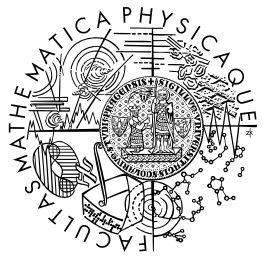

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Gabriela Suchopárová

Graph neural networks for NAS performance prediction

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis: Mgr. Roman Neruda, CSc.
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2021

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor Mgr. Roman Neruda, CSc. for his amazing support and guidance, and for all our inspiring discussions.
I am deeply grateful to my partner and family for their love and support throughout my studies.

Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures.

Title: Graph neural networks for NAS performance prediction

Author: Bc. Gabriela Suchopárová

department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Roman Neruda, CSc., Institute of Computer Science, The Czech Academy of Sciences

Abstract: In this work we present a novel approach to network embedding for neural architecture search — info-NAS. The model learns to predict the output features of a trained convolutional neural network on a set of input images. We use the NAS-Bench-101 search space as the neural architecture dataset, and the CIFAR-10 as the image dataset. For the purpose of this task, we extend an existing unsupervised graph variational autoencoder, arch2vec, by jointly training on unlabeled and labeled neural architectures in a semi-supervised manner. To evaluate our approach, we analyze how our model learns on the data, compare it to the original arch2vec, and finally, we evaluate both models on the NAS-Bench-101 search task and on the performance prediction task.

Keywords: neural architecture search, graph neural networks, AutoML, performance prediction

Contents

Introduction

In recent years, machine learning has entered almost every part of our everyday lives. Thanks to the success of deep learning, we can use a wide range of tools — home assistants based on advanced voice recognition systems aid us with tasks like shopping, our mobile phones produce beautiful artificially enhanced photography without any settings, and websites use recommendation systems to predict our taste in movies or music. Machine learning has also aided other fields like neuroimaging or chemical structure prediction, where the results surpass the previous state of the art methods.

However, behind every machine learning system is a large team of machine learning experts, who spent days choosing a good model and tuning its performance, and large computational resources were needed in the process. Machine learning is also inaccessible to domain experts with little knowledge of the field, as they often use only models they know or models with default hyperparameters, achieving suboptimal result. The automatic machine learning (AutoML) is a field that aims to alleviate these issues, aiding the experts in choosing the best-performing model, and making machine learning more accessible through automated tools. Since many successful applications rely on deep learning model, the neural architecture search (NAS), a the subfield of AutoML, attracted a lot of attention in the research community, with new breakthroughs appearing many times throughout the year.

Although the results of NAS are impressing, it still consumes a lot of resources — during the search for the best performing architecture, neural networks need to be trained and evaluated, which is a very long process. As so, the performance estimation methods for neural architecture performance are highly sought-after. A specific class of neural networks, the graph neural networks, has been often used for this task.

In this work, we use a graph variational autoencoder that embeds neural networks into a vector representation, the arch2vec. This model has achieved very good results in the NAS search process that used the latent features, and relatively interesting results in performance prediction. The goal of our work is to extend the model with additional information that could improve the perfor-

mance prediction results. To do so, we train the variational autoencoder along with a regressor that predicts the *output features* of a neural architecture given input images. We create an input-output dataset of images and outputs for a small number of trained networks for this task. The model is trained in a semi-supervised manner on both labeled (with input-output information) and unlabeled data.

The thesis has the following structure: in Chapter 1, we present the theoretical background for our work, describing basic concepts of machine learning, deep learning and AutoML. In Chapter 2, we mention some NAS performance estimators, with particular focus on graph neural network-based predictors and arch2vec. Chapter 3 introduces our model — info-NAS — in means of how we create the input-output dataset, what extensions were made to the original arch2vec model, and the training process. Finally, in Chapter 4, we run some experiments to assess how info-NAS learns from the data, and we compare it with arch2vec in terms of training metrics, NAS search performance and NAS performance prediction.

Chapter 1

Preliminaries

In this chapter we will introduce the theoretical background of machine learning, deep learning and AutoML. We describe the main machine learning tasks, the workflow of developing a model, a machine learning pipeline, and the evaluation of its performance. Then, we describe the basics of deep neural networks and their training, as well as some more advanced models — convolutional network, autoencoders and graph neural networks. Finally, we define the AutoML and its subfield, the neural architecture search.

1.1 Machine Learning

Machine learning is a broad field that encompasses a wide range of algorithms and different approaches. Mitchell defines machine learning as follows:

> A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E [1].

Another definition may be that machine learning provides a set of methods that can detect *patterns* in data, and use them to *predict* future data or other outcomes [2] [3].

Machine learning tasks are usually divided into three main types — supervised learning, unsupervised learning and reinforcement learning.

In the supervised case, the data has the form of input-output pairs $D = \{(x_i, y_i)\}_{i=1}^{N}$, where D is the *training dataset* and N is the number of *examples* (size of the dataset). The task is then to predict labels for the *test dataset*, which is a dataset with only unlabeled examples x_i available. There are two main types of supervised learning — classification, where the labels y_i are categorical (that is, $y_i \in \{1, \ldots, C\}$ — a finite set of classes), and regression, where $y_i \in \mathbb{R}$.

Contrary to the supervised task, the data in unsupervised learning is not labeled ($D = \{(x_i)\}$). The task definition is more ambiguous, generally the goal is to extract potentially interesting knowledge from the data. For example, *clustering* aims to group similar examples x_i to create a finite number of clusters [2]. Another example is *representation learning*, a method that has gained importance in recent years. It uses unsupervised methods to extract general features that would make subsequent training easier [4]. One popular application of this paradigm is *word2vec*, which computes vector representations of words. These extracted features improve the results on tasks such as word similarity [5]. Section 1.2.3 describes autoencoders, deep neural networks for unsupervised representation learning.

The reinforcement learning is fundamentally different from the supervised and unsupervised approach. Instead of a fixed dataset, the subject of reinforcement learning is an *agent* that learns to interact with its *environment* to fulfill a certain *goal*. It is often modeled as a Markov decision process defined as (S, A, p, γ), where S is the set of states (of the agent and the environment), A is the set of actions, $p(s', r|s, a)$ is the probability that action a in state s produces reward r and leads to state s', and γ is the discount factor. Initially, the agent is in the start state S_0, and moves to another state S_1 by means of action A_0, obtaining reward R_0. The same applies for any state-action pair (S_i, A_i) and the resulting (S_{i+1}, R_i). The goal is then to optimize the average *return* $\mathbb{E}[G_0]$, with the return defined as $G_t = \sum_{k=0}^{\infty} R_{t+1+k}$ [6].

Along with these main learning tasks, there are other task types that combine both approaches or do not fall strictly into one of the main areas. An example of such tasks is the *semi-supervised learning*. The input data for this task is a large number of unlabeled examples D_u, and a small number of labeled examples D_l. The main objective is similar to supervised learning — predict labels y for data x. However, since the number of labeled examples is small, it is necessary to learn from unlabeled data as well, and entirely different algorithms have to be used. There are two main paradigms of semi-supervised learning — the *generative* paradigm, where we estimate $P(x|y; \theta)$ — the conditional probability of data x given model parameters θ and labels y — and the final prediction is computed by means of the Bayes formula, and the *diagnostic* paradigm, where we estimate $P(y|x; \theta)$ directly (θ are the parameters of the used model) [7].

The next section will describe mainly the process of solving a supervised task, although most of the steps are used in the other types of learning as well.

1.1.1 Machine Learning Pipeline

Machine learning has been successfully applied to many practical problems. When a ML task is defined, there are several steps that must be done before a

working model is found and applied. This process is called the machine learning pipeline.

The steps of the ML pipeline are as follows: for a given task, sufficient amounts of data have to be collected. Then, the resulting data has to be cleaned, handling noise or missing data.

The next step is the feature engineering, where the goal is to create meaningful features using the raw data — the three main approaches are feature selection, feature extraction, and feature construction. During feature selection, redundant features are omitted from the raw data. Feature extraction reduces the dimensionality of the feature space as well, but unlike feature selection, existing features are altered. Finally, feature construction combines existing features into new ones to enhance the robustness and generalizability of the model [8]. Nowadays, deep neural networks combine the steps of feature extraction and prediction by creating latent features [4].

As the last step, a model is trained using the training data or features, and evalutated on a *validation set* — a held-out dataset that is used for test error estimation (details are explained in Section 1.1.2. This process is iteratively repeated until a satisfactory model is found, which is then used for final prediction on the test set [2]. The training procedure differs for every kind of machine learning algorithm. The training of neural network models is described in more detail in Section 1.2.

1.1.2 Performance of Models

The training procedure introduced in the previous section follows the definition of machine learning in Section 1.1 — a machine learning algorithm should improve its performance through learning from some experience. In this section, we describe the basic concepts of evaluation of model performance.

The performance measure or *metric* is usually specific to the defined task. For classification, we frequently use the *accuracy* metric, which is the proportion of correctly classified examples (Equation 1.1, $\hat{y}_i$ is the predicted label for x_i and y_i is the correct label) [4]. For regression, we often use the *mean squared error* (MSE, Equation 1.3) or the mean absolute error (MAE or also called the L1 error, Equation 1.4) [2].

Often, instead of a general metric we talk about the *loss*, which is a measure of how compatible our prediction and the ground truth is. Then, during the training, the goal is to *minimize* the expected loss. The MSE is a frequently used loss in regression problems. In case of classification, we could define the *0-1 loss* as the negative accuracy. However, the loss is not usually optimized directly, as it is a discrete measure that is hard to minimize. Instead, we usually minimize the *negative log likelihood* or NLL (Equation 1.2), where $\hat{p}(y_i|x_i, \theta)$ is the predicted

probability of the correct label y_i given x_i and model parameters θ [2] [4].

$$\frac{1}{n}\sum_{i=0}^{n}(\hat{y}_i = y_i) \quad (1.1) \qquad -\sum_{i=1}^{n}\log \hat{p}(y_i|x_i,\theta) \quad (1.2)$$

$$\frac{1}{n}\sum_{i=0}^{n}(\hat{y}_i - y_i)^2 \quad (1.3) \qquad \frac{1}{n}\sum_{i=0}^{n}|\hat{y}_i - y_i| \quad (1.4)$$

Although the training objective is to minimize the training loss, a good model has a small *generalization error* — the expected value of the loss on future data. When the model is too complex, it *overfits* on training data and learns every minor variation in the input, and is sensitive to noise — it has a large variance. Similarly, a model not complex enough does not have the capacity to learn enough — it is *biased* [2]. This is sometimes called the *bias-variance dilemma*; the 'ideal' model will have a complexity somewhere in between. In practice, the validation set from Section 1.1.1 is used to tune the model complexity, and the test set is used for the final estimation of the generalization error.

1.2 Deep Learning

Conventional machine learning algorithms relied heavily on feature engineering. Often, features were selected by domain experts who estimated which of the available information is relevant to the problem at hand. Data types like images or texts in the natural language were difficult to use, as the traditional models were prone to small variations in inputs, and it was impossible to process raw features like pixels. In other word, the models needed a suitable *representation* of the data [9].

Deep learning solves these problems by combining the step of learning the representation of the data and prediction. The prototypical deep learning model is the *multilayer perceptron* (MLP), also called the *(deep) feed-forward network*. It is composed of a chain of *layers* comprising multiple *neurons*, inspired by the structure of the biological brain. This network type has the form of a directed acyclic graph (DAG), that is, there are no connections from layers further in the chain to earlier layers. Additionaly, neurons in the same layer have no connections to each other [10]. Layers with this specific structure are called *dense* layers, or also *fully-connected* layers [11]. The first layer of a feed-forward network is called the *input*, the last one is the *output* layer. Finally, all layers in-between are called the *hidden* layers. A feed-forward network is deep if it has several hidden layers [4].

Mathematically, the goal of a feed-forward network is to approximate a function $\boldsymbol{f}$ using a sequence of function compositions $\boldsymbol{f} = f^{(n)}(\ldots(f^{(2)}(f^{(1)}(x)))$, where $f^{(i)}$ is the function computed at the i-th layer. Usually, the function has

the form $f^{(i)}(x_{i-1}) = a(x_{i-1}^T \cdot \theta_i + b_i)$, where a is the *activation* function, θ_i is the *weight matrix*, b_i is the *bias* and x_{i-1} is the output of the previous layer. The matrix θ_i has the shape (s_{i-1}, s_i), with s_i being the number of neurons at layer i. Similarly, x_{i-1} has shape s_{i-1} and the bias has shape s_i [10].

