ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Mutual information for fine-grained network analysis in neural architecture search

Bachelor Thesis

Ferdinand Kossmann

August 30, 2019

Advisors: Prof. Dr. Andreas Krause, Johannes Kirschner

Department of Computer Science, ETH Zürich

Abstract

With the rise of neural networks, there has been a focus shift from feature design to architecture design. Recent research suggests to automate neural architecture search (NAS) using search methods like Bayesian Optimization, Reinforcement learning or evolutionary algorithms. Given a data set, these methods are applied to search for a (local) maximum on the function that maps the architectures to an accuracy on the data set. Some of these searches have already delivered architectures, that compete with state-of-the art human-generated ones.

However, the above-mentioned methods are generally expensive. This also comes from the fact that they evaluate neural networks at the granularity of the whole network: Evaluating a network by a metric like accuracy doesn't inform about how well individual parts of the network perform and what parts need to be changed.

In previous research, neural networks have already been analyzed using mutual information. Because the mutual information measures arbitrary dependencies between random variables, it is suitable for assessing the "information content" of the layers in complex classification tasks. However, in this work we will show that mutual information per se is not sufficient for neural architecture search tasks - at least not when estimated with MINEs, a current state of the art method.

The above finding is shown in two experiments. In the first experiment, we try to predict a distribution over good architectures on a data set given an architecture and its MI on the data set. In the second experiment, we try to predict good sampling distributions for Hyperbands, given an architecture and its MI on the data set the sampled architectures should perform well on. In both experiments, a neural network predictor leveraging the MI was not able to compete with simple base lines. We suggest that this is due to mutual information not being expressive enough for such complex tasks.

Contents

Chapter 1

Introduction

1.1 Neural Architecture Search

With the rise of neural networks, there has been a focus shift from feature designing to architecture designing [1]. However, finding a good neural network architecture for a given data set is still a non-trivial and time consuming task. It is traditionally performed by a human expert who tries out several architectures until deciding for one that delivers satisfying results. More recent research suggests to tune neural network architectures using automated methods (AutoML). The automation of this task brings two key advantages over human search:

1. Scaling machine learning: The automation of model choice, hyperparameter tuning and training would make neural networks a tool that is easy and fast to deploy for anyone. There is no machine learning specific knowledge needed in order to build a well-tuned NN solution, which would make neural networks more accessible to non-data scientists. The democratization of machine learning is important since the real power of machine learning lies in the fact, that it can be used for thousands of individual use cases. Having an efficient way to deploy machine learning for these use cases is therefore crucial.
2. Beating human search: Using ″intelligent″ and highly optimized automated search methods can outperform human search. An example for that is demonstrated by Google Brain's NASNet, which found an architecture that outperforms state of the art on the Penn Treebank dataset. The architecture doesn't just have significantly lower test perplexity but is also more than 2 times faster than the previous best architecture [1]. Figure 1.1 shows how architectures found with the help of NASNet perform on ImageNet. The most accurate architecture performs slightly below state of the art while, however, being significantly faster

[2].

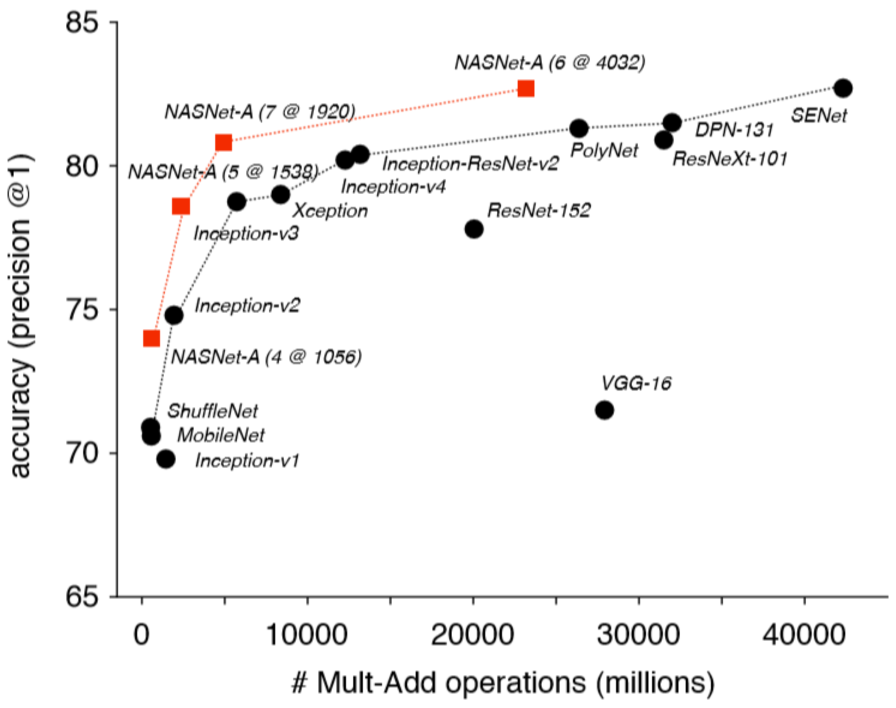


Figure 1.1: Accuracy versus computational demand across top performing CNN architectures on ImageNet 2012 ILSVRC.

1.2 Learn Neural Architecture Search

Most published neural architectures search (NAS) methods focus on suggesting search methods, that find a (local) maximum of the function that maps architectures to accuracy on a given data set. Some of these search methods are listed in section 1.3.

Although little research is done in that direction, there is evidence, that there are certain recurring principles that hold for a NAS on any data set. This becomes evident when observing that many good architectures on different data sets share certain properties, that people found heuristics on how to find a good architecture [3], and that one can find partial architectures, that can be stacked together in different ways to work well on different data sets [2].

By exploiting these underlying principles, one could custom tailor a NAS that is more efficient than a search with a generic search method. One of the difficulties in building such a search lies in capturing the underlying patterns. In this work, we will investigate if mutual information can be used as a metric to capture these patterns. Mutual information is an interesting candidate because it measures arbitrary dependencies between random variables, which makes it suitable for assessing the "information content" of the layers in complex classification tasks [4].

More specifically, we will look at the mutual information between layer activations in a neural network and a target vector. High mutual information between a layer and the target vector means that the layer is able to capture essential information needed for classification. Lower information however tells us the layer is not able to do so. Using mutual information in our search could therefore have two advantages: First, it sets an architecture into the context of a data set, giving us the needed information to evaluate patterns in architecture search independent of the data set. Second, compared to using other common metrics like accuracy, mutual information is more fine-grained, giving feedback about each layer and not only about the network as a whole.

We will evaluate mutual information by looking how much a neural net predictor can learn from it in an attempt to solve two NAS relevant tasks. The tasks as well as the predictor are more closely described in section 2.

1.3 Related work

In this chapter, we will give an overview over similar work on both, neural architecture search as well as using mutual information to evaluate neural networks.

1.3.1 Neural architecture search

In recent years, there has been an increasing research interest into NAS. Most research focuses on how to search the function that maps architectures to accuracy. Approaches include the following:

Bayesian Optimization

Searching the function with Bayesian Optimization (BO). This has the advantage of being sample efficient since the amount of samples taken is minimized. Evaluating the function is very expensive because it means training a neural network to convergence. However, BO has the disadvantage of searching a fixed length search space. It is not flexible and cannot predict variable-length features like a search with an RNN, described in the section below, could. This is a limitation when searching for architectures with varying amount of layers. [1] BO of the currently suggested form also needs to be restarted from scratch for every new data set. Early work on NAS with BO includes [5]. More recent work includes [6], [7], [8].

Reinforcement learning

There has been research by Google to train a controller neural network to specify a good architecture given a data set [1]. The controller is trained using policy gradients and a reward signal that is composed of the accuracy on the data set subtracted by a baseline function to reduce signal variance. By using an RNN as controller, the search space can be implemented to be very flexible, as shown in Figure 1.2. This method has achieved impressive results: It has delivered architectures that compete with state of the art ImageNet architectures (see figure 1.1) and has broken state of the art for the Penn Treebank data set. However, training the controller required 800 GPUs and training 12’800 child models [9].

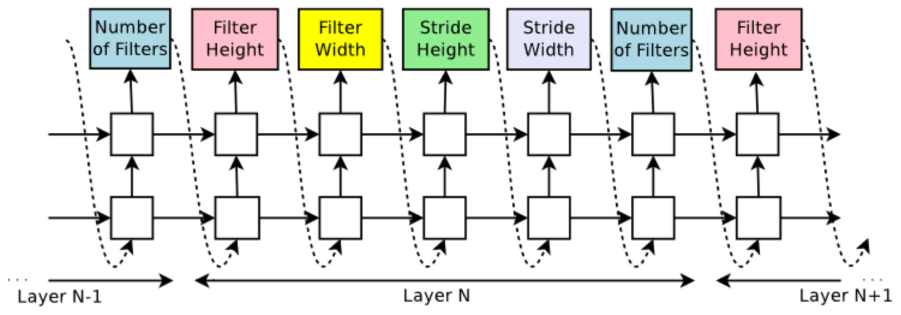


Figure 1.2: Specify varying amount of layers using an RNN

In a second paper, the authors suggest to learn transferable partial architectures, so-called ”cells”, that can be stacked together to form bigger architectures [2].

