classifiers," *34th International Conference on Machine Learning, ICML 2017*, vol. 6, pp. 4429–4446, 2017.

[6] H. Liu et al., "Hierarchical representations for efficient architecture search," *international conference on learning representations*, 2018.

[7] D. Whitley, S. Rana, and R. B. Heckendorn, "The island model genetic algorithm: On separability, population size and convergence," *Journal of Computing and Information Technology*, vol. 7, pp. 33–47, 1998.

[8] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[9] H. Xiao, K. Rasul, and R. Vollgraf. (2017) Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

[10] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[11] A. Klos, M. Rosenbaum, and W. Schiffmann, "Scalable and highly available multi-objective neural architecture search in bare metal kubernetes cluster," in *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2021, pp. 605–610.

[12] Y. Jaâfra, J. L. Laurent, A. Deruyver, and M. S. Naceur, "A review of meta-reinforcement learning for deep neural networks architecture search," *CoRR*, vol. abs/1812.07995, 2018. [Online]. Available: http://arxiv.org/abs/1812.07995

[13] P. Ren, Y. Xiao, X. Chang, P.-y. Huang, Z. Li, X. Chen, and X. Wang, "A comprehensive survey of neural architecture search: Challenges and solutions," *ACM Comput. Surv.*, vol. 54, no. 4, May 2021. [Online]. Available: https://doi.org/10.1145/3447582

[14] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *Journal of Machine Learning Research*, vol. 20, no. 55, pp. 1–21, 2019. [Online]. Available: http://jmlr.org/papers/v20/18-598.html

[15] P. Yotchon and Y. Jewajinda, "Hybrid multi-population evolution based on genetic algorithm and regularized evolution for neural architecture search," in *17th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, 2020, pp. 183–187.

[16] J. Hajewski and S. Oliveira, "A scalable system for neural architecture search," in *10th Annual Computing and Communication Workshop and Conference (CCWC)*, 2020, pp. 0053–0060.

[17] F. Assunção, N. Lourenço, P. Machado, and B. Ribeiro, "Denser: deep evolutionary network structured representation," *Genetic Programming and Evolvable Machines*, vol. 20, no. 1, p. 5–35, Sep 2018. [Online]. Available: http://dx.doi.org/10.1007/s10710-018-9339-y

[18] A. Marchisio et al., "Nascaps: A framework for neural architecture search to optimize the accuracy and hardware efficiency of convolutional capsule networks," *Proceedings of the 39th International Conference on Computer-Aided Design*, Nov 2020. [Online]. Available: http://dx.doi.org/10.1145/3400302.3415731

[19] B. Ma, X. Li, Y. Xia, and Y. Zhang, "Autonomous deep learning: A genetic dcnn designer for image classification," *Neurocomputing*, vol. 379, pp. 152–161, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231219313797

[20] C. Szegedy et al., "Rethinking the inception architecture for computer vision," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2818–2826, 2016.

[21] M. Luksa, *Kubernetes in Action*. Manning Publications, 2018, ISBN: 9781617293726. [Online]. Available: https://books.google.de/books?id=8bE5MQAACAAJ

[22] "Kubernetes documentation," https://kubernetes.io/docs/home/, accessed: 2021-03-30.

[23] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST database of handwritten digits," http://yann.lecun.com/exdb/mnist/, accessed: 2021-02-13.

[24] "Fashion-mnist," https://github.com/zalandoresearch/fashion-mnist, accessed: 2021-02-13.

[25] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 6105–6114. [Online]. Available: https://proceedings.mlr.press/v97/tan19a.html

[26] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[27] K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, 2014.

[28] E. Zitzler, M. Laumanns, and L. Thiele, "SPEA2: Improving the strength pareto evolutionary algorithm," in *EUROGEN 2001. Evolutionary Methods for Design, Optimization and Control with Applications to Industrial Problems*, 2002, pp. 95–100.

[29] N. Beume, B. Naujoks, and M. Emmerich, "Sms-emoa: Multiobjective selection based on dominated hypervolume," *European Journal of Operational Research*, vol. 181, no. 3, pp. 1653–1669, 2007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221706005443

[30] E.-G. Talbi, S. Mostaghim, T. Okabe, H. Ishibuchi, G. Rudolph, and C. A. Coello Coello, *Parallel Approaches for Multiobjective Optimization.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 349–372. [Online]. Available: https://doi.org/10.1007/978-3-540-88908-3_13

[31] A. Klos, "Example application of the best performing, evolved neural networks during neural architecture search," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4654465

A Configuration of the Neural Architecture Search

Table 5: Configuration of the proposed NAS. BN denotes batch normalization.