The simplest activation function is *linear* (Equation 1.5), which is the identity. It is rarely used in the hidden layers, but it can be used in the output layer for regression problems. Some common activation functions for the output layer are the *sigmoid* function (Equation 1.7), used for binary classification, and the *softmax* function (Equation 1.8), used for multi-class classification. Nowadays the most used activation function in the hidden layers is *ReLU* (rectified linear unit, Equation 1.6) [4].

$$linear(z) = z \qquad (1.5) \qquad ReLU(z) = \max(0, z) \qquad (1.6)$$

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \qquad (1.7) \qquad softmax(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \qquad (1.8)$$

There are many other types of neural networks for different data types. Feed-forward networks process data that can be converted to vector examples, but there exists specialized architectures that solve the problem more accurately and efficiently. The convolutional networks, that are used for image data processing (or any grid-like data), are described in more detail in Section 1.2.2. For sequence data, we use the *recurrent neural networks* — networks with a hidden state and recurrent connections. Some common applications of this architecture are the natural language processing, as the language can be seen as a sequence of characters and words, and reinforcement learning, where we process the sequence of state-action pair along with the rewards.

1.2.1 Training of Deep Neural Networks

In the previous section, we described the neural networks as a function of the input, activation function, and weights and bias. In practice, the bias is included as the last row in the weight matrix, and the inputs are extended to $(x_i, 1)^T$.

Suppose we want to approximate a certain value, the *target* function $y = f^*(x)$, using a neural network. In other words, we want to find such a set of weights θ, so that the neural network approximates the target function by $y = f(x; \theta)$. Moreover, the architecture of the network and the activation functions are fixed [4].

To evaluate the approximation, we use a suitable loss function, as defined in Section 1.1.2. Since the other inputs are fixed, the loss depends only on the weights, denoted as $J(\theta)$. That is, to obtain a good approximation, we need to

find suitable weights θ. The formal definition of *training* a neural network is provided in Equation 1.9 [10].

$$\arg\min_{\theta} J(\theta) \tag{1.9}$$

Optimizers Most frequently, neural networks are optimized using the (batch) *gradient descent* algorithm. During the optimization, the weights are iteratively changed as described in Equation 1.10. In each update, the gradient of the loss function (evaluated on all of the input data) is computed with respect to the weights. The hyperparameter η is the *learning rate* or step size. The weight update is repeated for a certain number of *epochs* or until a convergence criterion is reached [12] Batch gradient descent converges to the global optimum of convex loss surfaces, and to local minimam of non-convex [10].

The process of computing the loss function $J(\theta; x; y)$ is called the *forward* propagation, as the data is passed forward through the layers. The process of computing the gradient is called the *back-propagation*, as the gradients are propagated from the output layer backwards [4].

In practice, the vanilla gradient descent can be intractable for large datasets, as all data needs to be loaded into memory for computing the gradient. We can instead use the *stochastic gradient descent* (SGD), where the update is performed for every example separately (Equation 1.11 for $n = 0$) [12]. It has been shown that SGD converges as well if the following conditions for the learning rates are satisfied:

$$\sum_{k=1}^{\infty} \eta_k = \infty, \sum_{k=1}^{\infty} \eta_k^2 < \infty,$$

where k is the number of the epoch and η_k is the learning rate chosen for the epoch. This sequence of learning rates is called the learning rate *schedule* [2].

The updates in SGD have a great variance, as the gradient is estimated using only one training example. A compromise between batch gradient descent and SGD is the *mini-batch gradient descent* (Equation 1.11 for $n > 0$). Its updates have a lower variants and the computation can be accelerated using the GPUs. The parameter n is also called the *batch size* [12]. Sometimes SGD is used as a synonym for mini-batch gradient descent [11].

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta) \tag{1.10}$$

$$\theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{i:i+n}; y^{i:i+n}) \tag{1.11}$$

The previously defined algorithms still have caveats — for instance, the learning rate or its schedule has to be set beforehand, and it does not adapt to the input

data. Also, although the stochastic nature of the SGD helps to avoid local minima, it may still be difficult for some functions. These problems can be avoided using some extensions.

A robust variant of SGD is the SGD with *momentum* (Equation 1.12). The momentum term helps to accelerate SGD in a relevant direction with less oscillations, and possibly improve the convergence [13] [12].

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \cdot \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{1.12}$$

There is a wide number of optimization algorithms that *adapt* the learning rate to the training process, e.g. AdaGrad [14], RMSProp [15] and Adam [16].

Training improvements In practice, there are some commonly used approaches that aim to improve the training process. An important step is *input normalization*, where the input data is scaled to have zero mean and unit covariance [17]. The weight initialization is also essential — weights are usually uniformly selected from a given interval, commonly using the *normalized initialization* (for a layer with m inputs and n outputs [18]:

$$W \sim U\left[-\frac{6}{\sqrt{m+n}}, \frac{6}{\sqrt{m+n}}\right].$$

There are numerous *regularization* techniques used in order to avoid overfitting (Section 1.1.2). One of them is the L2 regularization (also known as *weight decay*), which adds the following *penalty* term to the loss function: $\alpha \cdot \frac{1}{2}||w||$. This drives the weights closer to the origin [4]. Another powerful regularization method is the *dropout* layer. It accepts outputs from the previous layer and for every output, it sets it value to 0 (with probability p). It can be seen as a form of ensembling, as it trains an exponential number of smaller subsampled networks at the same time. The final model is then agreggated from these networks, using smaller weights [19].

1.2.2 Convolutional Neural Networks

Convolutional networks are specialized neural networks for data that has a grid-like topology [4]. They were first applied to 2-dimensional image data in a classification problem [20], but many other applications emerged, for example in sequence classification on 1-dimensional text data [21]. In this section, we will describe the essentials of 2-dimensional convolutional networks.

The architecture of a convolutional neural network can be chain-structured, however more complex networks with *skip-connections* (sub-chains that connect

layers outside of the main chain) have been empirically shown to perform well (e.g. ResNet [22]). Just like the feed-forward networks with dense layers, they are trained using the back propagation algorithm described in Section 1.2.1 [23]. Most frequently, the networks are composed from the *convolutional* layers, *pooling* layers, and *batch normalization* layers.

Convolutional layer The layer relies on the *convolution operator*, denoted by the sign $*$, which has origins in the signal theory. In the continuous case, it is defined as follows [24]:

$$y(t) = (x * h)(t) = \int_{-\infty}^{\infty} x(t) \cdot h(t - \tau)\, d\tau. \tag{1.13}$$

The function x in Equation 1.13 is called the *input* and function h is called the *kernel*. The operator can be defined for the discrete case as well, but usually the *cross-correlation* operator is used instead, and it is implemented as

$$(x \star h)(i, j) = \sum_{m=0}^{M} \sum_{n=0}^{N} x(i + m, j + n) \cdot h(m, n), \tag{1.14}$$

where (M, N) is the shape of the discrete kernel. This operator is the same as a convolution but without a flipped kernel (for discrete convolution, Equation 1.14 would differ in the term $x - x(i - m, j - n)$), and it maintains the properties that are useful to the implementation of a convolutional network [4].

The process of applying the convolution operator on the input can be seen as moving the kernel over the grid, producing one output at each position. For inputs with multiple channels, there are kernel weights specific to the channel, and the result is the sum of all the output values [25]. This way, every output depends only on a small number of inputs, which makes the connections between two adjacent layers *sparse*. Also, *parameter sharing* is present through the kernel — per one output channel, we use the same kernel weights for every input position. As a consequence, the convolutional layer is *equivariant* to translation, in other words, if the input changes (e.g. an object is moved to another position in the image), the output changes in the same manner [4].

When moving the kernel over the input, the output feature maps do not have the same width and height as the input (unless the kernel size is 1). This may be inconvenient, as the size of the input shrinks each time the operator is applied, and the border features are used to form less outputs than the other features. As so, we sometimes choose to apply some *padding* to the input, e.g. by zeros, which means that the input is surrounded with zeros k times to preserve the dimensions — that is, we apply k times a 'frame' of zeros, so that the network can produce a valid output outside of the image [11].

Pooling layer The pooling layer also moves a rectangular window over the input, but it does not have any weights. Instead, it summarizes the data inside the window — common variants are max pooling and average pooling — and computes the values separately for each channel [25]. Pooling is used to make the representation invariant to small translations in the input — that is, when an object is moved in the input by a slight amount, the representation does not change [4]. As so, the output is less sensitive to distortions and shifts [25].

Another variant of a pooling layer is the global pooling layer, which is the same as a regular pooling layer with the window overlapping the whole input — that is, per one channel, a single value is produced. The global average pooling is commonly used before the output layer [26].

Batch normalization As described in Section 1.2.1, normalizing the inputs speeds up the training process. When training a deep neural network, a small change in parameters of a layer affects all following layers. In other words, the *distribution* of inputs to the layer changes each time we update the network parameters. This phenomena is called the *internal covariate shift* and it is known to slow down the training process.

Batch normalization has been proposed to adress this problem. The layer first computes mean and variance of the batch $x^{(i)}$ (Equation 1.15), then it normalizes it (Equation 1.16). Finally, it also learns a suitable scale γ and shift β, so that the original representation ability of the network is preserved (Equation 1.17) [27].

$$\mu = \frac{1}{n}\sum_{j=1}^{m} x_j^{(i)},\ \sigma^2 = \frac{1}{n}\sum_{j=1}^{m}(x_j^{(i)} - \mu)^2 \tag{1.15}$$

$$\hat{x}^{(i)} = \frac{x^{(i)} - \mu}{\sqrt{\sigma^2}} \tag{1.16}$$

$$\hat{y}^{(i)} = \gamma\hat{x}^{(i)} + \beta \tag{1.17}$$

Network architecture There are several approaches to creating the architecture of a convolutional network. Generally, a convolutional layer is applied, followed by batch normalization and some nonlinearity (usually ReLU) [22]. This motif can be repeated multiple times, afterwards, the max pooling is applied to decrease the spatial dimensions [28]. Commonly, a strided convolution is used instead of max pooling [29]. Usually, when halving the spatial dimensions, the number of channels is doubled [22].

This process is repeated multiple times. Then, the output can be flattened (reshaped into a vector with the same number of features) to be processed through

multiple dense layers to produce the final prediction [25], or, more often, the global average pooling is applied before the output dense layer [26].

1.2.3 Unsupervised Representation Learning

When solving a machine learning problem, choosing a suitable representation can be essential. With the success of deep neural network, the step of feature extraction has been incorporated directly into the learning process. However, there are still problems that can benefit from a pretraining step. For example, during transfer learning, the model is built on top of a representation extracted when solving a different task. Other applications include dimensionality reduction for traditional ML algorithms, or denoising [4].

One of the models used for representation learning is the autoencoder. It is a neural network that aims to copy its input to the output. Internally, it consists of an *encoder* that encodes the input into a lower-dimensional representation, typically into a *latent* vector. Then, the *decoder* network reconstructs the vector into the output [4]. The loss function of such a model can be MSE for continuous data, or binary cross entropy for continuous data scaled to the interval $(0, 1)$.

Variational autoencoder An extension of the basic autoencoder is the *variational autoencoder* (VAE). It has a similar architecture, but the learning objective differs — it is a deep generative model that aims to learn a distribution $P(x)$ from which the input data x was sampled [4].

Suppose we have data $X = \{x^{(i)}\}_{i=1}^{n}$ that was generated through a random process using unobserved continuous random variables $\boldsymbol{z}^{(i)}$ — for a sample, $\boldsymbol{z}^{(i)}$ had been drawn from an unknown (parametric) distribution $p_\theta(\boldsymbol{z})$, then, $x^{(i)}$ was drawn from some conditional parametric distribution $p_\theta(x|\boldsymbol{z})$ (both probability density functions are assumed to be differentiable). Our objective is to estimate the parameters θ^* using the VAE [30].

When solving the problem, we would like to sample z using the posterior $p_\theta(z|x)$. However, it may be intractable to evaluate this expression; instead, we learn an approximation $q_\phi(z|x)$ (the encoder). Then, we will sample z from this distribution and use it to learn a generator model $p_\theta(x|z)$ (the decoder) [4].

With this setting, the VAE can be trained by maximizing the *variational lower bound*, which is defined in Equation 1.19 [30] [31]. In the final form, the first term is the *reconstruction error*, and the second term is the KL divergence (Equation 1.18) of the posterior estimate and the prior distribution of z. Since the KL divergence is non-negative, it holds that $L(\theta, \phi, x) \leq \log p_\theta(x)$ [4].

$$D_{KL}(P(x)||Q(x)) = \sum_x P(x) \log \left(\frac{P(x)}{Q(x)} \right) \tag{1.18}$$

$$
\begin{aligned}
L(\theta, \phi, x) &= \log p_\theta(x) - D_{KL}(q_\phi(z|x) \,||\, p_\theta(z|x)) \\
&= \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x, z) - \log q_\phi(z|x)] \\
&= \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{KL}(q_\phi(z|x) \,||\, p_\theta(z))
\end{aligned} \tag{1.19}
$$

In practice, we use the so-called *reparametrization trick* to sample from $q_\phi(z|x)$. We express the sampled z as $z = g\phi(\epsilon, x)$, where ϵ is an independent variable and g is a vector-valued function parametrized by ϕ. For example, for gaussian noise vector $\epsilon \sim \mathcal{N}(0, \mathbf{I})$, the value $z^{(i)}$ is expressed as $z^{(i)} = \mu^{(i)} + \sigma^{(i)} \odot \epsilon$ ($\odot$ is elementwise product). Using this representation, the KL divergence can be evaluated and differentiated without estimation, and it the following holds [30]:

$$
\log q_\phi(z|x^{(i)}) = \log \mathcal{N}(z^{(i)}, \mu^{(i)}; \sigma^{2(i)}\mathbf{I}).
$$

The variational autoencoder has been shown to perform well in a generative semi-supervised setting, where it is jointly trained with a classifier. The classifier takes advantage of the representation extracted through VAE on unlabeled data, while it learns on labeled data [32].