The search suggested by the authors is generally restarted for new data sets. Since the controller is not given any input, it can only adapt to new data sets by learning new parameters. It would be interesting to build a similar

[1]The limitation can be addressed by workarounds like searching over architectures with a lot of layers and including a stop symbol as a prediction possibility. The predicted model will then correspond to the architecture specified up until the stop symbol.

controller and provide it with input that allows to adapt predictions to different data sets without additional learning. If the controller RNN is implemented as a sequence-to-sequence model, this change could be understood as adding the encoder part to the already existing decoder part.

Evolutionary search

Using evolutionary algorithms for NAS has been suggested early on by [10]. The method is still popular with many project being done, including a large scale search by Google Brain [11] and a reinforced search by [12], which found a a state-of-the-art ImageNet classifier for "mobile setting" (image size 224x224 and less than 600M multi-add operations). However, evolutionary search also has the disadvantage of needing to be restarted from scratch for every new data set.

Inception Layers and Convolutional Neural Fabrics

Inception layers are not a search method but are mentioned here as an orthogonal approach to making architecture decisions [13]. They are widely popular when designing CNNs. When the architect has a hard time deciding on what layer to insert next (i.e. what kernel size or pooling layer), an inception model let's the architect just chose all options he has in mind. In the inception layer, all options are evaluated (and back-propagated) and are then concatenated and fed as an input to the next layer. To reduce computational cost, a "bottleneck layer" is often inserted before the inception layer. The bottleneck layer downscales the origin input to make the evaluation of the several options cheaper. It is found in practice, that the bottleneck layer seldomly affects network performance (when chosen reasonably).

Inception layers are a less powerful concept than NAS, since they do not specify a whole architecture but rather let the architect use several architecture options at once [14]. Furthermore, the additional options cause additional computation cost and can thus make the specified architecture suboptimal.

Convolutional Neural Fabrics take that concept to specify whole architectures. Within a fabric, there exist a range of different scaling and convolution options. The fabric creates a grid that systematically contains all combinations of a certain length of these options and therefore does not only specify one single architecture, but basically trains a model as an ensemble accross many architectures. A big enough fabric can express a huge among to different CNN architectures. Furthermore, the fabric can be pruned. However, such a fabric is still not an optimal network architecture and computationally very expensive (that problem cannot be fully addressed by pruning). Appart from that, Convolutional Neural Fabrics can only specify architectures of a certain length. Convolutional neural fabrics can be transfer learned when confronted with a new, similar dataset according to [15].

Hyperband

There are basic search methods which aim at being very fast at the cost of not even finding a local maximum. One of them is Hyperband which will also be used in this work. Hyperband takes n input architectures and presents a heuristic for selecting the best k architectures out of them [16]. In each iteration of Hyperband, the architectures are trained for a number of epochs. After the training, the architectures are evaluated on a test set and the ones with the worst performance are discarded for the next round. Like this, there's a step-wise reduction of the surviving architectures while at the same time proceeding with training. This is done until only k architectures survive and are fully trained.

1.3.2 Mutual information

There is also previous work which uses mutual information to evaluate the goodness of features. Works include:

Using Mutual information for feature selection

For an initial set F a subset $S \subset F$ should be chosen such that S serves as the best NN input out of all subsets of equal size. The author suggests to use the mutual information between features f and the target classes y for the selection [4]. The features are then chosen greedily based on $I(f;y)$. Doing so, the author achieved similar results to comparable methods in test cases. The suggested method is invariant to input variable transformations. The author suggests that this is an advantage over PCA and Fisher's linear discriminant analysis, although the latter two use mappings instead of selection for dimensionality reduction.

InfoGAN

GANs are a framework for using deep generative models to generate output using a minimax game. The GAN will be fed with noise on which the output depends. This raises the question on how to "wish" for an output. Since the noise is entangled, it's not clear how to change it in order to manipulate the result. The authors of [17] suggest to use a "latent code" that is added to the noise in order to wish for a certain output property. The authors suggest to use a regularization term during training, that maximizes the mutual information between latent code and the output. In initial results, the authors created a 10-state latent code for MNIST: Depending on which code you chose, the GAN will generate a different digit. Figure 1.3 shows their results: In each row a certain noise vector has been held constant, in each column a certain latent code has been held constant. One can see on the right hand side, that choosing different latent codes will lead to different digits being output (there's an error in the second row, last column). As a comparison, on the left hand side are shown the results for a normal GAN that hasn't been trained with the regularization parameter. Even when not labeling the data, there is only a 5% error rate in matching latent codes to the right variable when training with the regularization parameter.

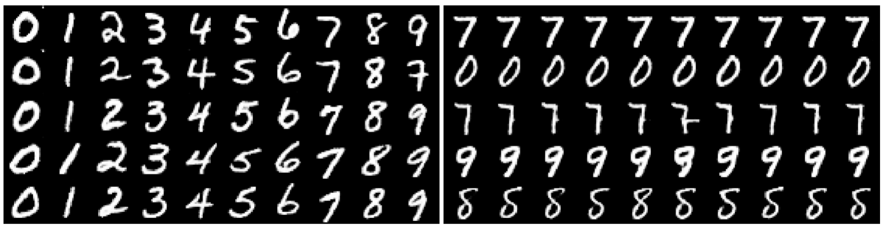


Figure 1.3: Varying the 10-state latent code in the columns leads to different digits being output when training with the regularization parameter (left) while not showing any interpretable changes when training normally (right).

Evolution of Mutual Information over the course of gradient descent

The authors look at the mutual information between the input X and the layer activations, as well as at the mutual information between the layer activations and the output y [18]. As better feature representations are learned, the mutual information between the layer activations and y increases. The authors also claim to see patterns in the evolution of the mutual information between X and the activations. However, latter observation is highly controversial. It has widely been claimed to be irreducible and has later been proven to be resulting from noisy measurements [19]. In own attempts to reproduce the results, I could observe that the mutual information between the layer activations and y going up but couldn't observe the patterns between X and the activations. Especially when using MINEs [20] or a higher binning resolution (i.e. more precise MI estimation than in the paper) the patterns became unapparent.

Chapter 2

Method

We used two different methods to evaluate how much the mutual information between a networkwork's layer activations and a data set's target vector reveals about good architectures on that data set. The following section describes these two methods.

Both methods trained neural network predictors to predict properties of good architectures on a data set. To predict these properties, the neural networks are given an input containing the mutual information of an architecture and a data set. Their task is then to find a mapping between the inputs and the data set properties. Both neural networks received the same input, which is described in section 2.1. The predicted properties as well as the training procedures are described in section 2.2 and 2.3.

2.1 Model input

The mutual information (MI) between two random variables is defined by Definition 2.1.

Definition 2.1. Mutual information. Let X, Y be two random variables on $\mathcal{X}$, $\mathcal{Y}$ and let $P_{X,Y}$ denote their joint distribution and P_X and P_Y their marginals. The mutual information $I(X;Y)$ between X and Y is

$$\begin{aligned} I(X;Y) = H(X) - H(X|Y) &= \sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} P_{X,Y}(x,y) \log \frac{P_{X,Y}}{P_X(x) * P_Y(y)} \\ &= D_{KL}(P_{X,Y}(x,y) || P_X(x) \bigotimes P_Y(y)) \end{aligned}$$

where H denotes the Shannon entropy and D_{KL} denotes Kullback–Leibler divergence.

Let the mutual information between a model architecture A and a data set D be defined by 2.2.

Definition 2.2. Mutual information between an architecture and a data set. Let $\mathcal{A}$ specify a neural network architecture and $\mathcal{D} = \{X, y\}$ a data set with X as input and y as prediction target. Let C_i be the layer activations in layer i when propagating X through $\mathcal{A}$ after training a network with architecture $\mathcal{A}$ on $\mathcal{D}$ until convergence. Then, let the mutual information $I(A; D)$ between $\mathcal{A}$ and $\mathcal{D}$ be defined as

$$I(\mathcal{A}; \mathcal{D}) = \begin{bmatrix} I(C_1; y) \\ \vdots \\ I(C_L; y) \end{bmatrix}$$

where L is the amount of layers contained in $\mathcal{A}$.

An input sample to our experiment predictors contains a random architecture and the mutual information between the architecture and a data set. Definition 2.3 therefore describes an input sample for a data set $\mathcal{D}$. We are hereby interested if a neural network can infer any information on $\mathcal{D}$ just given the architecture and its mutual information to $\mathcal{D}$.