Subpopulation	**Value**
Subpopulation count	5
Number of Individuals	20
Migration period	Every 20 generations
Migration	**Value**
Potential migrants selection	30% with highest fitness value
Final migrant selection	Roulette wheel: 5% of population size
Fitness measure	Accuracy on test dataset till threshold exceeded by worst individual, then number of free parameters
Evolution	**Value**
Initial population	1,000 random mutations
Mutation operator: Probability	Insert layer: 20% Switch two layers: 10% Delete layer: 20% Adjust layer: 40%
Crossover rate	10%
Crossover kind	Single point
Termination criterion	1,000 generations
Parent selection	Roulette wheel
Replacement strategy	Individual with lowest fitness score, if the new individual has higher fitness value
Fitness measure	Same as for migration
Activation function	LeakyReLU
Regularization	Dense: Dropout: 0.5 Conv: BN: Momentum:0.99
Training parameter	**Value**
Number of epochs	100
Early stopping	No improve after six epochs
Loss function	Mean squared error
Batch size	256
Optimizer	Adam
Learning rate	10^{-3}
Weight decay	10^{-5}
Dataset	**Value**
Augmentation	Standardization: Mean: 0, Deviation: 1
Fitness threshold	MNIST: 99.6% Fashion MNIST: 94% CIFAR10: 90%
Training/Validation ratio	10/1

B Best Evolved Architectures

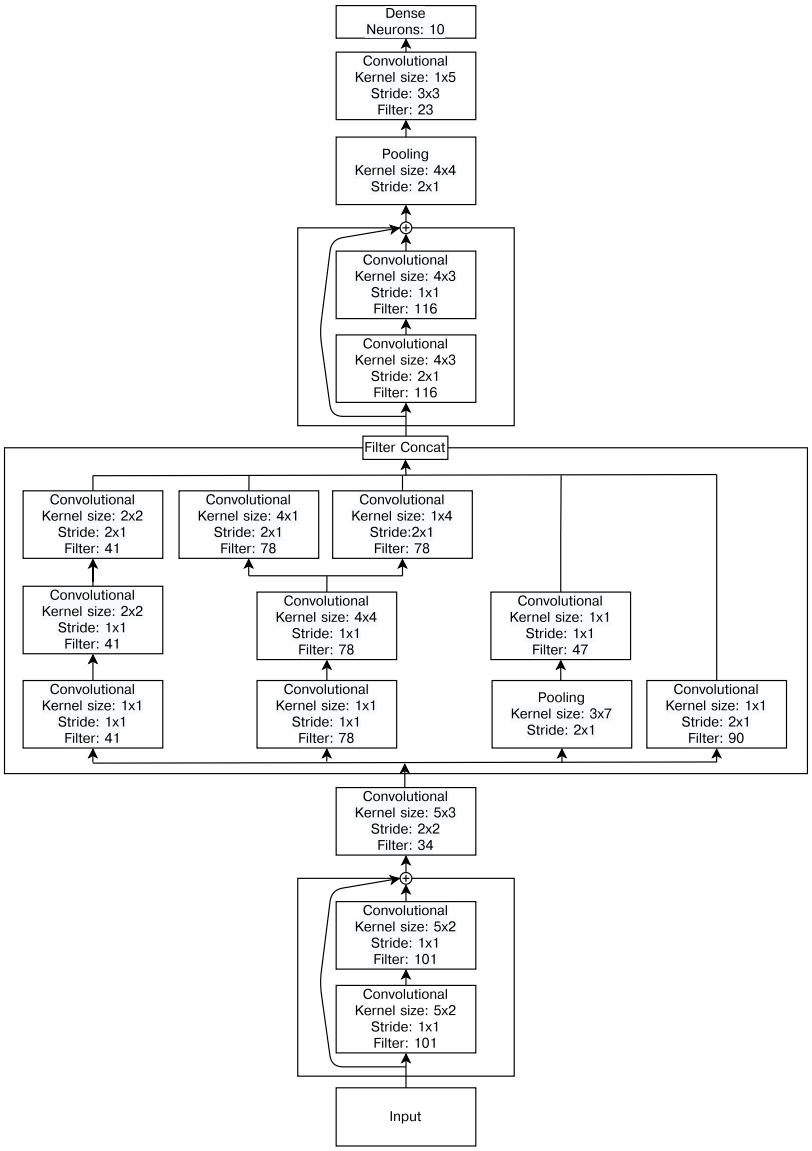


Figure 9: Best neural network architecture evolved for MNIST dataset.