1.2.4 Graph Neural Networks

In this section we will describe a specific type of neural networks applied to structural data — *graph neural networks*. A graph is defined as $G = (V, E)$, where V is the set of *vertices* and $E \subset V \times V$ is the set of *edges* [33]. Many theoretical as well as practical concepts can be described by graph theory, for example in chemistry [34], satisfiability of logical formulae [35], and the neural networks themselves can be seen as directed graphs [4].

Graph neural networks (GNNs) are a broad class of deep networks processing graph data. There exist different architectures, for example recurrent graph neural networks, convolutional graph neural networks or graph autoencoders. The last two will be described in more detail [36].

Convolutional graph neural networks The convolutional GNNs stack multiple convolutional layers to extract node representations, aggregating the features of a node (x_v) with features of its neighbors ($\forall u, (u, v) \in E : x_u$). Compared to the convolution on grid-structured data introduced in Section 1.2.2, the graph convolution is much more difficult to define. For instance, in image data every pixel has a fixed number of neighbors, and the output is aggregated from the same number of pixels every time. On the other hand, a node in a graph may have an arbitrary number of neighbors, and there may not be an ordering

defined on them. There are two main approaches to graph convolutions; the first are *spectral* methods, which have origins in signal processing, and the second are *spatial* methods, that focus on information propagation [36].

The spectral-based methods work with the so-called normalized graph Laplacian matrix, defined as $L = I_n - D^{-1/2}AD^{-1/2}$, where A is the adjacency matrix and D is a diagonal matrix of node degrees. This matrix is further factorized to $L = U\Lambda U^T$, and the graph convolution on node features x is then defined [37]:

$$x *_G g = U(U^T x \odot U^T g).$$

The specific spectral convolutions then differ in how they choose the filter g [36]. The graph convolutional network (GCN) is a spectral method, but it can be also seen as a spatial method, as it also aggregates the nodes' neighborhood [38].

The spatial-based methods work in a message-passing manner. The general framework has been defined by the Message Passing Neural Network (MPNN) [39]. Equation 1.20 specifies the message passing function, where $h_v^{(0)} = x_v$ is the feature vector of x, x_{vu}^e is the edge feature vector, and U_k, M_k are functions with learnable parameters.

$$h_v^{(k)} = U_k\left(h_v^{(k-1)}, \sum_{u \in N(v)} M_k(h_v^{(k-1)}, h_u^{(k-1)}, x_{vu}^e)\right). \tag{1.20}$$

The message passing is executed in K steps using the aforementioned rule.

An important spatial convolutional network is the Graph Isomorfism Network (GIN), an architecture more powerful than previous message-passing networks, as it enables to distinguish different graph structures based on the output embedding. It has the following update rule:

$$h_v^{(k)} = MLP\left((1 + \epsilon^{(k)})h_v^{(k-1)} + \sum_{u \in N(v)} h_u^{(k-1)}\right),$$

MLP is the multi-layer perceptron and $\epsilon^{(k)}$ is a learnable parameter [40].

Graph autoencoders Graph autoencoders apply convolutional networks, such as GCN, to encode a graph into a latent *matrix* embedding Z [36]. Just like in standard deep learning, there exists a variational graph autoencoder (VGAE) that works in a similar way and optimizes a modification of the variational lower bound [41]:

$$L = \mathbb{E}_{q_\phi(Z|X,A)}[\log p_\theta(A|Z)] - D_{KL}(q_\phi(Z|X,A) \,||\, p_\theta(Z)) \tag{1.21}$$

The generative model is described in Equation 1.23. It handles only the adjacency matrix reconstruction, as the features are not reconstructed in this implementation, and it decodes the matrix from the latent matrix $Z = \{z_i\}_{i=1}^{N}$ using the inner product between matrix rows (Equation 1.22). The VGAE operates on networks with a fixed maximum number of nodes N [41].

$$p(A_{ij} = 1|z_i, z_j) = \sigma(z_i^T z_j) \tag{1.22}$$

$$p(A|Z) = \prod_{i=1}^{N} \prod_{j=1}^{N} p(A_{ij}|z_i, z_j) \tag{1.23}$$

There exist many different approaches to graph decoding or generation, some of them output the nodes and edges in a sequential manner, some output the whole graph at once as in VGAE [36].

1.3 AutoML

When solving a practical machine learning problem, several experts work together to develop a working solution. This process can be summarized by the ML pipeline, as is described in Section 1.1.1.

However, the steps of the machine learning pipeline are not without problems. Data collection may be very challenging, as it is not always possible to gather enough clean and labeled (in case of supervised learning) data. Also, both data collection and feature engineering sometimes need considerable domain knowledge to construct a dataset that it is possible to use for predictions [8]. A single ML algorithm that outperforms all other algorithms on every task cannot exist (as a consequence of the 'no free lunch' theorem [42]).

Thus, the process of finding a good model for a particular use-case is tedious and is usually done by means of trial-end-error. As a fall-back, users sometime use models with default hyperparameter settings, or models they know, especially when lacking experience in ML. To summarize, the whole machine learning workflow is very time-consuming and requires a large amount of available resources, and it may sometimes lead to suboptimal results [8].

1.3.1 Definiton of AutoML

The automated machine learning (AutoML) is a novel field that aims to alleviate these issues. It aims to automate a subset of the ML pipeline, or even the whole process. Formally, the process of finding a suitable pipeline (or a part of it) can be defined as an optimization problem.

First, let us define a ML pipeline as the triplet $(g, \vec{A}, \vec{\lambda})$, denoted $P_{g,\vec{A},\vec{\lambda}}$, where g is a *valid* structure of the pipeline, $\vec{A} \in A^{|g|}$ is a vector of algorithms chosen from the set of possible algorithms $\{A_0, A_1, \ldots, A_n\}$, and $\vec{\lambda}$ is a vector of hyperparameters chosen for each algorithm. This means that for the position number i in the architecture, we have chosen algorithm $\vec{A}_i = A_k$ with hyperparameters $\vec{\lambda}_i$, and $\vec{\lambda}_i$ is a list of hyperparameter values chosen from the hyperparameter domain corresponding to algorithm k: $\Lambda_k = \Lambda_1 \times \Lambda_2 \times \ldots \times \Lambda_m$.

Then, the *pipeline creation problem* is defined as:

$$P_{g^*,\vec{A}^*,\vec{\lambda}^*} = \arg\min_{g,\vec{A},\vec{\lambda}} = R(P_{g,\vec{A},\vec{\lambda}}, D), \tag{1.24}$$

where P is a pipeline, D is the dataset and function R evaluates the pipeline and returns the resulting loss [43].

The optimization problem specified in Equation 1.24 can be divided into three main parts — *architecture selection* (also called structure selection), *algorithm selection* and *hyperparametetr optimization* [43]. The last two are sometimes optimized together and form the so-called *CASH* problem — combined algorithm selection and hyperparameter optimization problem [44].

Some popular AutoML systems are e.g. Auto-WEKA [44], auto-sklearn [45] or TPOT [46]. These systems use mostly conventional machine learning algorithms, although TPOT supports some neural network models as well, and they try to optimize the whole ML pipeline.

The subset of AutoML that limits the model search space to neural networks is called the *neural architecture search* (NAS). Compared to the aforementioned AutoML frameworks, most NAS systems focus only on the architecture search part, although there have been some attempts at jointly solving HPO as well [47] [48] [8]. This rapidly evolving field will be described in more detail in the following subsection.

1.3.2 Neural Architecture Search

NAS is a rapidly evolving field, with research targeting a wide range of both practical and theoretical aspects of neural networks, their architecture and training [49]. The family of models called the EfficientNets has been designed using NAS [50], and extensions of these are among the current state-of-the-art models on the ImageNet dataset [51] [52] [53].

This section presents basic theoretical aspects that are common for most of the works in this field [54].

Search Space As has been already mentioned in previous sections, a neural architecture can be seen as a directed acyclic graph (DAG). To describe the graph in more detail, it is defined as (Z, E), where E are connections (edges) between nodes, and Z are nodes represented by a tensor. Furthermore, we define for each node $z^{(k)} \in Z$ an operation $o^{(k)} \in O$ on its set of parent nodes $I^{(k)}$ (specified by the edge set) [55]. The set of operations is specific to the search space, an example being the convolution or an activation function [8].

The search spaces may differ depending on what type of architectures is desired, or what task is to be solved. These include search spaces of graph convolutional networks [56], convolutional or recurrent neural networks [57], as well as many other [49].

We will now describe the two main search space groups of convolutional NAS, although the general ideas may be applied to an arbitrary search space. The first is the *global search space* (also called the entire-structured search space). It represents the whole architecture, with nodes corresponding to layers, and it may be also limited to a specific structure — e.g. a chain-structured search space [55]. This representation allows a large amount of freedom in designing the architectures, it may however be hard to optimize due to a possibly large size of the networks [8]. The second approach is the *cell-based* search space, a popular model widely used in benchmarks and applications. Instead of designing the whole architecture, we optimize a *cell* — a smaller motif of connected layers — and the resulting network is composed from the resulting cells (usually in a predefined manner) [55]. The first cell-based search space was the *NASNet* [58] that consisted of normal (spatial dimensions preserving) and reduction cells. Other search spaces followed, the most popular ones being NAS-Bench-101 [59] (described in detail in Chapter 3.1), DARTS [60] (where the search space is relaxed to a continuous version) or NAS-Bench-201 [61].

Search Strategy The search strategy specifies the algorithm of navigating through the search space to find the best performing architecture. Historically, neural architectures were optimized by means of evolutionary algorithms [54] [62] [63], and nowadays there are new applications being developed. The architectures in the search process are then represented as individuals and evolved to obtain better networks [8].

Another group of algorithms applied to NAS is reinforcement learning. The search is guided through a controller (for example a RNN) that samples architectures as an action and receives a reward and an observation of the state, updating the controller sampling strategy)[58] [8]. The controller can be trained e.g. by using the policy gradient algorithm (REINFORCE) [64].

Other optimization methods include bayesian optimization, differentiable ar-

chitecture search (an application of gradient descent on the search space [60]) and random search — the latter is commonly used as a baseline [8].

Performance Estimation The evaluation of all networks throughout the search process is usually infeasible due to the long training time. As so, performance predictors are highly sought-after. One possibility is to approximate the true performance through *low-fidelity* approximations. These include lower number of epochs, lower resolution of images or subsampling of the input dataset [8]. Another broad area is the one-shot NAS, where only one large network is trained, and the search space are the subsets of this large networks — these small networks are not evaluated again [54]. A different weight-sharing approach is the so-called *network morphism*, where new networks are created from trained parent networks using function-preserving mutations [65].

A promising approach to performance estimation is surrogate modelling, a method for estimating the outcome of a black-box model. Surrogate models are used in engineering to estimate the outcomes of a time-consuming or difficult simulation [66]. In machine learning, they are used to approximate the *predictions* of machine learning. The advantage is that the chosen model may be simpler to evaluate as well as more interpretable than the original one [67]. Specific examples of surrogate models in NAS will be introduced in the following chapter (Section 2.1).

Chapter 2

Related Work

In this section we describe some recent works relevant for our solution.

2.1 Surrogate Models for Neural Architecture Search

Surrogate-based modelling has been used in NAS to avoid long performance evaluation times. The main approach how to leverage performance predictors is the *surrogate model-based optimization* (SMBO), where the surrogate model is trained during the search process along with the NAS optimizer. Commonly used model classes include bayesian optimization methods (such as gaussian processes [68]), and neural networks [8]. Other works introduced an offline pretraining of performance predictors, such as a random forest based predictor [69] or graph neural network predictors [70].

When training a performance estimator on neural architectures, it relies on a suitable representation of the network graph. In PNAS, the network is represented as a variable-length string of node and operations one-hot encodings, and the performance predictor is a RNN (specifically, a LSTM model [71]) [72].

In neural architecture optimization (NAO), a LSTM is trained on a sequence of string tokens to extract continuous embeddings, then, a simple MLP predictor is trained on them [73]. An extension of NAO, SemiNAS, trains the performance predictor in a semi-supervised manner, where it first trains on few labeled architectures, and then it gradually adds unlabeled networks along with their predicted labels to the training dataset [74].

A different approach, BANANAS [75], combines bayesian optimization with a neural predictor. The architectures are represented as possible paths from input to the output (encodet as a binary string, where 1 indicates the path is present),

and they are passed to an ensemble of dense networks to produce the performance prediction.