Definition 2.3. Input sample. Let $\mathcal{A}$ be a random architecture. An input sample $x(\mathcal{D})$ is then defined by

$$x(\mathcal{D}) = (\mathcal{A}, I(\mathcal{A}; \mathcal{D})$$

When feeding an input sample to a neural network, we first standardize the sample's mutual information. We hereby standardize every entry of the MI vector according to values obtained from other architectures on the same data set.

This is done because mutual information are highly correlated and thus, mutual information varies greatly between different data sets. By standardizing it, we tell the neural network predictor how big the mutual information is compared to other mutual information vectors on that data set.

In order to learn patterns that hold for any data set, we sampled inputs from different data sets. Section 2.1.2 describes which data sets were used for training and testing. Section 2.1.1 describes how we estimated the mutual information for an architecture $\mathcal{A}$ and data set $\mathcal{D}$.

2.1.1 Mutual information estimation

To estimate the mutual information between activations and target we used the concept of a Mutual Information Neural Estimator (MINE) [20]. MINEs make use of the Donsker-Varadhan representation [21] of Kullback-Leibler divergence (KL divergence). Theorem 2.4 states the Donsker-Varadhan representation.

Theorem 2.4 (Donsker-Varadhan representation)**.** Let $\mathbb{P}$ and $\mathbb{Q}$ be two probability distributions over the same probability space Ω. The KL divergence admits the following representation:

$$D_{KL}(\mathbb{P}||\mathbb{Q}) = \sup_{T:\Omega\to\mathbb{R}} \mathbb{E}_{\mathbb{P}}[T] - \log(\mathbb{E}_{\mathbb{Q}}[e^T])$$

where the supremum is taken over all functions T such that the two expectations are finite.

It follows from Theorem 2.4 that, if we can find a certain function f in the set of all functions T, we can calculate the KL-divergence between any two random variable distributions $\mathbb{P}$ and $\mathbb{Q}$. f hereby needs to be a function that maps the probability space Ω to $\mathbb{R}$ such that $\mathbb{E}_{\mathbb{P}}[T] - \log(\mathbb{E}_{\mathbb{Q}}[e^T])$ is maximized (and the two expectations are finite). Since mutual information can be written as a KL-divergence (see Definition 2.1), we can search a function f for our random variables in order to find the MI.

In order to search for f, MINE uses gradient ascend and approximates f by a neural network. Since the neural network can only represent a subset $\mathcal{F} \subseteq T$ of all functions, the following bound holds:

$$D_{KL}(\mathbb{P}||\mathbb{Q}) \geq \sup_{T\in\mathcal{F}} \mathbb{E}_{\mathbb{P}}[T] - \log(\mathbb{E}_{\mathbb{Q}}[e^T]) \tag{2.1}$$

In our application, the above probability distributions correspond to $\mathbb{P} = P_{X,Y}(x,y)$ and $\mathbb{Q} = P_X(x) \bigotimes P_Y(y)$ where P according to Definition 2.1. We now want to estimate the mutual information for X being layer activations for some layer i and Y being a target vector on some data set. We are given C_i and a target vector y as in Definition 2.2. Note that C_i and y are a list measurements for our variables X and Y.

We define the following bag of tuples given C_i and y:

$$J_i = \{(c_j, y_j)|c_j, y_j \text{ occur in the } j^{th} \text{ row of } C_i, y \text{ respectively}\} \tag{2.2}$$

We also define the bag of tuples

$$M_i = \{(c_j, y_k) | c_j \in C_i, y_k \in y; c_j \text{ and } y_k \text{ are only contained in one tuple in } M_i\} \tag{2.3}$$

that takes the tuple value from from J_i and randomly recombines them to form permuted tuples. This operation is equivalent to randomly shuffling the rows of C_i and then computing J_i again.

Note that J_i and M_i describe measurements that follow probability distributions $\mathbb{P} = P_{X,Y}(x,y)$ and $\mathbb{Q} = P_X(x) \otimes P_Y(y)$ respectively. We can thus empirically approximate 2.1 by 2.4.

$$\sup_{T \in \mathcal{F}} \frac{1}{|J_i|} \sum_{p \in J_i} T(p) - \log\left(\frac{1}{|M_i|} \sum_{q \in M_i} e^{T(q)}\right) \tag{2.4}$$

In summary, MINE gives a lower bound to the mutual information by expressing the supremum function f in theorem 2.4 through a neural network. The neural network is trained with gradient ascend to maximize $\mathbb{E}_{\mathbb{P}}[T] - \log(\mathbb{E}_{\mathbb{Q}}[e^T])$ in order to give a tight lower bound. MINEs are generally found to be accurate in practice.

In this work, we apply the concept of MINEs to fit our use case: Since our data set of data sets consists of image classification tasks, a fully connected MINE suffers from having to deal with high-dimensional layer activations. We therefore used CNN MINEs, MINEs that have convolutional layers.

In order to estimate the mutual information in layer i of a k-layered network-work, we built a MINE with $k - i$ convolutional layers. The output of the convolutional layers is reduced to a one dimensional vector by taking the mean over the 2 additional dimensions. The resulting vector is fed to a fully connected layer and afterwards mapped to a single number t. The convolutional layers as well as the fully connected layer and the mapping to a single number are optimized by gradient ascend on $t - e^t$. The expectation values of Theorem 2.4 are implemented in the sense, that every input is fed through the network and that gradient ascend has to find weights that optimize $t - e^t$ for all inputs.

We trained our MINEs for at least 18 epochs and then until the lower bound increase averaged over two epochs is below a threshold.

2.1.2 Dataset of datasets

To predict neural network architectures for different datasets, we need a dataset of datasets. Our dataset of datasets was curated from 9 standard image classification datasets and consisted of 27 datasets in total:

1. **MNIST:** Classify handwritten digits into 10 classes. Images are 28x28 pixels greyscale. 60000 training samples, 10000 testing samples. 6000 training samples/class. [22]

2. **Fashion MNIST:** Classify clothes into 10 classes. Images are 28x28 pixels greyscale. 60000 training samples, 10000 testing samples. 6000 training samples/class. [23]

3. **EMNIST Letters:** Classify handwritten letters into 26 classes. Images are 28x28 greyscale. 124800 training samples, 20800 testing samples. 4800 training samples/class. [24]

4. **Kuzushiji-MNIST:** Classify selected Japanese Kanji into 10 classes. Images are 28x28 pixels greyscale. 60000 training samples, 10000 testing samples. 6000 training samples/class. [25]

5. **Balanced Kuzushiji-49:** Classify selected Japanese Kanji into 49 classes. Images are 28x28 pixels greyscale. The original dataset is unbalanced but has been balanced for our use: 60000 training samples, 10000 testing samples. 6000 training samples/class. [25]

6. **CIFAR10:** Classify objects into 10 classes. Images are 32x32 pixels rgb. 50000 training samples, 10000 testing samples. 5000 training samples/class. [26]

7. **CIFAR100:** Classify objects into 100 classes. Images are 32x32 pixels rgb. 50000 training samples, 10000 testing samples. 500 training samples/class. [26]

8. **20 subdatasets of Tiny Imagenetwork:** Tiny Imagenetwork contains 200 classes with 500 training images and 100 test images per class. Images are 64x64 pixels rgb. For our purpose, the images were downsampled to 32x32 pixels rgb using Box algorithm. The dataset was split up into 20 datasets each consisting of 10 classes. Subdataset I contains classes 10*I to 10*(i+1) of the origin Tiny Imagenetwork dataset. [27]

The 27 datasets show a high variance in difficulty of classification, with very simple-to-classify datasets like MNIST (state of the art accuracy: 99.79% [28]) to more complex ones like CIFAR100 (state of the art accuracy: 75.72% [29]). Furthermore, there is a high variation in difficulty of recognizing different classes in Tiny Imagenetwork. Figure 2.1 shows example images of the class "school bus", which is considered easy to recognize, and "nail" which is con-

Figure 2.1: Example images of the class "school bus" (left) and of the class "nail" (right)

sidered hard to recognize [30].

20 of the 27 datasets were used for training, the remaining 7 were used for testing. The following datasets were used for testing:

1. CIFAR10,
2. Kuzushiji-49
3. Fashion MNIST
4. 4 subdatasets of Tiny Imagenetwork

2.2 Learning Hyperband posterior distributions

In order to evaluate how much mutual information reveals about a data set for NAS, we tried to predict a probability distribution over good CNN networkwork architectures. More specifically: Given filter options for a finite number of layers, predict how likely it is that a filter option is included in a "good" neural network architecture. We want to find out if such a prediction is possible given a mutual information input sample as described in definition 2.3. The intuition is that the mutual information vector could show where in the architecture information is lost and where important features are learned. Combining this knowledge with the actual architecture, we would be able to predict how many filters a layer should contain.

The probability space over which we predicted, contained 4 layer CNNs that have between 10 and 120 filters per layer, increasing in steps of 5. The predicting model therefore has to predict 4 probability distributions over 23 discrete options. The prediction value for each filter option should represent how likely it is that an architecture with that filter option performs "well" on the data set.