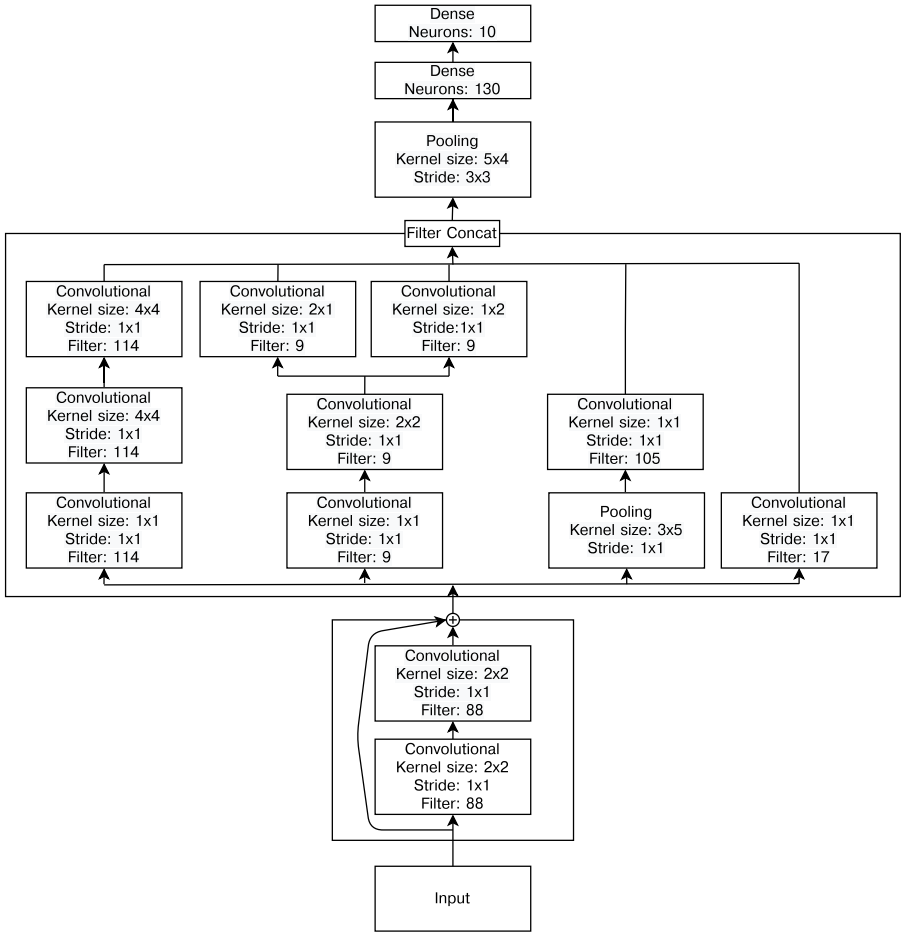


Figure 10: Best neural network architecture evolved for Fashion-MNIST dataset.

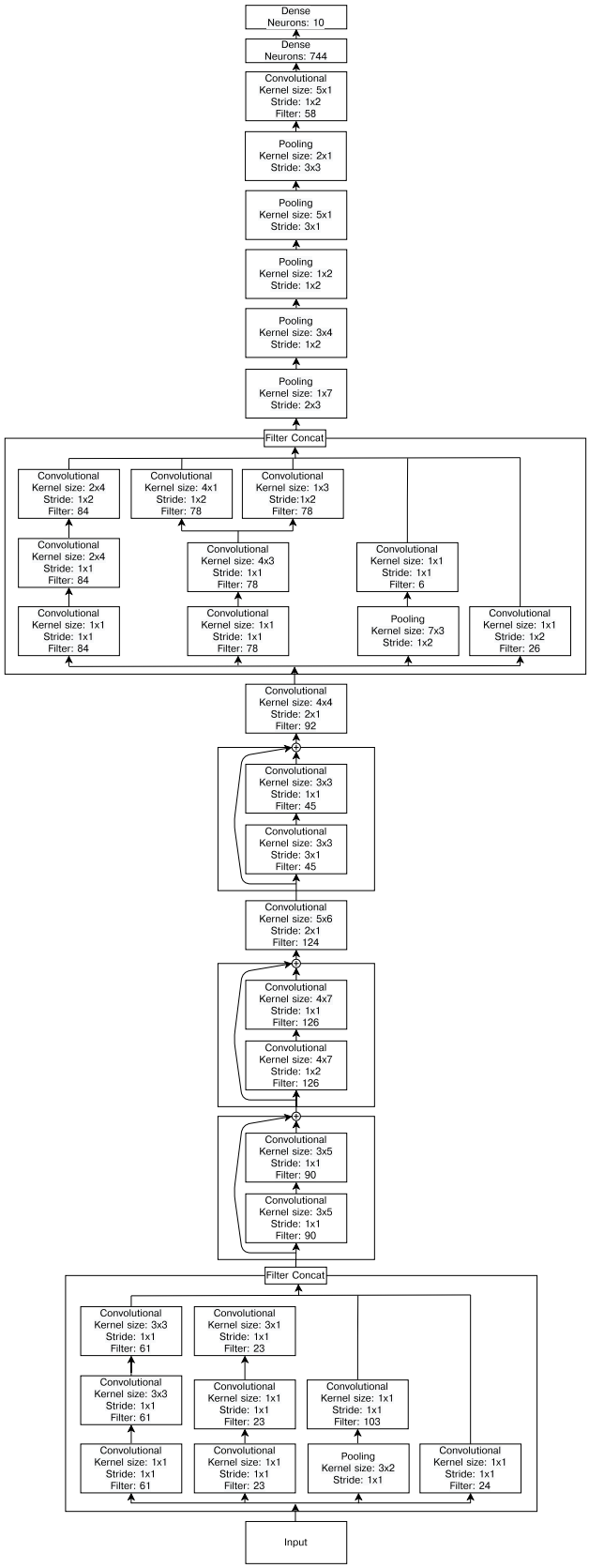


Figure 11: Best neural network architecture evolved for CIFAR-10 dataset.