In the following section, we will focus on graph neural networks applied to neural architectures, possibly combined with performance estimators. Compared to the above-described models that embed strings of information, the advantage of GNNs is that they embed the structural information as well.

2.2 Graph Neural Network Embeddings

The input neural architectures are usually embedded as a binary adjacency matrix along with a one-hot feature vector. As described in Section 1.2.4, there exist several different approaches to graph convolutions. Moreover, since the convolution operates on node-level, there is not one straightforward way how to combine node features to a global representation of the graph.

Graph neural networks have become popular in NAS performance prediction — in fact, there exists a meta-framework for evolving GCN architectures for performance prediction [76]. We will now describe some specific works along with their prediction procedure.

In BONAS [77], a GCN is trained to perform the embedding, then a regressor is trained on the embeddings to predict network accuracy. The aggregation of features is done through an additional *global node*, a node connected to all original nodes in the graph.

The model GATES [78] introduces an additional vector that represents the input nodes, and then simulates the flow of the information through the computational graph. The graph convolutional operations are specific for this work, and the model is trained jointly with the predictor.

In the following section, we will describe *arch2vec*, an unsupervised variational graph autoencoder that creates neural architecture embeddings. The authors also compared it with its 'supervised' variant, which is the encoder part trained jointly during the search process.

2.2.1 Arch2vec

The arch2vec [70] is the model we extend in out work. It is a model similar to the VGAE described in Section 1.2.4, but with key differences. First, it uses GIN layers instead of GCN (both described in Section 1.2.4), as they represent better the original architectures. Second, the network features X are also decoded, using one dense layer (parametrized by W, b) and one softmax layer (Equation 2.1, N is the number of nodes and $k_i \in K$ are operations from the set of possible operations). Since $\hat{X}$ and $\hat{A}$ are reconstructed independently, we can update the

reconstruction loss to $p(\hat{X}|Z) \cdot P(\hat{A}) = P(\hat{A}, \hat{X}|Z)$. Otherwise, the loss is the same as in Equation 1.21 in Section 1.2.4.

Table 2.1: Model and training hyperparameters of arch2vec.

model parameter	
latent dimension	16
adjacency activation	sigmoid
operations activations	softmax
reconstruction loss	binary crossentropy
dropout	0.3
GIN MLP layers	2
GIN MLP features	128
GIN iterations per layer	5
training parameter	
batch size	32
epochs	8
optimizer	Adam
learning rate	10^{-3}
betas	[0.9, 0.999]
eps	10^{-8}

To improve the training, the input matrix A is augmented to $\bar{A} = A + A^T$ before being passed to the model, as to allow information flow in both directions. Also, before reconstructing the A and X in the decoder, the latent matrix Z is passed through a dropout layer. Table 2.1 lists all additional model and training parameters.

$$p(\hat{X} = [k_1, \ldots, k_N]^T|Z) = \prod_{i=1}^{N} P(\hat{X}_i|z_i) = \prod_{i=1}^{N} \text{softmax}(W \cdot Z + b)_{i,k_i} \quad (2.1)$$

The search spaces chosen for evaluation were NAS-Bench-101, NAS-Bench-201 and DARTS. The model was trained for 7 epochs on 90% of the search space and evaluated on the remaining 10%. In terms of *reconstruction accuracy*, *validity* and *uniqueness*[1] (metric for graph reconstruction proposed by [79]), it outperformed the baseline GAE (VGAE, but regular autoencoder) and VGAE on all

[1]Validity is computed by sampling from the latent space a vector $z \sim N(\mu_z, \sigma_z)$ (μ_z and σ_z are mean and standard deviation of latent space encodings of the training data), and determining whether the result is a valid network. Uniqueness is the proportion of unique networks sampled this way.

of the search spaces. Arch2vec itself also performed better as a variational autoencoder rather than a regular one.

The arch2vec model has also been used during optimization, not as a performance predictor, but to provide the extracted features. Also, for comparison the encoder part was jointly trained in a supervised manner, to asses if the pretraining step is necessary for extraction of good features.

The search algorithms used along with arch2vec (both supervised and unsupervised) were REINFORCE and BOHB [80]. For comparison, the authors ran the optimization using random search [81], regularized evolution [82] and vanilla REINFORCE and BOHB (for the DARTS search space, different methods were used). In all search spaces, arch2vec outperformed the other algorithms as well as its supervised counterpart.

Lastly, both arch2vec and supervised arch2vec were evaluated as performance predictors on NAS-Bench-101 using RMSE and Pearson's r metrics. Specifically, a gaussian process model was trained on 250 sampled architectures (their extracted features) and evaluated on points with test accuracy > 0.8. The unsupervised arch2vec performed better than its supervised counterpart, and its RMSE/Pearson's r were 0.018 ± 0.001 / 0.67 ± 0.02 (RMSE closer to 0 is better, Pearson's r $\in [-1, 1]$ closer to 1 is better) [70].

Chapter 3

Our Solution

In our work, we extend an existing graph autoencoder by trying to learn the *outputs* of a network given input data. The motivation is that networks with different architectures may still approximate the same function — a property employed in network morphisms — and maybe even learn the same set of features. This additional information could improve the performance of graph embeddings for NAS tasks, since it takes into account not only the structure of the networks, but also how the networks operate on a specific task. Also, it is an opportunity to study what kind of features the different networks learn. We call our system *info-NAS*, since it learns the information processed by neural networks.

We have chosen the neural graph autoencoder *arch2vec* (Section 2.2.1), as it achieved good results on NAS benchmark tasks, and its architecture is relatively simple, compared to other works. In Section 3.3, we describe how we extended the model, and in Section 3.4 we provide details on how we train it.

In the first implementation of info-nas, we focus on the NAS-Bench-101 search space, described in detail in Section 3.1.2, evaluated on the CIFAR-10 dataset (Section 3.1.1). We have chosen these datasets due to their simplicity, as NAS-Bench-101 contains network with a fixed maximal node number, and the image size of CIFAR-10 is small.

Although the search space is provided along with performance metrics on the dataset, the checkpoints of trained networks are not available. As so, it is necessary to train them, which is however infeasible for the whole search space[1]. To alleviate the need of a fully-labelled dataset, we train the model by means of *semi-supervised* learning — we sample a small number of networks to train, create the labeled dataset (Section 3.2), and then jointly train the extended model on both labeled and unlabeled data.

Figure 3.1 summarizes all steps of the process, i.e. dataset creation and joint

[1]The authors of NAS-Bench-101 utilized over 100 TPU years for the training [59].

training. In the following sections, we describe the implementation in detail along with some issues we had to solve. Since the original arch2vec is written in PyTorch, we use it for all our networks as well [83].

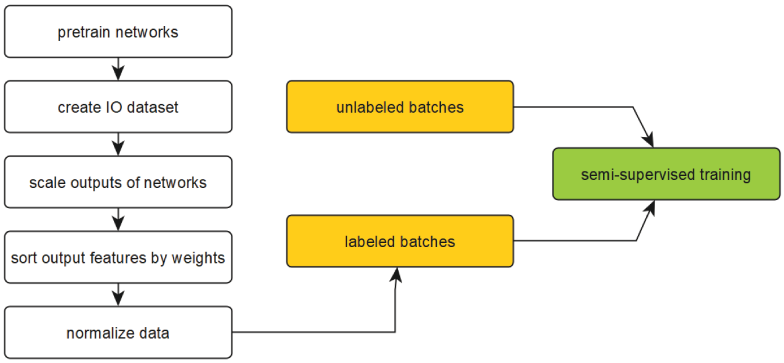

Figure 3.1: The workflow of training info-NAS.

3.1 Network Pretraining

In this section, we first describe the dataset and search space used for pretraining, then we summarize the training details.

3.1.1 CIFAR-10

The CIFAR-10 dataset [84] is an image classification dataset. It consists of 50 000 training images and 10 000 test images with a resolution of 32x32 pixels and 3 channels (RGB). The images are separated into 10 mutually exclusive classes[2], and the dataset is balanced, i.e. both the training and the test set contain the same number of images in each class. The dataset is commonly used in benchmarks.

3.1.2 NAS-Bench-101

NAS-Bench-101 is a tabular benchmark for NAS problems, and also the first publicly available neural architecture dataset. It is an exhaustive dataset, containing 423 thousand unique architectures along with their training and evaluation metrics on CIFAR-10. The authors have identified isomorphic network, so that no two networks of the extracted dataset are isomorphic, and the dataset can be queried using any isomorphic variant of the same network.

[2]The classes are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck.

The search space is cellular, that is, it is a fixed structure with a variable cell motif. The overall structure consists of a *stem*, one 3×3 convolution. Then, we add a stack of three cells followed by a downsampling max pool layer. This sequence is repeated three times and the last time, the global average pooling is applied instead of downsampling. Finally, one dense softmax layer produces the final prediction. All networks produce the same output size regardless of the cell design (Table 3.1).

Table 3.1: NAS-Bench-101 shapes after applying some layers.

layer	channels	width / height
input	3	32
stem	128	32
downsampling 1	256	16
downsampling 2	512	8
global avg. pooling	512	-

The cell can be any valid neural DAG on 7 nodes or less with at most 9 edges, and on every node, an operation is defined. The possible operations are:

- 3×3 convolution

- 1×1 convolution

- 3×3 max pooling

Two of the nodes are labeled IN and OUT instead, representing the input and output nodes. Additionally, every convolution is followed by a batch normalization layer and ReLU activation. The architectures are encodes using binary adjacency matrices and one-hot encoding of the 5 possible operations.

The training hyperparameters are specified in Table 3.2. Also, some standard data augmentation techniques (like random crops and normalization) were used [22]. The results are reported for 3 different training seeds and include the training time and accuracies on training, validation and test sets.

3.1.3 Training Details

From the whole NAS-Bench-101 search space, we sampled 608 networks for the labeled training dataset and 77 networks for the validation dataset — we used the same train-test split as in the original arch2vec, and the validation networks were selected from the test set. Then, we trained them on the CIFAR-10 dataset. Since

Table 3.2: Pretraining hyperparameters for NAS-Bench-101 networks.

hyperparameters	
batch size	256
initial convolution filters	128
validation size	10000
num_epochs	[4, 12, 36, 108]
optimizer	RMSProp
initial learning rate	0.2
final learning rate	0.0
momentum	0.9
weight_decay	10^{-4}
batch normalization momentum	0.997
batch normalization epsilon	10^{-5}
learning rate schedule	cosine annealing

the original implementation of NAS-Bench-101 is in TensorFlow [85], we did not use the original code, but a PyTorch implementation (NASBench-PyTorch[3] [86]).

The training hyperparameters are specified in Table 3.3. The validation set has been split off from the original CIFAR-10 train set. Some of the hyperparameters we used differ from those in the original paper (Table 3.2) due to resource limits — batch size 256 caused memory overflow in some networks, and the original batch normalization and learning rate setup did not produce good results. Moreover, we used the same augmentation techniques. We performed the training on a cluster with the following resources per training process: 16 GB GPU[4], 4 core CPU[5], and 16 GB RAM. Finally, checkpoints of the trained networks were saved for later dataset generation.

3.2 Input-output dataset

The input-output dataset (IO dataset) is created by passing input images into the trained networks and collecting the outputs of some particular layers. There are multiple possibilities what kind of data to extract — first, there are multiple layers to choose from, second, we need to determine the input image dataset.

Generally, for the *input* data, one can either select the input image or some of the intermediate layer outputs — feature maps. Similarly, the *output* data can be either a feature map, the 1-dimensional output of the global average pooling

[3]Used with kind permission of the author, Romulus Hong.

[4]nVidia Tesla T4

[5]Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz, total 16 cores

Table 3.3: Pretraining hyperparameters for NAS-Bench-101 networks — our settings.

hyperparameters	
batch size	128
initial convolution filters	128
validation size	1000
num_workers	4
num_epochs	12
gradient clipping norm	5
optimizer	SGD
initial learning rate	0.025
final learning rate	0.0
momentum	0.9
weight_decay	10^{-4}
learning rate schedule	cosine annealing [87]

or class probabilities. Although the feature maps would be very interesting to study, as the spatial information is preserved, their main disadvantage is that they are very large — using all our training networks, the dataset would easily occupy tens to hundreds of gigabytes.

To simplify the implementation, we have chosen the input images as the *input* data, and the output of the global average pooling layer as the *output* data. The images do not need to be copied, since they are the same for all networks (unlike the feature maps), and the output is a vector of size 512. As so, the dataset represents the outputs of black box feature extractors given input data.

As for the image dataset, we have chosen the validation set of CIFAR-10, with 1 000 images total. The reasons are twofold — one is its relatively small size, the second is that the network outputs could be more variable according to generalization abilities of the network[6]. Table 3.4 summarizes how the train, validation and test datasets are created using the different splits of CIFAR-10 and network dataset. There are two validation datasets — one with seen images and unseen networks, the second with unseen images and seen networks. The latter is created from a fraction of the test set — we selected 2 000 random images from the CIFAR-10 test set (denoted test_2k), and evaluated train networks on it. We used the results of 0.1 of all networks (61 networks) to create the aforementioned validation set, and the remaining networks to create the test set. The second test set are then unseen networks evaluated on unseen images test_2k ('seen' specifies whether the network or image is a part of the training set).