To approximate the ground truth probablity distribution, we used Hyperband [16]. After feeding Hyperband with random input architectures, the architectures that are selected by Hyperband are considered as "good". These architectures form a Hyperband posterior distribution as defined in definition 2.5. The selectivity of Hyperband (how many architectures chosen out of how many inputs) decides on how hard it is to be "good". When Hyperband is fed with enough input architectures, the Hyperband output distributions determine representatively how often a certain filter option was labeled as "good". Big Hyperband input is also important since being labeled as "good" also depends on how good the other input architectures perform. If the set of input architectures only contains bad performing architectures, it is easier to be labeled good. We will call the distribution of filter options in Hyperband outputs the Hyperband posterior distribution.

Definition 2.5. Hyperband posterior distribution. Let A_{post} be a set of n neural network architectures that were selected by a Hyperband instance out of $m > n$ neural network architectures A_{prior}. We then define the Hyperband posterior distribution over layer i as the probability distribution that encodes how many times a filter option is relatively contained in A_{post} in layer i. We define the Hyperband posterior distribution over a whole architecture as the set of Hyperband posterior distributions for every layer in the architecture.

Hyperband evaluated architectures according to a score that considers accuracy and complexity of the model. The score takes the architecture's accuracy on the given data set and penalizes high filter numbers (see definition 2.6).

Definition 2.6. Hyperband score. Let $\mathcal{A} = \{a_1, ..., a_L\}$ be a neural network architecture specified by a list of L filter options for L layers. Let $acc(\mathcal{A}, \mathcal{D})$ be the accuracy of a model with architecture $\mathcal{A}$ on data set $\mathcal{D}$. The Hyperband score $hs(arc, acc(\mathcal{A}, \mathcal{D}))$ for arc on data set $\mathcal{D}$ is then

$$hs(\mathcal{A}, \mathcal{D}) = acc(\mathcal{A}, \mathcal{D}) - \lambda * \sum_{a_i \in \mathcal{A}} a_i$$

where λ is a chosen regularization term.

We encoded Hyperband posterior distributions by counting how often a filter option occurs in the Hyperband posterior distribution. However, we didn't count an appearance as 1 but weighted it according to the Hyperband score of the architecture it appeared in. This should take into account, that being labeled as good by Hyperband also depends on an architecture's competition. Furthermore, we smoothed the Hyperband posterior distribution (see appendix A.3). By doing so, we assume that similar filter options will have similar performance. Smoothing is important because having big many Hyperband inputs is computationally expensive. We want to address that problem by inferring from the goodness of one sample, that similar samples have a higher probability of also being good.

For our predictions, we used a fully connected feed-forward neural network. We trained the neural network using standard supervised learning with back-propagation: We collected training samples of different architectures and scored them and calculated their MI on different data sets. We then collected Hyperband posterior distributions on those data sets. The MI and architecture served as the input, the Hyperband posterior distributions as the target of the supervised learning procedure.

In summary, we want to find neural net parameters θ that minimize metric 2.5, training our predictor f with an MSE loss. We are hereby given our training set $\mathcal{T}$ that consists of tuples containing a input sample inp and a Hyperband posterior distribution $post$.

$$\frac{1}{|\mathcal{T}|} \sum_{(inp,post)\in\mathcal{T}} (f_\theta(inp) - post)^2 \tag{2.5}$$

2.3 Learning Hyperband prior distributions

To prototype a very simple NAS that leverages MI patterns, we also learned Hyperband prior distributions. We define a Hyperband prior distribution as the distribution from which Hyperband inputs are sampled (see definition 2.7). Our goal is to use MI to predict a prior distribution for a data set, such that sampling from that prior distribution maximizes the sum over Hyperband scores of the Hyperband outputs (i.e. the Hyperband summed output score as defined in Definition 2.8).

Definition 2.7. Hyperband prior distribution. Let A_{prior} be the set of input architectures that are given to a Hyperband instance H. We define the distribution of which the architectures in A_{prior} were sampled as the Hyperband prior distribution $\mathbb{H}_{prior}$ for H.

Definition 2.8. Hyperband summed output score. Let A_{post} be a set of architectures selected by a Hyperband instance on data set $\mathcal{D}$. The Hyperband summed output score $HS(A_{post}, \mathcal{D})$ is

$$HS(A_{post}, \mathcal{D}) = \sum_{\mathcal{A} \in A_{post}} hs(\mathcal{A}, \mathcal{D})$$

where $hs(\mathcal{A})$ is the Hyperband score of $\mathcal{A}$ as defined in definition 2.6.

Note that this is only a very simple form of NAS which is not able to specify complex architectures like some of the NAS methods introduced in section 1.3. However, the predicted distribution goes over a probability space that fully specifies neural network architectures. Predicting Hyperband prior distributions, sampling, and then selecting architectures with Hyperband can thus be seen as a very simple form of NAS with a learned component in it.

We train a controller to learn the Hyperband prior distributions using policy gradient learning: Our goal is to find a policy $\pi_\theta(\mathbb{P}|s)$ that maps an input sample to a prior distribution. The predicted distribution maximizes an expected reward R. The optimized reward R is over the Markov Decision Process (S, A, P, R):

- $S = \{(a_{1:L}, I^{(D)}_{a_{1:L}})\}$. The state space consists of tuples made of: 1. CNN architecture $a_{1:L}$ that contains L layers with a_i filters in layer i. 2. Mutual information $I^{(D)}_{a_{1:L}}$ between the activations of architecture $a_{1:L}$ trained on data set $\mathcal{D}$ and the target vector of $\mathcal{D}$.
- $A = \{\{\mathcal{A}_1, ..., \mathcal{A}_n\}\} = \{c\}$. An action is a set of n Hyperband inputs. The probability with which a set is chosen as action is determined by the predicted prior distribution $\mathbb{P}$.
- $P(s'|s, c) = \mathcal{U}$ is uniform. The next state s' is chosen at random independent of the previous state s and the action c. We choose the next state at random and don't let the agent decide on how to transition the state space because our primary goal is to map a state to an optimal prior distribution. We hereby assume, that for every data set the optimal prior distribution is unique. To learn that mapping, we do not need to let the agent decide on how to transition the state space.
- $R(A, \mathcal{D})_{1:M})$. Our reward is the Hyperband summed output score of a Hyperband ran with inputs sampled from action A. In order to evaluate how policy $\pi_\theta(\mathbb{P}|s)$ performs on different data sets, a reward is evaluated on M different data sets. In each round, the Hyperband summed output scores of each of the M different data sets are summed up into one reward. Let A_i be the action samples from $\mathbb{P}_i$, then 2.6 describes the reward function.

$$R((\mathbb{P}, D)_{1:M}) = \sum_{i=1}^{M} HS(A_i, \mathcal{D}_i) \tag{2.6}$$

To reduce the high variance of the reward signal, we subtract a baseline function. The subtracted baseline function is the sum of the output architecture scores from the previous time a given data set has been chosen, summed over all data sets that have been chosen in the round. If $HS(\mathcal{A}_{prev}, \mathcal{D}_i)$ denotes the Hyperband summed output score achieved by the previous action $\mathcal{A}_{prev}$ for $\mathcal{D}_i$, the reward signal including subtraction of the baseline function is given by 2.7.

$$R((\mathbb{P}, D)_{1:M})) = \sum_{i=1}^{M} HS(A_i, \mathcal{D}_i) - HS(\mathcal{A}_{prev}, \mathcal{D}_i) \tag{2.7}$$

The utility function to be maximized during training is thus the following:

$$U(\theta) = \mathbb{E}_D[\mathbb{E}_{\pi_\theta(\mathbb{P}|s)}[R((\pi_\theta(\mathbb{P}|s), D))]] \tag{2.8}$$

Since the reward signal R is non-differentiable, we need to use a policy gradient method to iteratively update θ. For that, we used the REINFORCE rule from Williams [31] [1]:

$$\nabla_\theta U(\theta) = \mathbb{E}_D[\mathbb{E}_{\pi_\theta(\mathbb{P}|s)}[\nabla_\theta \log(\pi_\theta(\mathbb{P}|s)) * R((\mathbb{P}, D))]] \tag{2.9}$$

To approximate the expectation values, we sample M data sets from our data set of data sets $\mathcal{D}$ and we sample K actions from the predicted prior distribution $\mathbb{P}$. The empirical approximate of the gradient in 2.9 is hence given by 2.10.

$$\frac{1}{d} \sum_{d \in \mathcal{D}} \frac{1}{m} \sum_{k=1}^{m} \nabla_\theta \log(\pi_\theta(\mathbb{P}|s)) * R((\mathbb{P}, D)) \tag{2.10}$$

2.10 is differentiable and we can optimize it using gradient ascend.

In summary, Algorithm 1 describes how we learn prior distributions.

Algorithm 1 Training of policy networkwork

```
$\theta \leftarrow$ random initial controller parameters

for $i \leftarrow 1 ... train\_iters$ do
    input_arc $\leftarrow$ random architecture
    datasets[ ] $\leftarrow$ $M$ random data sets

    total_score $\leftarrow$ 0.0
    ds_mi[ ] $\leftarrow$ [ ]
    predictions[ ] $\leftarrow$ [ ]
    sampled_inputs[ ] $\leftarrow$ [ ]

    for $ds \in datasets$ do
        mi $\leftarrow$ MI of input_arc on $ds$
        ds_mi[ ] append mi

        ds_prior $\leftarrow$ prior prediction from controller given (input_arc, mi)
        predictions[ ] append ds_prior
        inputs $\leftarrow$ sample $N$ inputs from ds_prior
        sample_inputs[ ] append inputs

        summed_score $\leftarrow$ do Hyperband with inputs
        total_score += summed_score
    end
    R $\leftarrow$ total_score $- \sum_{ds \in datasets}$ previous summed_scores of $ds$
    loss $\leftarrow$ cross entropy predictions and sampled_inputs
    gradients $\leftarrow$ gradients to loss with (input_arc, mi) tuples as input
    gradients $*=$ R

    $\theta$ += gradients $*$ learning_rate
end
```

Chapter 3

Results

3.1 Posterior distributions

The following section presents the results for predicting Hyperband posterior distributions as described in section 2.2. Given an input sample of the test set, the predictor was not able to deduce properties of the unseen data set. Although he bet some base lines, a simple Gaussian fitted on the average distribution of the training data was able to significantly outperform the neural network predictor.

This shows that Mutual information as provided in this context is not sufficient for deducing properties of unseen data sets. The significance of this experiment is further discussed in section 4.2.

3.1.1 Noise in train and test set

We split 27 data sets into 20 data sets for training and 7 for testing as described in section 2.1.2.

For our training data, we collected 11 training samples per data set, so 220 in total. Each sample consists of an input-output pair as described in 2.1: The input contained a random architecture and the mutual information of the architecture on a random data set (defintion 2.3). The output contained a Hyperband posterior distribution that should approximate a ground-truth distribution over good filter numbers (definition 2.5).

The Hyperband instances chose 4 out of 12 architectures, eliminating 4 architectures in each of two elimination rounds.

The goal of this experiment was to find out whether it is possible to predict the posterior distribution on a data set given the mutual information. However, since a posterior distribution also depends on the random Hyperband inputs, the prediction target is noisy.

This noise is reduced if the Hyperband instances are given many input architectures. The reason for that is, that the probability of one Hyperband instance seeing better input architectures than another Hyperband instance becomes smaller as more input architectures are fed. In fact, when inputting and outputting sufficiently many architectures, the Hyperband posterior distribution will heuristically converge towards the ground truth distribution.

We can take the average pairwise mean squared error between posterior distributions on the same data set to quantify how noisy they are. We generally want noise to be low to provide cleaner targets for training. Still, some noise is acceptable, since we don't expect our predictor to perfectly predict ground truth posterior distributions but just to come close to the noise level introduced by Hyperband. Like the noise levels, the predictions can also be judged according to their mean squared error to the target. This enables a comparison between prediction error and target noise.

In table 3.1 we can see that the noise between the sampled posterior distributions is high. In order to reduce the noise in the training data, we can group several Hyperband posterior distributions together. This follows the intuition of Hyperband posteriors converging towards ground truth distributions as more inputs are given: We added 5 Hyperband posterior distributions together to form one *clean* posterior distribution. This gave us $\binom{11}{5} = 462$ clean posterior distributions per data set. We combined each of these distributions with all input samples from the same data set to form complete, *clean* training samples. This gave us $\binom{11}{5} * 11 = 5082$ training samples per data set or $\binom{11}{5} * 220 = 101640$ training samples in total. The combination of output distributions reduced the noise by a factor of 9, as can be seen in table 3.1.

The test set was created in analogous fashion: We collected 5 samples for each of the 7 data sets. We then combined the 5 posterior distributions of a data set to get a clean posterior distribution. Finally, we combined this clean distribution with each input sample of the same data set. The test set thus contained $\binom{5}{5} * 5 * 7 = 35$ samples.

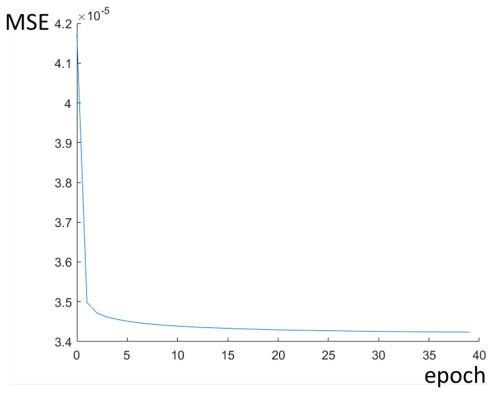


Figure 3.1: Loss against epochs for predictor using MI. Standard deviation in training is given by shaded area but barely visible.

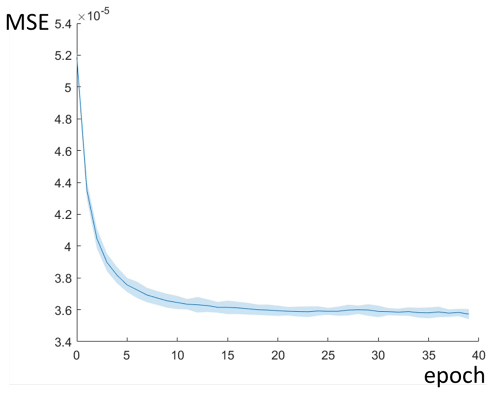


Figure 3.2: Loss against epochs for predictor using accuracy. Standard deviation in training is given by shaded area.

3.1.2 Performance of predictor

To find a mapping between input (architecture and MI) to output (Hyperband posterior distribution) we trained a fully-connected, feed-forward neural network. The specifications of the network can be found in the appendix. The training progress of the network is shown in Figure 3.1.

The performance of the network is given by the mean squared error of its prediction to the target distribution. The performance was compared to 4 baselines:

1. **Uniform distribution:** As an easy baseline, the mean squared error between the target distribution and uniform random is given.
2. **Gaussian fitted on train set:** The output distributions resemble a Gaussian, since similar filter numbers often perform similarly well, whereas extremely small or big filter numbers are often not selected by Hyperband. The small filter numbers deliver poor accuracy and the big filter numbers lead to a high penalty (as described in section 2.1). As a baseline, we fitted a Gaussian that minimizes the mean squared error to the average distribution of all training samples across data sets.
3. **Neural network given only accuracy as input:** Similar to the MI predictor, we fitted a neural network on the training data, however, providing accuracy instead of mutual information as input. The neural network thus received an architecture and its accuracy on the data set as input. The architectures are the same ones used for the MI predictor. The training progress of the network is shown in figure 3.2.
4. **Noise level in output distributions:** We also compare the performance of the predictor to the noise level in the target distributions.

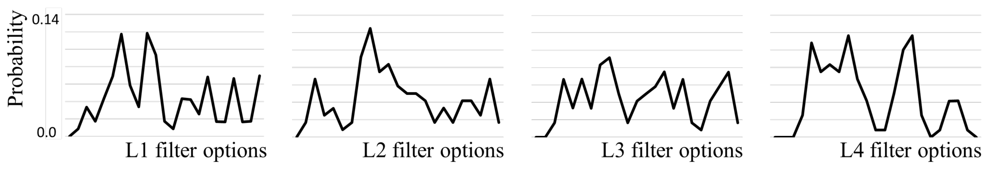
The mean squared errors for predictor and baselines are given in table 3.1. Note that the scale of the entropy is very small since a distribution is represented as a normalized 92 dimensional vector, leading to low entries in each dimension.

Method	Average train set error	Average test set error
Uniform distribution	7.80e-05	8.13e-05
Fitted Gaussian	**6.92e-05**	**6.70e-05**
Neural net with accuracy	3.57e-05 ± 0.03e-05	7.58e-05 ± 0.04e-05
Neural net with MI	3.42e-05 ± 0.04e-05	7.38e-05 ± 0.05e-05
Noise in dirty samples	6.20e-04	6.41e-04
Noise in clean samples	6.84e-05	-

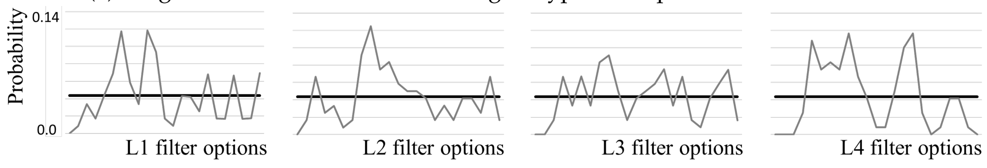
Table 3.1: Mean squared errors of prediction to target posterior.

Table 3.1 shows that the MI predictor only slightly outperformed a predictor that is fed accuracy instead of MI. This shows that MI doesn't seem to deliver valuable insights in this context. It is far away from the noise levels in the target distributions and is even beaten by a baseline that isn't even given any input (the Gaussian baseline).