[6]This assumption would be verified in subsequent works.

Table 3.4: Division into train/validation/test datasets according to source dataset types.

labeled	networks	CIFAR-10	dataset size
train	train	validation	608 000
validation	validation	validation	77 000
validation	0.1 train	test_2k	122 000
test	0.9 train	test_2k	1 094 000
test	validation	test_2k	154 000

3.2.1 Data Preprocessing

In this section we describe what preprocessing needs to be done before we can learn from the data. The preprocessing of the CIFAR-10 dataset has been already described in Section 3.1.1, so the *input* data does not have to be modified any further. However, the *output* data cannot be used in a straightforward way for several reasons. We will describe the two main problems.

Sorting The main problem is that there is no ordering defined on the feature maps and because of that, the output of the global average pooling of two different network may be the same, but permuted. Since there is not a simple way to determine the unique permutation of features of each trained network, we need to choose a different approach.

The information we *have* available are the weights — specifically, the weights of the last dense layer. These have a nice property in that they determine the *feature importance* of the global average pooling outputs for a specific class. We use them as follows: for an input image, we choose the dense weights that correspond to its true label. We also append 1 to the vector to represent the bias. Then, we sort the feature vector using the weights — this way, the first feature is the most important one for the corresponding class and so forth. Lastly, we optionally multiply the outputs by weights (in Section 4.1, we compare the results with and without this step).

This preprocessing ensures better comparability between different networks. Of course, not all shuffling errors have been alleviated — two networks may still extract the same feature with a different importance, e.g. as feature number 2 in the first network and feature number 3 in the second one. Nevertheless, two very similar networks should still have a similar response to the same image.

Scaling The second issue is that the networks may produce differently scaled outputs. Since the prediction function is softmax, the final output is normalized

and it does not matter whether the values before applying softmax are large or small. However, in our use-case we need to make the *outputs* comparable.

To alleviate this issue, we fit a scaler for every network separately and *normalize* the features before sorting them. There are multiple ways to do so, e.g. scaling each feature separately or scaling all features by a global value. Some of the variants are explored in an experiment in Section 4.1.

Lastly, we normalize each feature in the final sorted dataset. The main reason to do so is to make every feature contribute to the loss in the same way, but in the aforementioned experiment, we try out both variants.

3.3 Extended Models

In this section, we describe how we have extended the arch2vec model to handle neural architecture data and the IO dataset at the same time. Figure 3.2 shows the overall architecture of the extended info-NAS model.

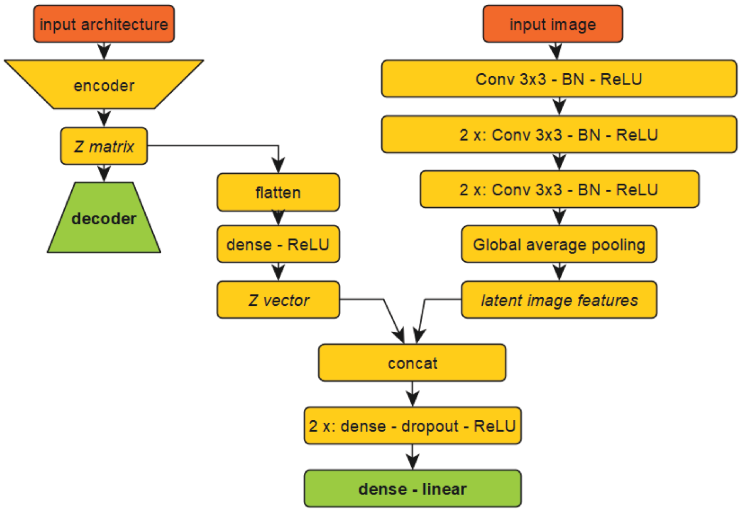

Figure 3.2: The architecture of the info-NAS model.

To encode the images, we use a convolutional layer of kernel size 3×3, then we repeat 3 times two convolutional layers of the same kernel size, where the second has stride 2 to decrease the resolution. Every convolution is followed by a batch normalization layer and ReLU activation. Finally, we use the global average pooling to extract a vector representation.

Then, we connect the two networks together — when we sample the latent matrix $Z \in \mathbb{R}^{n \times l}$ (n is the number of nodes, and l is the dimension of the latent

space) from the original model's encoder, we not only send it to the decoder, but also to the extended part. We first flatten the matrix, so that it has dimensions $Z_f \in \mathbb{R}^{n \cdot l}$, and pass it through a dense layer followed by ReLU activation. Finally, we concatenate the result with the image vector representation, and pass the final vector through two dense layers, each followed by a ReLU and dropout (probablitity of dropout is 0.5), and finally through a linear dense layer to produce the final feature vector.

3.4 Training and Evaluation

Since we have both labeled and unlabeled data, the info-NAS model is trained in a semi-supervised manner — that is, we train it using both labeled and unlabeled data. Since there is more labeled than unlabeled data, we interchangeably train on 300 labeled batches followed by 200 unlabeled batches — we don't want the labeled batches to appear too often, since the labeled networks appear 1000 times (for each image) in the train set [7].

Our approach differs from the main semi-supervised paradigms (Section 1.1) — it is not generative as the reconstructed X and A are not generated using the labels y, and it is not strictly diagnostic, since the *output* features depend on the latent matrix Z instead of X (see architecture of the model in Figure 3.2). The diagnostic paradigm has been shown to be less powerful than the generative one [7], as the unlabeled samples do not change the classifier part in any way. However, since our main aim is to improve the latent representation Z and *not* to reconstruct the *output* features as accurately as possible (although we still want our model to have some generalization capabilities), this modified diagnostic approach is sufficient.

We use the same model and training hyperparameters as in the original arch2vec. As for the parameters specific for our model, in the following chapter we explore what labeled losses work the best with the predicted *outputs* (MSE or L1), weighting of both the original VAE loss and the labeled loss, and whether a different learning rate on labeled batches yields better results. The full list of hyperparameters is tabularized in the same chapter for convenience (Table 4.1).

[7] Some better sampling strategies will be explored in followup works.

Chapter 4

Experiments

In this section, we describe the experiments with info-NAS that we have done. In the first experiment (Section 4.1), we tried out different scaling and hyperparameter settings of info-NAS. Then, in Section 4.2, we analyze how the info-NAS model behaves — specifically, we analyze the distribution of losses during the training (Section 4.2.1), then we visualize how the *output* features info-NAS learns look like (Section 4.2.2) and lastly, we compare the performance of info-NAS with the original arch2vec model 4.2.3. Finally, we evaluate the performance of info-NAS on the whole NAS problem — we run the reinforce search on the NAS-Bench-101 search space (Section 4.3), and we train and evaluate performance predictors on the latent features (Section 4.4).

Table 4.1: Default info-NAS hyperparameters

hyperparameter	
batch size	32
epochs	30
num workers	8
checkpoint frequency	5
loss	L1
VAE loss weight	1.0
labeled loss weight	1.0
optimizer	Adam
learning rate	10^{-3}
betas	[0.9, 0.999]
eps	10^{-8}
scale axis	null
scale weighted	false
multiply by weights	true

Table 4.1 summarizes the hyperparameters of info-NAS used for all visual-

izations and in all experiments. In the scalings and hyperparameter experiment (Section 4.1), we try out different values for some of the default parameters, but all other settings remain the same. The specific parameter changes are indicated in the corresponding results tables. All runs that involved training of info-NAS or of the reference arch2vec were executed on the same cluster as in the pre-training (Section 3.1.3), but the number of reserved CPUs was increased to 8. The experiment running time for 30 epochs was between 5 to 6 hours. As for the default training parameters, most of them were set to be the same as arch2vec, and the parameters specific to info-NAS were selected according to some early results. While the chosen scaling setting has proven to be the best one (Section 4.1), it is possible that some different loss would yield better and more stable results (e.g. the Huber loss). The L1 loss (also called mean absolute error) has been chosen since it is less sensitive to outliers with large outputs, and initially produced better results than the MSE.

In the experiments, we use the IO datasets defined in Section 3.2 — a train set, two validation sets (unseen images, unseen networks) and two test sets (larger set of unseen images, unseen networks and images).

Reference When running info-NAS, the labeled loss is a regression loss between the predicted and original features. However, since the task of predicting network features is a novel approach, we lack a suitable baseline to compare with. Therefore, for every result on a dataset (either train, both validations and test sets), we report a *baseline reference* loss computed as follows: we compute the mean feature vector using all *output* data in the dataset (after applying the scaling and sorting transformations). Then, for every batch we compute the loss between a batch of the mean vectors and the batch itself. Finally, we compute summary statistics (mean, std, min, max, median) and report them as the baseline loss. Additionally, in some experiments we compute the 95 % confidence interval using the sample mean and std (n is the number of batches, batch size is the same as in training):

$$x = \mu \pm \frac{\sigma}{\sqrt{n}}.$$

4.1 Info-NAS Scaling and Hyperparameter Settings

The goal of this experiment is to explore how different scaling approaches, as well as some hyperparameter settings, influence the results of info-NAS. The experiment is divided into three sections: scaling, training parameters, and loss weighting. In every section, we present and discuss the following metrics:

- labeled losses — training loss, unseen image validation loss, unseen network validation loss
- training unlabeled losses — reported separately on labeled and unlabeled batches
- validation reconstruction accuracy of node operations and adjacency matrix — reported separately on labeled and unlabeled batches

We do not report uniqueness and validity in this experiment, because uniqueness stays close to 1.0 for all experiments, and validity changes only in the loss weighting experiment, but in the same way as reconstruction accuracy. Originally, the experiment was expected to run for multiple seeds, however we reached the resource quota limits after the second seed. Therefore, we report results of seed=1 only, and indicate only some noticeable ranking differences between the runs. In this experiment, we report the results for epoch 8.

4.1.1 Scaling Experiment

In this experiment, we tried to determine which scaling approach works best. There were 3 different parameters, each with 2 different values with the following meanings:

- axis 0 — each feature is normalized separately
- axis null — features are normalized using the mean and std of the whole *output* dataset
- weighted true — weight the features before computing the statistics
- weight false — compute statistics from raw features
- multiply true — multiply the final output by the weights
- multiply false — output the raw features.

Since the weights are specific for every class, we compute the statistics 10 times for each label, using the whole dataset every time. Then, every image is scaled using the values of its true label. When using the weighted variant, we multiply the output by weights, so not all settings are valid — in total, we run the experiment for 6 different settings.

In Tables 4.2, 4.3 and 4.4, we report the VAE loss and labeled L1 loss on the train set and on the two validation sets. Noticeably good results are bold, similarly runs with a poor performance are in italics. The reconstruction accuracies are not reported in a separate table, since the metrics were very similar, and the reconstruction performance was not damaged by the scaling approaches. The same holds for the validity of the architectures.

Table 4.2: Training loss of info-nas with different scaling approaches.

scale parameters			VAE loss		labeled loss		
axis	weighted	multiply	labeled	unlabeled	loss	train	reference
0	*false*	*false*	*0.2154*	*0.2292*	*L1*	*0.0017*	0.0017 ± 0.0000
0	false	true	0.2173	0.2309	L1	0.0436	0.0855 ± 0.0000
0	true	true	0.2139	0.2266	L1	0.6474	0.7131 ± 0.0001
null	false	false	0.2182	0.2280	L1	1.8351	2.0912 ± 0.0004
null	**false**	**true**	**0.2324**	**0.2355**	**L1**	**0.1815**	$\mathbf{0.7656 \pm 0.0003}$
null	true	true	0.2167	0.2285	L1	0.5520	0.6691 ± 0.0002

Table 4.3: Validation loss of info-nas with different scaling approaches — unseen architectures.

scale parameters			labeled loss		
axis	weighted	multiply	loss	unseen networks	reference
0	*false*	*false*	*L1*	0.0016 ± 0.0000	0.0016 ± 0.0000
0	*false*	*true*	*L1*	117.4418 ± 5.5340	234.3804 ± 5.5087
0	true	true	L1	0.6637 ± 0.0006	0.6980 ± 0.0005
null	false	false	L1	1.9465 ± 0.0021	2.0696 ± 0.0017
null	**false**	**true**	**L1**	$\mathbf{0.3824 \pm 0.0020}$	$\mathbf{0.7106 \pm 0.0037}$
null	true	true	L1	0.6012 ± 0.0007	0.6717 ± 0.0007

Table 4.4: Validation loss of info-nas with different scaling approaches — unseen images.

scale parameters			labeled loss		
axis	weighted	multiply	loss	unseen images	reference
0	*false*	*false*	*L1*	0.0018 ± 0.0000	0.0018 ± 0.0000
0	false	true	L1	0.0448 ± 0.0000	0.0877 ± 0.0000
0	true	true	L1	0.7331 ± 0.0007	0.7707 ± 0.0003
null	false	false	L1	2.1036 ± 0.0024	2.2572 ± 0.0011
null	**false**	**true**	**L1**	$\mathbf{0.2106 \pm 0.0008}$	$\mathbf{0.7254 \pm 0.0006}$
null	true	true	L1	0.6463 ± 0.0008	0.7404 ± 0.0004

In Table 4.2, we see that in all cases except for the first setting, the network managed to learn on the train data. The approach with global scaling, no weighting and multiplied outputs performed noticeably well. On unseen networks (Table 4.3), both experiments with scaling per feature, no weighting, performed poorly — in the first case, the loss did not differ from the reference, and in the second case, the validation loss was much higher than the training loss on the same settings. Lastly, the results on unseen images (Table 4.4) were very similar to the training losses and references.

We can draw two conclusions from this experiment. The first is that when scaling each feature separately without taking the weights into account, on new networks the scale is too different to achieve good results. The second is that the approach where we scale the outputs of a network by the global mean and std, scale the original *output* data, and afterwards multiply it by weights (according to true labels) works the best. Another nice property is that all three losses have a similar baseline loss.

4.1.2 Training Parameters

This experiment had the goal to compare two different losses (MSE and L1) along with different learning rates. The results have the same format as in the scaling experiment (here Tables 4.6, 4.7 and 4.8), and we additionaly summarize the reconstruction accuracies in Table 4.5.

Table 4.5: Reconstruction accuracies on architectures from labeled and unlabeled validation sets (training params experiment).

training parameters		operators accuracy		adjacency accuracy	
loss	learning rate	labeled	unlabeled	labeled	unlabeled
L1	10^{-2}	0.9777	0.9777	0.9462	0.9422
L1	10^{-3}	0.9777	0.9778	0.9666	0.9669
L1	10^{-4}	0.9777	0.9779	0.9882	0.9848
MSE	10^{-2}	0.9777	0.9777	0.9326	0.9243
MSE	10^{-3}	0.9777	0.9778	0.9858	0.9815
MSE	10^{-4}	0.9777	0.9779	0.9932	0.9877

The reconstruction accuracy of operators was the same for all settings, while the reconstruction accuracy of the adjacency matrix increased with a smaller learning rate. This is probably because the adjacency matrix depends directly on the latent space (no weights are learnt), while the ops are reconstructed through a learnable dense layer. The MSE loss combined with the two smaller learning rates performed better than the L1 loss for this seed, but in the second seed the L1 loss matched the reconstruction accuracy of MSE.

Table 4.6: Comparison of some training parameters — training loss.

training parameters		VAE loss		labeled loss		
loss	learning rate	labeled	unlabeled	loss	train	reference
L1	10^{-2}	*0.2253*	*0.2611*	*L1*	*0.7542*	0.7656 ± 0.0003
L1	10^{-3}	0.2343	0.2389	L1	0.1818	0.7656 ± 0.0003
L1	10^{-4}	0.2283	0.2293	L1	0.1771	0.7656 ± 0.0003
MSE	10^{-2}	0.2664	0.2769	MSE	0.2836	1.0002 ± 0.0007
MSE	10^{-3}	0.2260	0.2325	MSE	0.0832	1.0002 ± 0.0007
MSE	10^{-4}	0.2229	0.2258	MSE	0.0804	1.0002 ± 0.0007

Table 4.7: Comparison of some training parameters — validation loss (unseen networks).

training parameters		labeled loss			
loss	learning rate	loss	unseen networks	reference	L1 loss
L1	10^{-2}	*L1*	0.7046 ± 0.0040	0.7106 ± 0.0037	*0.7046*
L1	10^{-3}	L1	0.4080 ± 0.0022	0.7106 ± 0.0037	0.4080
L1	10^{-4}	L1	0.4165 ± 0.0025	0.7106 ± 0.0037	0.4165
MSE	10^{-2}	MSE	0.3603 ± 0.0032	0.8346 ± 0.0080	0.4558
MSE	10^{-3}	MSE	0.2894 ± 0.0030	0.8346 ± 0.0080	0.3922
MSE	$\mathbf{10^{-4}}$	**MSE**	$\mathbf{0.1964 \pm 0.0015}$	$\mathbf{0.8346 \pm 0.0080}$	**0.3246**

Table 4.8: Comparison of some training parameters — validation loss (unseen images).

training parameters		labeled loss			
loss	learning rate	loss	unseen networks	reference	L1 loss
L1	10^{-2}	*L1*	0.7256 ± 0.0034	0.7254 ± 0.0006	*0.7256*
L1	10^{-3}	L1	0.2292 ± 0.0009	0.7254 ± 0.0006	0.22925
L1	$\mathbf{10^{-4}}$	**L1**	$\mathbf{0.1369 \pm 0.0002}$	$\mathbf{0.7254 \pm 0.0006}$	**0.1369**
MSE	10^{-2}	MSE	0.3400 ± 0.0035	0.9585 ± 0.0019	0.4015
MSE	10^{-3}	MSE	0.1506 ± 0.0009	0.9585 ± 0.0019	0.2603
MSE	10^{-4}	MSE	0.0764 ± 0.0003	0.9585 ± 0.0019	0.1410

The L1 loss with learning rate 10^{-2} performed poorly in this seed, not even surpassing the baseline, but in the second seed (seed=2), it matched the performance of the MSE loss with the same learning rate (in this experiment). Similarly, the MSE loss with learning rate 10^{-4} performed exceptionally well for this seed, but in the second case it did not perform well (e.g. on unseen networks the loss was over 0.3). The reason for this behavior could be that for larger learning rates, it sometimes fails to learn from the train set, and for the smaller rates we risk

Table 4.9: Reconstruction accuracies on architectures from labeled and unlabeled validation sets (loss weighting experiment).

loss weights		operators accuracy		adjacency accuracy	
VAE weight	labeled weight	labeled	unlabeled	labeled	unlabeled
1.0	0.5	0.9777	0.9779	0.9858	0.9804
1.0	1	0.9777	0.9778	0.9839	0.9748
1.0	3	0.9777	0.9778	0.9777	0.9725
1.0	50	0.9777	0.9775	0.9499	0.9387
10^{-2}	0.5	0.9777	0.9777	0.9703	0.9595
10^{-2}	1	0.9777	0.9776	0.9518	0.9487
10^{-2}	3	0.9777	0.9775	0.9246	0.9177
10^{-2}	50	0.9777	0.9766	0.8806	0.8645
10^{-4}	0.5	0.9777	0.9774	0.9085	0.8972
10^{-4}	1	0.9777	0.9776	0.9208	0.9148
10^{-4}	3	0.9777	0.9767	0.8596	0.8530
10^{-4}	50	0.9777	0.9772	0.9239	0.9069

staying in a local minimum. The learning rate 10^{-3} still produces different results across different seeds, but the difference between the results is smaller than 0.1 (in Section 4.2.3, we run an experiment with this setting for multiple seeds).

The conclusion is that learning rates 10^{-3} and 10^{-4} work well, and although the learning rate 10^{-4} achieves sometimes great results, its variance is too big, and 10^{-3} is more suitable. The comparison between L1 and MSE losses is inconclusive, and more experiments are possibly needed, along with some other losses.

4.1.3 Loss Weighting

The goal of this experiment was to try to weight both parts of the losses — the VAE loss and the labeled loss. The weighting of the VAE loss (by a value less than 1) could make overfitting on the overrepresented labeled architectures less likely, and improve the generalization on unlabeled architectures. We also weight the labeled loss as to see if it leads to any improvement in the labeled validation losses. The results are tabelated in Tables 4.9, 4.10, 4.11, 4.12 (same format as in the previous experiment). Instead of printing the loss values that were actually used in the training, the losses are scaled back for comparability (i.e. for labeled weight 50, the loss is divided by 50).

Overall, none of the results except for no weights yielded any good results. The weights were often detrimental to the VAE loss of labeled batches (along with reconstruction accuracies and validity), and no improvement in the labeled validation losses occured.

Table 4.10: Loss weighting experiment — train loss.

loss weights		VAE loss		labeled loss		
VAE weight	labeled weight	labeled	unlabeled	loss	train	reference
1.0	0.5	0.2251	0.2329	L1	0.1864	0.7656 ± 0.0003
1.0	1	0.2330	0.2377	L1	0.1788	0.7656 ± 0.0003
1.0	3	0.2471	0.2400	L1	0.1749	0.7656 ± 0.0003
1.0	50	0.3279	0.2554	L1	0.1772	0.7656 ± 0.0003
10^{-2}	0.5	0.3262	0.2512	L1	0.1796	0.7656 ± 0.0003
10^{-2}	1	0.3462	0.2547	L1	0.1773	0.7656 ± 0.0003
10^{-2}	3	0.4162	0.2646	L1	0.1796	0.7656 ± 0.0003
10^{-2}	50	0.5191	0.2786	L1	0.1861	0.7656 ± 0.0003
10^{-4}	0.5	0.3720	0.2595	L1	0.1814	0.7656 ± 0.0003
10^{-4}	1	0.3372	0.2562	L1	0.1797	0.7656 ± 0.0003
10^{-4}	3	0.4011	0.2638	L1	0.1808	0.7656 ± 0.0003
10^{-4}	50	0.3938	0.2635	L1	0.1812	0.7656 ± 0.0003

Table 4.11: Loss weighting experiment — validation loss (unseen networks).

loss weights		labeled loss		
VAE weight	labeled weight	loss	unseen networks	reference
1.0	0.5	L1	0.3861 ± 0.0010	0.7106 ± 0.0037
1.0	1	L1	0.3598 ± 0.0016	0.7106 ± 0.0037
1.0	3	L1	0.3856 ± 0.0064	0.7106 ± 0.0037
1.0	50	L1	0.4287 ± 0.1257	0.7106 ± 0.0037
10^{-2}	0.5	L1	0.3817 ± 0.0011	0.7106 ± 0.0037
10^{-2}	1	L1	0.3905 ± 0.0021	0.7106 ± 0.0037
10^{-2}	3	L1	0.3978 ± 0.0067	0.7106 ± 0.0037
10^{-2}	50	L1	0.3746 ± 0.0869	0.7106 ± 0.0037
10^{-4}	0.5	L1	0.4273 ± 0.0011	0.7106 ± 0.0037
10^{-4}	1	L1	0.3306 ± 0.0015	0.7106 ± 0.0037
10^{-4}	3	L1	0.3361 ± 0.0051	0.7106 ± 0.0037
10^{-4}	50	L1	0.3426 ± 0.0834	0.7106 ± 0.0037

Table 4.12: Loss weighting experiment — validation loss (unseen images).

loss weights		labeled loss		
VAE weight	labeled weight	loss	unseen images	reference
1.0	0.5	L1	0.2320 ± 0.0005	0.7254 ± 0.0006
1.0	1	L1	0.2128 ± 0.0009	0.7254 ± 0.0006
1.0	3	L1	0.1822 ± 0.0022	0.7254 ± 0.0006
1.0	50	L1	0.2082 ± 0.0425	0.7254 ± 0.0006
10^{-2}	0.5	L1	0.1775 ± 0.0003	0.7254 ± 0.0006
10^{-2}	1	L1	0.1717 ± 0.0006	0.7254 ± 0.0006
10^{-2}	3	L1	0.1752 ± 0.0023	0.7254 ± 0.0006
10^{-2}	50	L1	0.1678 ± 0.0176	0.7254 ± 0.0006
10^{-4}	0.5	L1	0.1654 ± 0.0002	0.7254 ± 0.0006
10^{-4}	1	L1	0.1784 ± 0.0005	0.7254 ± 0.0006
10^{-4}	3	L1	0.1701 ± 0.0009	0.7254 ± 0.0006
10^{-4}	50	L1	0.1632 ± 0.0158	0.7254 ± 0.0006

4.2 Info-NAS Analysis

For the experiments in this section, we ran the info-NAS training for 10 different seeds (seed=1 to 10). Compared to the previous runs, we also trained the original arch2vec for reference, training on neural architectures from both labeled and unlabeled batches. The experiment is divided into 3 parts: we first examine the losses of info-NAS, then we visualize the learned *output* features, and finally we compare the model with the reference arch2vec.

4.2.1 Labeled Loss

In Figures 4.1, 4.2 and 4.3, we summarize the losses (train, validation unseen networks, validation unseen images) for the 10 different seeds during the 30 epochs. The bold line in the figures represents the mean value of the loss, and the shadow represents the bootstrapped 95 % confidence interval. For validation losses, we report the median loss as well, since it is less sensitive to outliers.