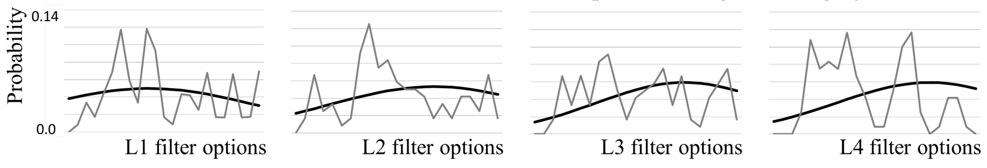
Figure 3.3 gives an example for predictions on a CIFAR10 sample of the test set. Each subgraph contains 4 graphs that represent distributions over layer 1 to 4 from left to right. We can see how some methods picked up on certain characteristics of the output distribution. For example, we can see in 3.3 (e) how the MI predictor picked up on the peak at high filter numbers in layer 2.



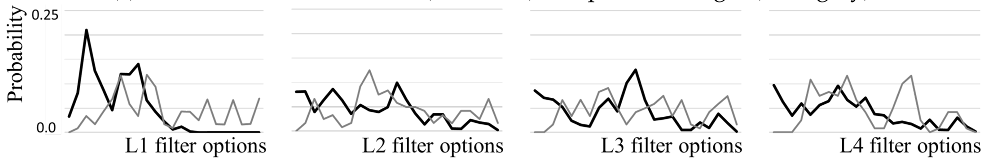
(a) Target distribution from combining 5 Hyperband posterior distributions.



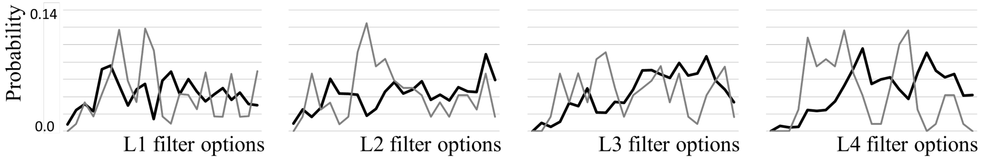
(b) Uniform random baseline (fat, black) compared to target (thin, grey).



(c) Fitted Gaussians baseline (fat, black) compared to target (thin, grey).



(d) Neural network prediction with accuracy as input (fat, black) compared to target (thin, grey).



(e) Neural network prediction with MI as input (fat, black) compared to target (thin, grey).

Figure 3.3: Predictions for a test sample of CIFAR10. The y axis represents a probability value, the x axis represents a filter option from 10 to 120 increasing in steps of 5 from left to right.

3.2 Predicting prior distributions

When trying to predict good prior distributions given an architecture's MI on a data set, we encountered similar results as in section 3.1: The controller couldn't extract meaningful information for a data set based on the MI. The controller wasn't able to predict a prior distribution that is custom tailored for a specific data set but predicted the same prior distribution for all data sets. That generic prior distribution performed okay on all data sets but didn't perform particularly well on any given data set.

We showed this behaviour in two experiments. In both experiments, we sampled Hyperband inputs from a specific distribution and evaluated how good the sum of the outputs was:

1. In the first experiment, we sampled 6 architectures from the predicted prior distribution and 6 architectures uniformly at random. Because the predicted distribution has to compete with the uniform distribution, the controller can only converge to giving a non-uniform output distribution if he finds a pattern that steadily outperforms uniform given a MI on the data set. We assume that, if such a pattern exists, the neural network would be expressive enough to capture it. If such a pattern does not exist, the controller will not be able to maximize a reward signal that leads it to a local maximum of predicting non-uniform distributions: Instead, the reward signal will not lead into any clear direction and will heavily depend on the uniformly sampled inputs.
2. In the second experiment, we only sampled from the predicted prior distribution. This leads to the fact, that the controller will definitely converge towards a non-uniform distribution because the Hyperband output distribution cannot be uniform and thus predicting uniform will always yield a high mean squared error. However, in this experiment, the output distribution is more dependent on the input distribution and doesn't have to compete with a baseline. If the reward signal can be maximized given the MI, the controller could pick up on MI patterns in early stages of training. If such patterns exist, it would take big steps towards a local maximum exploiting these pattern as that yields higher rewards than taking steps in a different direction. However, if the controller is not able to follow a reward signal, it will at some point converge to a distribution that has delivered positive rewards enough times within a certain time interval. When taking enough steps towards that distribution, that distribution will start to slightly dominate the controller output. Since the Hyperband output only depends on the controller output, predicted architectures will thus also start to dominate the Hyperband output. This encourages the controller to convert to that distribution in an attempt to reduce mean squared error.

   Before a distribution dominates the controller's output, the process of slightly converging from the initial uniform distribution to the dominating one resembles a random walk with step probabilities assigned according to how much rewards certain distributions have delivered.

3.2.1 Training of controllers

We trained both controllers for 49 iterations. The loss of the predictors trained for experiment 1 and 2 are shown in figure 3.4 and 3.5 respectively. The loss of the second controller started to drop after 22 iterations. The fact, that the loss of the first controller doesn't decrease, even after 49 iterations, signifies that the controller was not able to reduce mean squared error between prediction and Hyperband output. It was therefore not able to find a way to predict priors that are better than uniform.

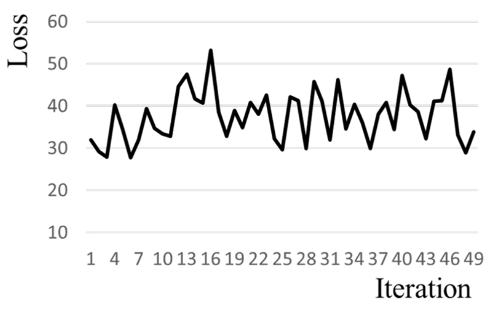


Figure 3.4: Loss of predictor 1 over the course of the training.

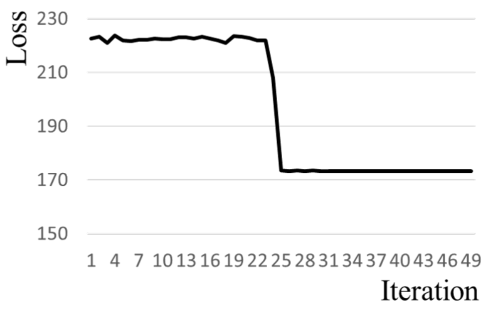


Figure 3.5: Loss of predictor 2 over the course of the training.

Figure 3.6 and 3.7 show the Hyperband summed output scores. Note that scores vary highly from data set to data set. Also note, that in each iteration, 5 Hyperbands were evaluated, which is why Figure 3.6 and 3.7 contain 49*5=245 data points.

A coarse trend of scores increasing over time cannot be observed in either of the graphs. This means that the neural networks were not able to follow a reward signal that maximizes the summed score early on in training.

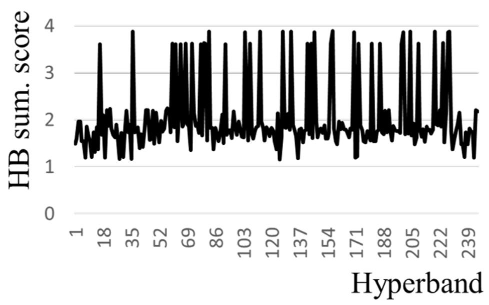


Figure 3.6: Hyperband summed output score of predictor 1 over the course of the training.

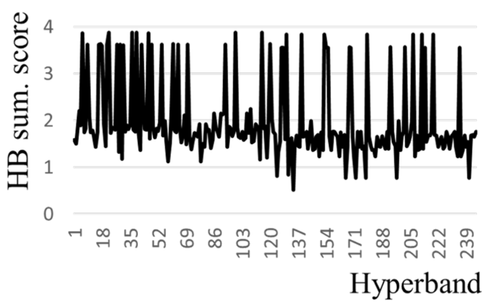


Figure 3.7: Hyperband summed output score of predictor 2 over the course of the training.

Figure 3.8 and 3.9 show the reward of both training processes.

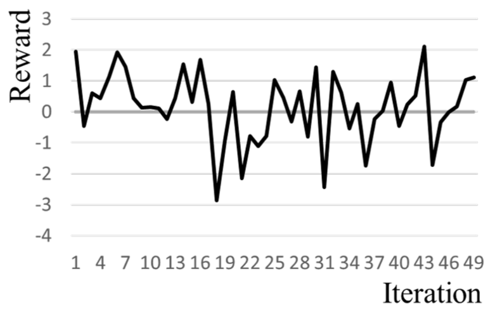


Figure 3.8: Reward of predictor 1 over the course of the training.

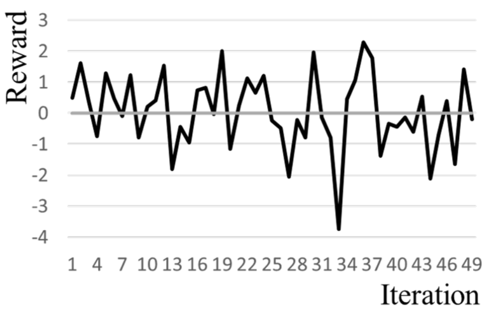


Figure 3.9: Reward of predictor 2 over the course of the training.