The training loss (Figure 4.1) and the unseen images validation loss (Figure 4.3) behave in a similar way during the training process — they both logarithmically decrease, and although the mean validation loss is still quite high, the median loss is very small and with little variations. The validation loss on unseen networks (Figure 4.2) remains quite high — this may be caused by the diagnostic approach to semi-supervised learning, where the unlabeled examples do not help directly to improve the classifier. Another possibility is that the ability to generalize on new networks is limited. Nevertheless, all of the three losses overpass the baseline loss, although the loss on unseen networks is closer to the

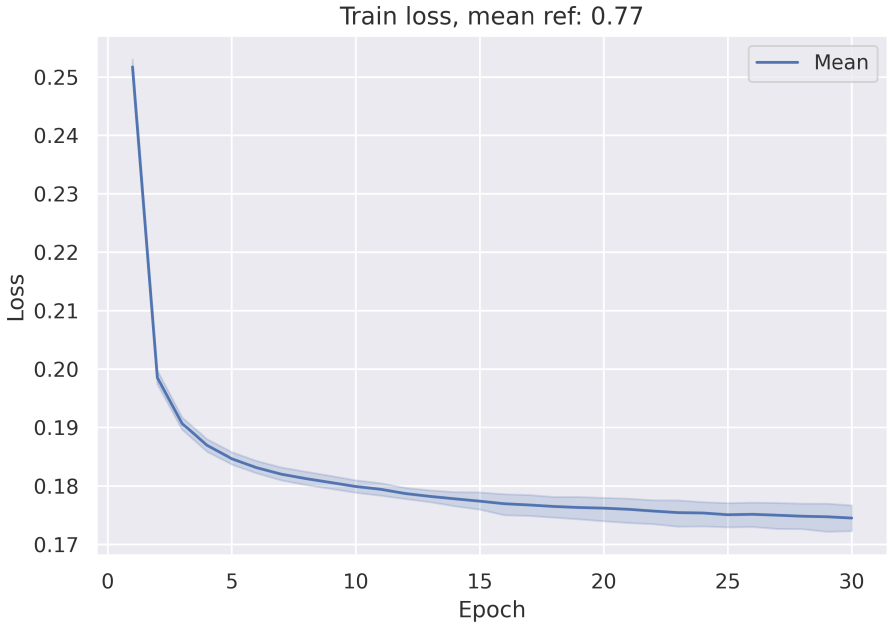

Figure 4.1: Mean training loss for 10 seeds.

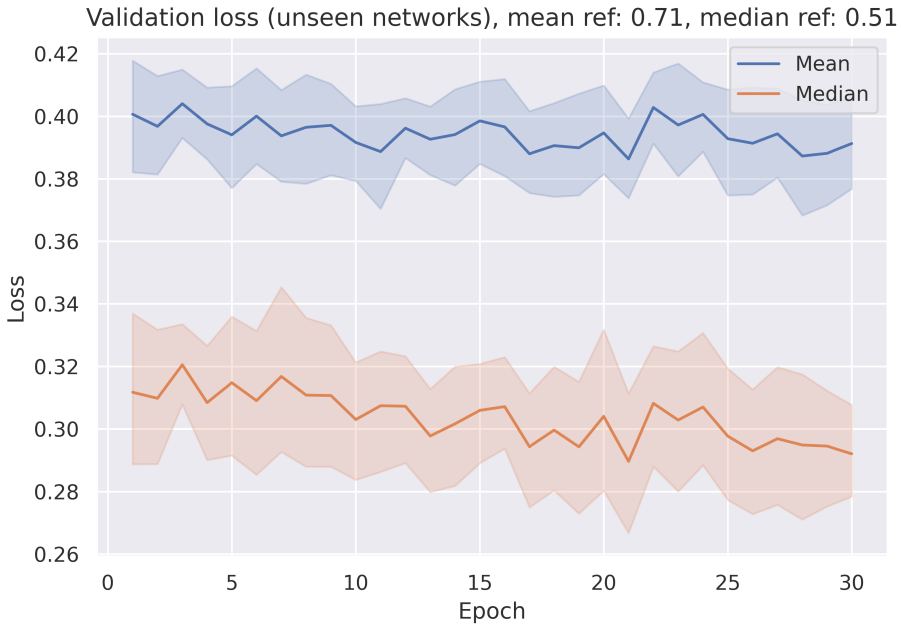

Figure 4.2: Validation loss un unseen networks during the training for 10 seeds.

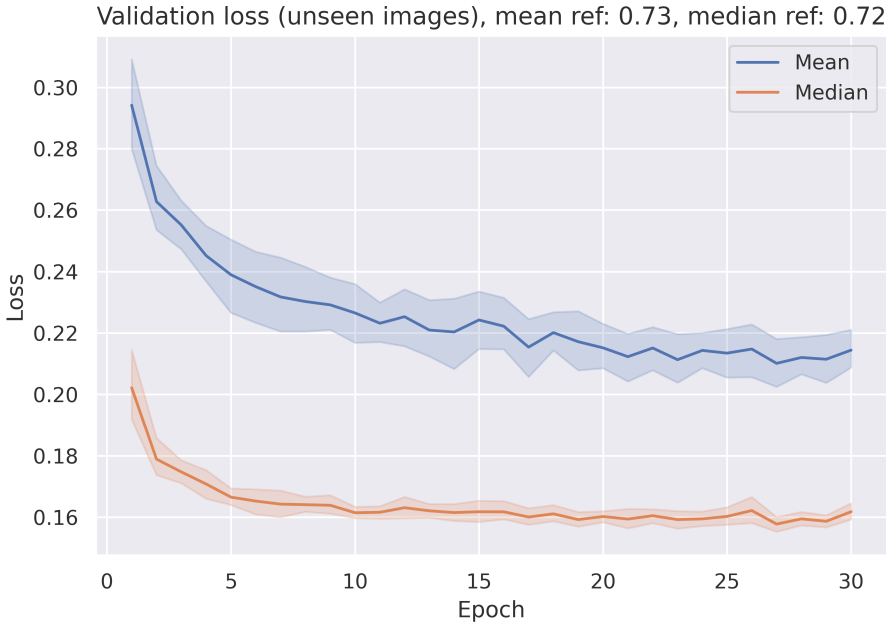

Figure 4.3: Validation loss un unseen images during the training for 10 seeds.

Figure 4.4: Distribution of the test loss for 10 seeds (unseen images).

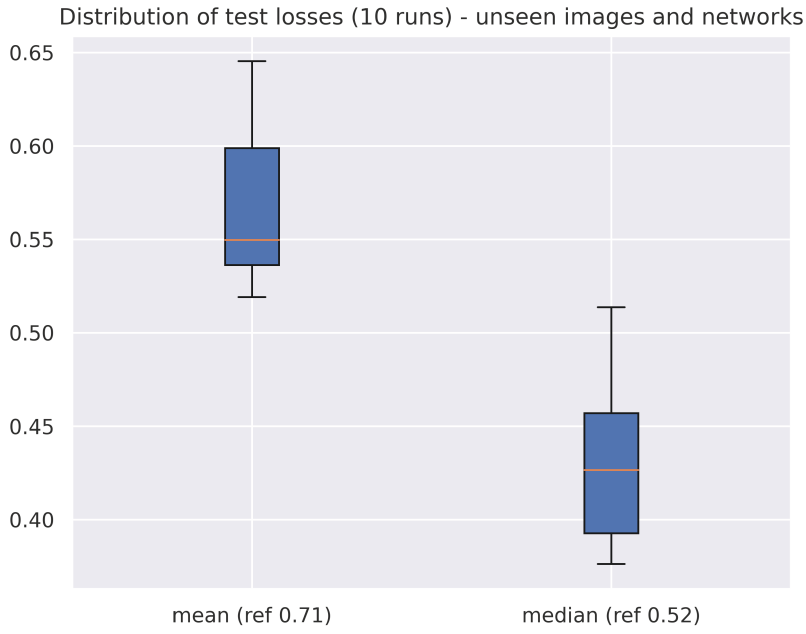

Figure 4.5: Distribution of the test loss for 10 seeds (unseen images and networks).

baseline than the other two.

After the training, we also evaluated the 10 trained networks on the two test sets (unseen images, unseen images and networks), and the distribution of the losses is depicted in Figures 4.4 and 4.5. On unseen images, the result is worse than for the smaller subset that was used as a validation set, it is still better than the baseline though. A possible explanation is that the networks used for the validation set were more similar to the train set than those in the test set. As for the unseen images and unseen networks, although the mean loss is slightly better than the baseline, on some seeds, the median loss was worse than the baseline. Since this is the hardest task for the model and the validation networks did not perform so well even on seen images, it is not a surprising result. The generalization on new networks should be an important aspect of followup works.

4.2.2 Feature Visualizations

In this section, we visualize the learned *output* features for both training data and unseen data. Figure 4.6 depicts the features of 50 chosen train networks on a random image, and Figure 4.7 shows features of one chosen network across all

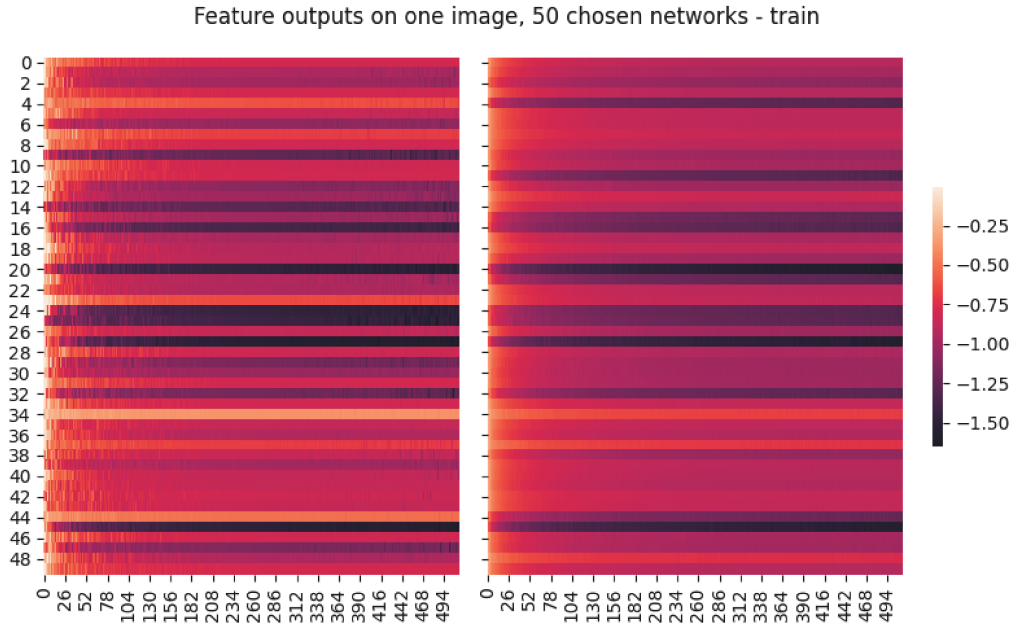

Figure 4.6: Feature outputs of 50 chosen networks on one image — train set.

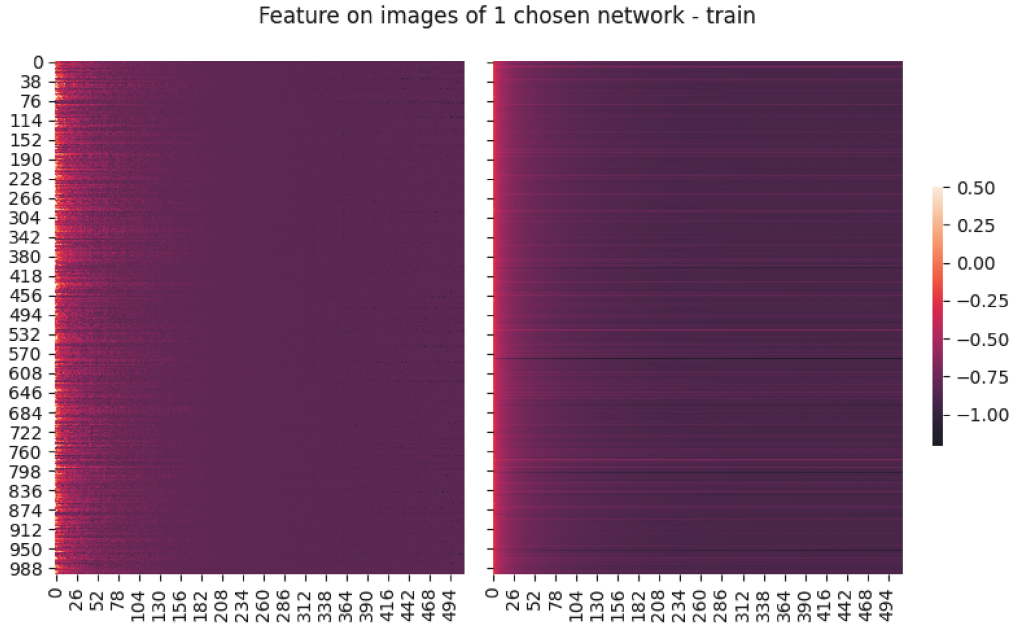

Figure 4.7: Feature outputs of 1 network on all images — train set.

training data. In all images, the left features are the original *outputs*, and the right

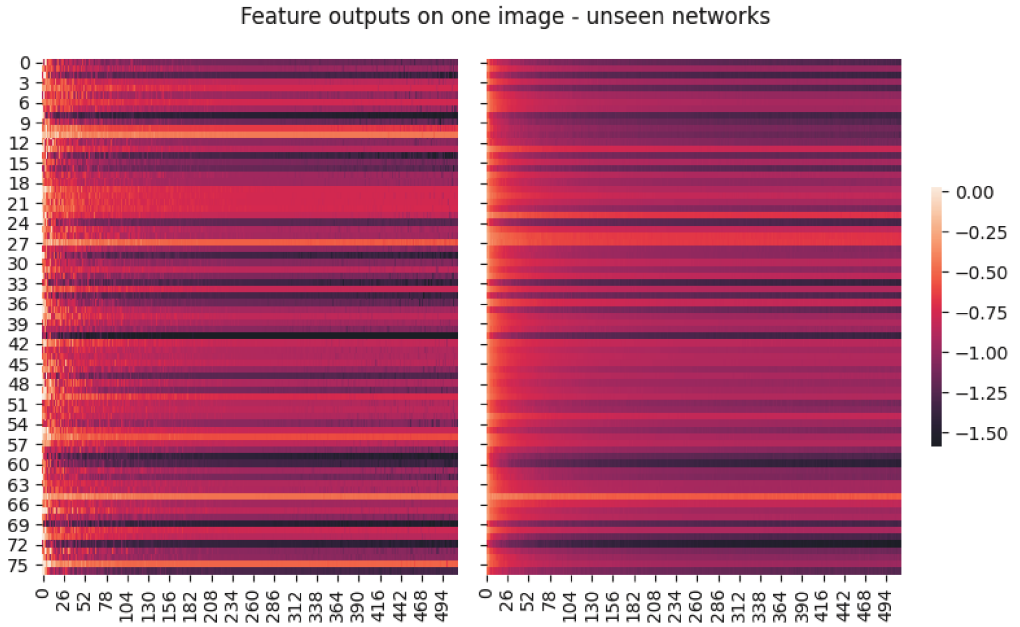

Figure 4.8: Feature outputs of 50 chosen networks on one image — unseen networks.