The output distributions and performances of the controllers are described more precisely in section 3.2.2.

3.2.2 Performance of controllers

In experiment 1, the controller never predicted a distribution that significantly differed from a uniform one. Figure 3.2.2 shows the output distribution at 3 point in training: In the first iteration, in the middle of the training course and in the last iteration.

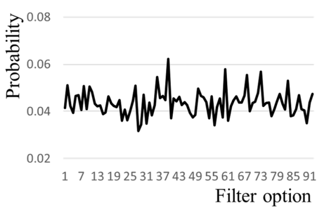


Figure 3.10: Controller prediction in iteration 1.

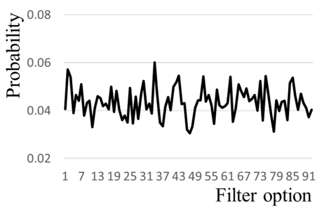


Figure 3.11: Controller prediction in iteration 25.

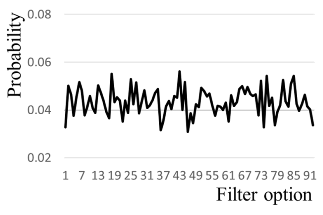


Figure 3.12: Controller prediction in iteration 49.

Similarly, the controller of experiment 2 was not able to output a tailored distributions given a MI on a data set either. Instead it converged to a single distribution that corresponds to outputting the architecture 115-90-70-10. Figure 3.2.2 shows 3 points in training of the controller:

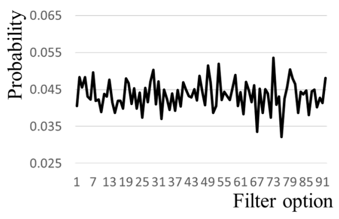


Figure 3.13: Controller prediction in iteration 1.

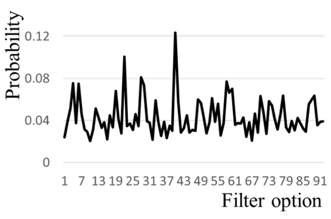


Figure 3.14: Controller prediction in iteration 22.

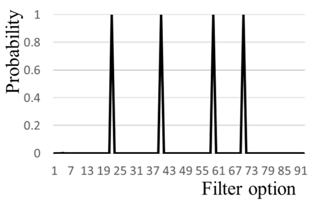


Figure 3.15: Controller prediction in iteration 31.

In both experiments the distributions (uniform or 115-90-70-10) were output independent of the input. They did not correspond to good prior distributions as can be seen in table 3.2: The controller of experiment 1 performs similar to a uniform prior distribution whereas the controller of experiment 2 performed even worse.

Prior distribution	Avg. score of Hyperband output
Uniform distribution	0.504 ±0.014
Predictor Exp. 1	0.506 ±0.015
Predictor Exp. 2	0.463 ±0.000

Table 3.2: The average score of a Hyperband output. Averaged over all 20 data sets such that each data set is represented equally much. Error bound is average standard deviation on a data set.

That the controller from experiment 2 performs worse than uniform distributions, comes from that the fact that once, a distribution has led to positive rewards enough often and becomes dominant in the controller output, the controller will converge towards that distribution in a vicious cycle: The Hyperband output relies fully on the controller output, thus gradients on the loss will be computed towards samples on the dominant function. If the function has previously been dominant on the data set as well, the base line function will lead to the fact that there will be no or only little negative rewards. This effect becomes stronger as more iterations are completed and the distributions converges more and more towards the dominant distribution.

The only way that this effect does not take place is when the controller finds a way to maximize the reward before a random walk will lead to a distribution being sufficiently dominant in the output. This however requires that the reward signal can hypothetically be maximized given the mutual information as input.

Chapter 4

Discussion

In this section, we discuss the limitations of our experimental set ups as well as the insights obtained by our results. In section 4.1 limitations of the set ups are listed. In section 4.2 we discuss what our results tell us about using mutual information in any neural architecture search and how likely it is, that mutual information will significantly improve the efficiency of that search.

4.1 Limiting factors

4.1.1 Sample size in posterior prediction

Although the training data for predicting posterior distributions contained 101640 samples, these samples only consisted of 220 independent measurements. In an attempt to improve this, we also added samples from the training process of the prior distribution predictors to the train samples. We hereby only took samples where the prior distribution fed to Hyperband closely resembled a uniform distribution. Like this, we were able to increase our training sample size to 520. However, the predictor delivered the same results and we thus included the results of the prediction with 220 samples for reasons of simplicity.

To drastically upscale the sample size further, a lot of computational capacity is required since each sample requires running an Hyperband instance (96 epochs of training neural nets), training a neural net to convergence and estimating the MI of the latter neural net.

4.1.2 Inaccuracy of MI estimation

For our experiments, we didn't use ground truth mutual information but rather an estimation using MINEs. Ground truth mutual information is infeasible to compute and thus wasn't an option. The estimation of the mutual information introduced noise to the experiments.

It is unclear how big that noise is. However, according to the universal approximation theorem, neural networks can approximate any function. In an attempt to evaluate how well our MINEs perform, we also found that choosing bigger sizes didn't provide a tighter lower bound. We trained all MINEs in the experiments until convergence. Furthermore, running a MINE on the same data several times (with different initializations) gave us very similar results.

4.2 Generalizing the experiment results

Many NAS algorithms come down to one of the following two tasks: Either, they sample architectures and evaluate how good they are. The goal here is to sample as few architectures as possible while not excluding good architectures from being sampled. (Examples: Reinforcement learning, Evolutionary algorithms). Or, the NAS tries to obtain information on a function that describes good architectures. Once it is enough confident on how the function looks like, it simply returns the function's argmax. (Example: Bayesian Optimization)

In our experiments, we trained two controllers that should predict good sampling distributions, as described in the first task. We also trained a predictor to predict a function describing good architectures as described in the second task.

Our versions of these tasks were simpler than the versions some NAS algorithms usually have to perform: We only worked with probability distributions over good architectures. In contrast many NAS algorithms not only have to determine what good hyperparameter choices are, but also how to combine them. Still, our predictors failed at leveraging MI to perform well at the tasks. We thus suggest, that mutual information is not suited for the above mentioned tasks and therefore not suited for most NAS approaches in general. Let's look at the current state-of-the-art search methods as examples, of why it is unlikely that their performance can be boosted using MI.

Bayesian Optimization

In Bayesian Optimization (BO) we optimize a black box function by taking samples from it. The efficiency of the method depends on how many samples we need to evaluate until finding a maximum of the function.

In the context of neural architecture search, that black box function is the function mapping architectures to accuracy (or some other score). The black box function is in that sense more complex than a probability distribution over good hyperparameter options: The black box function uniquely specifies an architecture and specific combinations of hyperparameters where as the probability distribution just specifies a distribution over those hyperparamters, not taking specific combinations into account.

Still, both functions have in common that they give information on what good architectures are. In the first experimental set up, we approximated the probability distribution function using Hyperbands. Our results show, that no strong statements about the approximated probability distribution could be made based on mutual information: The MI predictor performed worse on a held-out test set than Gaussians fitted on the average distribution of the train set. Our predictor was not able to find a connection between MI and the approximated probability function. To find a connection between MI and the more complex BO black box function therefore seems unlikely.

Reinforcement learning

Searching neural architectures using reinforcement learning has so for only included conducting a search on one data set. A neural network is simply fitted to convergence on that data set. For a new data set, a new search with a completely new training is started. It has been suggested to perform transfer learning to find an architecture on the new data set.

In the search procedure, a neural net predicts a probability distribution over a set of hyperparameter options. This is more closely described in section 1.3. An architecture is then sampled from that distribution and is trained. The reward signal is formed from the performance of the sampled architecture. Gradients are taken on the cross entropy of the predicted distribution and the distribution, that corresponds to the sampled architecture.

The reinforcement learning search is efficient, when only few distributions need to be predicted before a very good architecture is sampled from them. Usually the controller is trained until it converges to a distribution that corresponds to a single architecture. However, better architectures than the final one are often sampled along the way [32].

In our second experimental set up, we had a similar set up: We predicted prior distributions. We then sampled from the prior distribution and built our reward based on the performance of the samples. We defined a prior distribution to be good if its 4 best samples were good, ignoring some bad guesses.

In our experiments, our controllers couldn't optimize the reward signal in a way, that they converge to a local maximum that leads them to predict good prior distributions based on MI. Adding samples from a uniform distribution to the Hyperband input, translates to using an exploration rate in the reinforcement learning search. Both options, using an exploration rate or not, couldn't deliver convincing results. It seems therefore unlikely that MI as used in this context can increase efficiency for the very similar task of reinforcement learning NAS.

Evolutionary algorithms

When searching architectures using an evolutionary algorithm (EA), we start with a set of architectures called our first generation. We then evaluate the architectures and assign scores to them. Finally, we cross over architectures to form new ones for our second generation. Architectures with higher scores are more likely to be included in a cross over than architectures with low scores. We iterate through this process several times.

Generally, one iteration of an evolutionary algorithm comes down to sampling (crossing over) architectures and then evaluating them. An efficient EA NAS is one, where only few architectures need to be evaluated until finding a good one. This is achieved by only including good architectures in the evaluation set, or in other words, only sampling good architectures in the beginning of a round.

Our second experimental set up followed a similar procedure as one EA iteration: We first sampled architectures from a prior distribution. The goodness of the architectures of this round heavily depend on the prior distribution. After sampling, we evaluated the architectures, letting only the best 4 out of 12 survive.

As mentioned in the reinforcement learning subsection, the controllers of our experiment couldn't find a way to predict good prior distributions based on MI. The controller wasn't able to extract information on what hyperparamters he should try out. It seems therefore unlikely, that MI can increase efficiency for evolutionary algorithms in NAS.

4.3 Conclusion

Given the results in section 3 and the argument in section 4.2, we suggest that mutual information is not a suitable metric for evaluating neural networks in the context of NAS. Although mutual information provides a promising attempt at evaluating networks at a fine granularity, there remain open problems which our predictors have not been able to deal with: Firstly, it is not clear how the mutual information of individual layers should be interpreted to understand which parts of the network lead to good performance and which don't. Secondly, accurate estimation of MI remains difficult, even with methods like MINE. Since NAS is concerned with very subtle changes in hyperparameters, these inaccuracies can make mutual information unusable for tuning architectures at such a fine scale.

Apart from that, estimating mutual information with MINEs comes with big computational effort. For every neural network with n layers to be evaluated, n further neural networks have to be trained to convergence. It is not clear whether there is a fast way to deduce the mutual information of a layer from the mutual information of another layer. We found the following methods helpful in speeding up the training of the MINEs:

- The MINEs can be parallelized, since estimations do not depend on each other.
- We found, that MINEs between different layers of the same network can be transfer learned. That can drastically reduce the amount of iterations the MINE needs to converge. However, for one MINE to be initialized with the parameters of another MINE, the two MINEs cannot run in parallel.

Appendix A

Appendix

A.1 Hyperparameters for predictors

The architecture of the predictor neural networks was the same in all experiments:

Input	96
1. layer	128
2. layer	128
3. layer	64
output	92

All hidden layer units use ReLU activation.

A.2 Hyperparameters for MI neural estimation

The CNN MINEs for estimating the MI of layer i of a network with L layers, contained $L - i$ convolutional layers followed by average pooling, a fully connected layer, and finally the output layer. A convolutional layer contained 32 filters with kernel size 3x3.

The MINEs were trained for 18 epochs. They then were trained for additional epochs if the increase of the last epoch or the average increase of the last two epochs was above a threshold.

A.3 Smoothing Hyperband posterior distributions

We smoothed the Hyperband posterior distributions by adding 0.25 of the neighbouring scores to each field.

A.4 Standardizing the MI

For all train and test data sets, we first sampled 7 architectures at random, trained them on the data sets and derived their MI. That set of MIs was used as initial data to perform standardization. As we calculated more MIs for new architectures on a data set, those MIs were added to the data set's set of standardization MIs.

A.5 Caching of results

Since both training a neural network and estimating the mutual information of it are computationally expensive tasks, we cached all results for future use: Each neural network architecture is assigned an ID. When training that architecture and/or obtaining its mutual information on a data set, the results are written to a file and saved. Every time a new architecture is trained or that architecture's MI is required, we check if the information is already available for that data set and only calculate it if it isn't.

Bibliography

[1] Barret Zoph, Quoc V. Le. Neural Architecture Search with Reinforcement Learning. *arXiv preprint arXiv:1611.01578*, 2017.

[2] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, Quoc V. Le. Learning Transferable Architectures for Scalable Image Recognition. *arXiv preprint arXiv:1707.07012*, 2017.

[3] Angel Fernando Kuri-Morales. *The Best Neural Network Architecture*. Nature-Inspired Computation and Machine Learning. Lecture Notes in Computer Science, vol 8857. Springer, Cham, 2014.

[4] Roberto Battiti. Using Mutual Information for Selecting Features in Supervised Neural Net Learning. *IEEE Transactions on Neural Networks*, pages 537–547, 1994.

[5] Jasper Snoek, Hugo Larochelle, Ryan P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. *In NIPS*, 2012.

[6] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabás Póczos, Eric P. Xing. Neural Architecture Search with Bayesian Optimisation and Optimal Transport. *In NIPS*, 2018.

[7] Lizheng Ma, Jiaxu Cui, Bo Yang. Deep Neural Architecture Search with Deep Graph Bayesian Optimization. *arXiv preprint arXiv:1905.06159*, 2019.

[8] Hyunghun Cho, Yongjin Kim, Eunjung Lee, Daeyoung Choi, Yongjae Lee, Wonjong Rhee. DEEP-BO for Hyperparameter Optimization of Deep Networks. *arXiv preprint arXiv:1905.09680*, 2019.

[9] Quoc Le, Barret Zoph. Neural Architecture Search with Reinforcement Learning, Talk at Berkeley. 2017.

[10] Kenneth O. Stanley, Risto Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[11] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc Le, Alex Kurakin. Large-Scale Evolution of Image Classifiers. *In ICML*, 2017.

[12] Yukang Chen, Gaofeng Meng, Qian Zhang, Shiming Xiang, Chang Huang, Lisen Mu, Xinggang Wang. Reinforced Evolutionary Neural Architecture Search. *arXiv preprint arXiv:1808.00193*, 2018.

[13] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. Going Deeper with Convolutions. *In CVPR*, 2014.

[14] Shreyas Saxena, Jakob Verbeek. Convolutional Neural Fabrics. *In NIPS*, 2016.

[15] Lisa Torrey, Jude Shavlik. Transfer learning. *Handbook of Research on Machine Learning Applications*, 2009.

[16] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, Ameet Talwalkar. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *arXiv preprint arXiv:1603.06560*, 2016.

[17] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, Pieter Abbeel. InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets. *In NIPS*, 2016.

[18] Ravid Shwartz-Ziv, Naftali Tishby. Opening the Black Box of Deep Neural Networks via Information. *arXiv preprint arXiv:1703.00810*, 2017.

[19] Ziv Goldfeld, Ewout van den Berg, Kristjan Greenewald, Igor Melnyk, Nam Nguyen, Brian Kingsbury, Yury Polyanskiy. Estimating Information Flow in Deep Neural Networks. *In ICML*, 2019.

[20] Mohamed Ishmael Belghazi, Aristide Baratin, Sai Rajeswar, Sherjil Ozair, Yoshua Bengio, Aaron Courville, R Devon Hjelm. MINE: Mutual Information Neural Estimation. *In ICML*, 2018.

[21] Monroe D. Donsker, Srinivasa Varadhan. Asymptotic evaluation of certain markov process expectations for large time IV. *Communications on Pure and Applied Mathematics*, 36(2):183–212, 1983.

[22] MNIST. http://yann.lecun.com/exdb/mnist/.

[23] Han Xiao, Kashif Rasul, Roland Vollgraf. Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms. *arXiv preprint arXiv:1708.07747*, 2017.

[24] Gregory Cohen, Saeed Afshar, Jonathan Tapson, André van Schaik. EMNIST: an extension of MNIST to handwritten letters. *In 2017 International Joint Conference on Neural Networks, IJCNN 2017*, page 2921–2926, 2017.

[25] Tarin Clanuwat, Mikel Bober-Irizar, Asanobu Kitamoto, Alex Lamb, Kazuaki Yamamoto, David Ha. Deep Learning for Classical Japanese Literature. *In NIPS*, 2018.

[26] Alex Krizhevsky. Learning Multiple Layers of Features from Tiny Images. *University of Toronto*, 2009.

[27] Tiny ImageNet. https://tiny-imagenet.herokuapp.com/.

[28] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, Rob Fergus. Regularization of Neural Networks using DropConnect. *Proceedings of the 30th International Conference on Machine Learning, PMLR*, 28(3):1058–1066, 2013.

[29] Djork-Arné Clevert, Thomas Unterthiner, Sepp Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *arXiv preprint arXiv:1511.07289*, 2016.

[30] Jiayu Wu, Qixiang Zhang, Guoxi Xu. Tiny ImageNet Challenge. *In Stanford Convolutional Neural Networks for Visual Recognition Course*, 2017.

[31] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[32] Somshubra Majumdar. Implementation of Neural Architecture Search with Reinforcement Learning. https://github.com/titu1994/neural-architecture-search.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Mutual information for fine-grained network analysis in neural architecture search

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Kossmann	Ferdinand

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zurich, 30 August 2019	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.