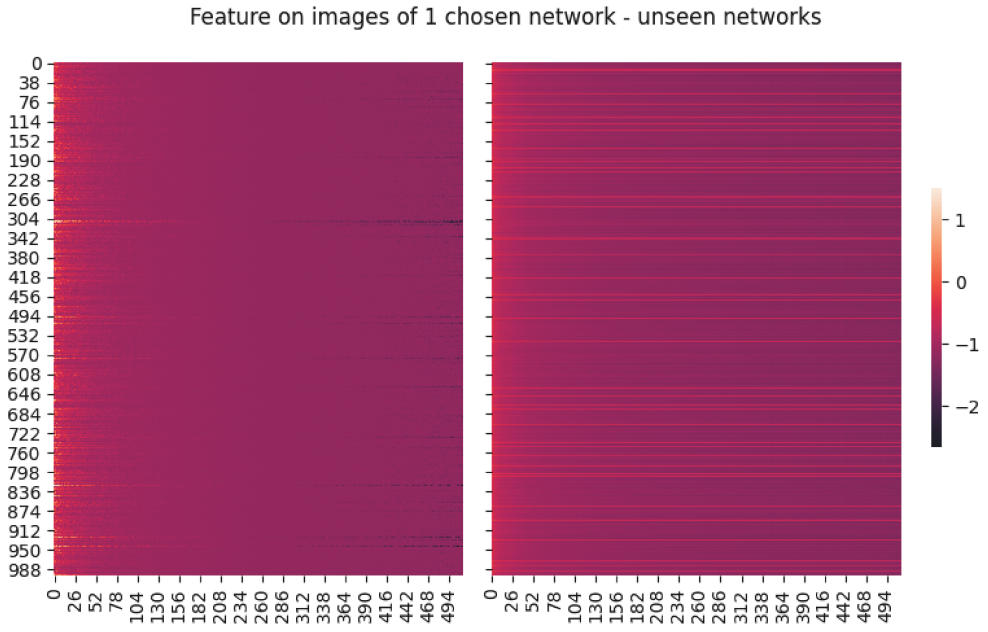

Figure 4.9: Feature outputs of 1 network on all images — unseen networks.

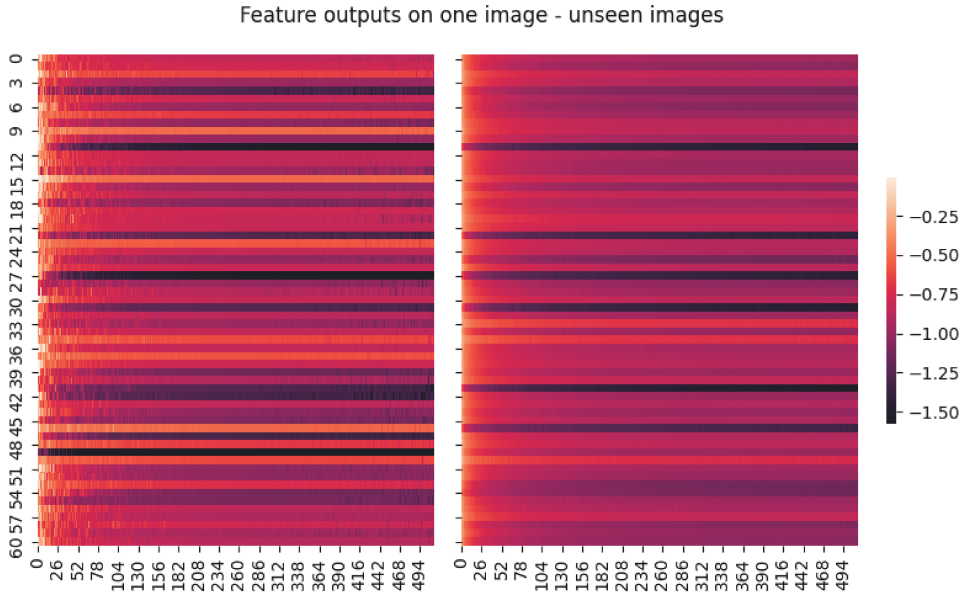

Figure 4.10: Feature outputs of 50 chosen networks on one image — unseen images.

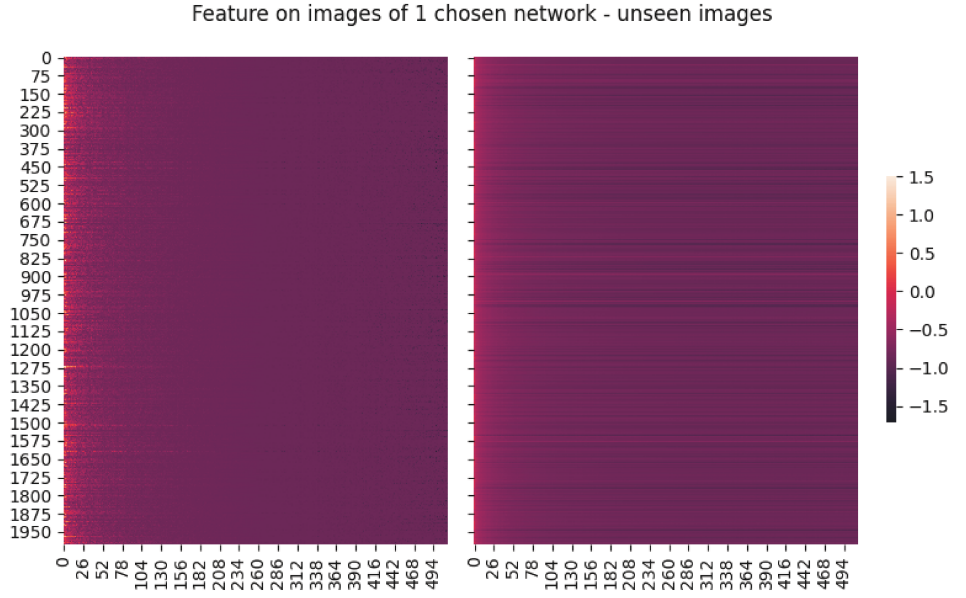

Figure 4.11: Feature outputs of 1 network on all images on all images — unseen images.

features are the predicted *outputs*. Since the features of the whole dataset were normalized, we scaled every feature back for better visualization purposes (not including the per-network scale from the first experiment). In the figure with 50 networks, we see that the model still makes some training errors, and the figure with one network shows that the model does not distinguish the input images very well.

The first problem could be targeted by a more powerful model, or by focusing on the input architectures with a large error. The second issue is a complex one — although the sorting by weights helped the model learn the signature features of a input network, it is unclear whether it suffices for learning fine-grained outputs per image, since the output is still quite noisy. Possible solutions to this problem could include a more elaborate loss function (ideally not overly dependent on feature position in the *outputs* vector), or a new model that tries to predict only one feature vector per neural architecture — e.g. its weights, or the mean of the *outputs* over all images.

Figures 4.8 and 4.9 show the same results on unseen networks, and Figures 4.10 and 4.11 for unseen images. We see that the model is still able to predict the unique features of an unseen network, and although the number of errors is large, the result could be improved along with a better performance on the train set, or maybe by using more labeled data. We can also see that the model makes more errors for different images passed to a new network, so maybe the combination of the image and network feature vectors could be improved as well. The results on unseen images are similar to the train set.

4.2.3 Comparison with Arch2vec

In this experiment we compare info-NAS with the reference arch2vec model. Both models were trained side by side on the same batches, with arch2vec handling labeled batches as unlabeled (ignoring IO information). For both models, we report mean results across 10 runs in the same way as in the previous experiment (Section 4.2.1.

The VAE loss is higher for Info-NAS on both types of batches (Figure 4.12). This may be due to the fact that Info-NAS has to optimize two parts of the model at the same time, while arch2vec focuses entirely on VAE loss. The difference is however very small, and looking at the validation metrics — reconstruction accuracy on operations and adjacency matrix (Figures 4.15 and 4.16) — the performance is practically the same. The mean validity (Figure 4.13) is larger for info-NAS, however the result is not statistically significant. Also, since the mean uniqueness (Figure 4.14) is *smaller* for info-NAS, a possible explanation for this behavior is that info-NAS outputs frequently the overrepresented labeled networks, since the latent space was tuned for the classifier performance on labeled

examples. However, this claim is yet to be supported with additional experiments.

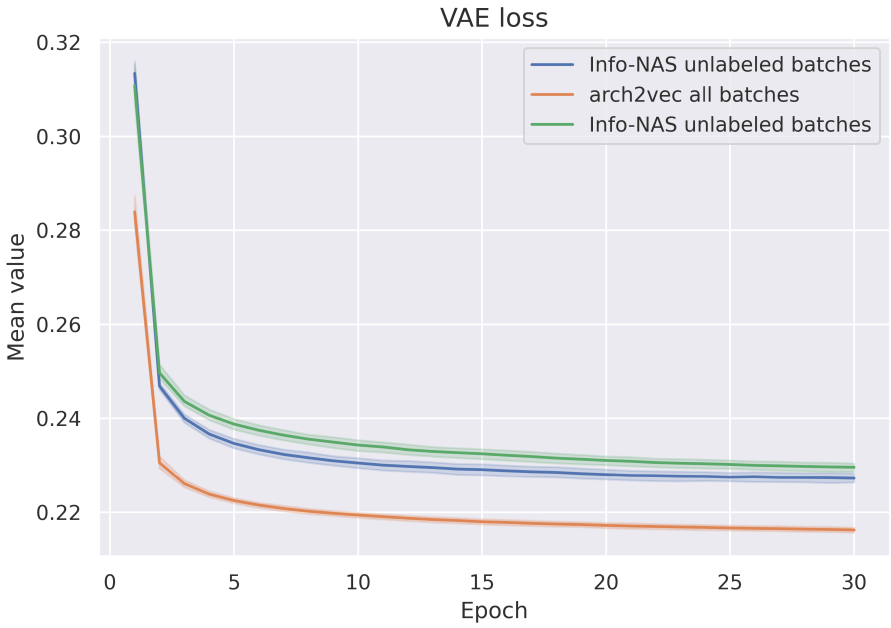

Figure 4.12: VAE loss on different batches for 10 seeds.

Now, we compare how the latent spaces of the two models look like. We use the algorithm T-SNE on 10 000 randomly sampled architectures for visualization [88]. We use the plotting function from the arch2vec repository. Figures 4.17 and 4.18 show results for info-NAS and arch2vec features extracted after epoch 10, and Figures 4.17 and 4.18 show results for epoch 30. The results look similar, although info-NAS seems to form large 'islands' of good performing architectures better (particularly noticeable for the case after epoch 30). However, it also splits off a small cluster of architectures — the same slightly happens for arch2vec as well, but the border is barely noticeable, while info-NAS has 2 separated clusters after 30 epochs. This may point to possible overfitting on labeled architectures that we discussed above.

4.3 Search Results on NAS-Bench-101

In this experiment, we ran the REINFORCE search on NAS-Bench-101 (and CIFAR-10 dataset) using extracted features by arch2vec or by info-NAS. We used two sets of latent features for each model — one set extracted after 10 epochs of

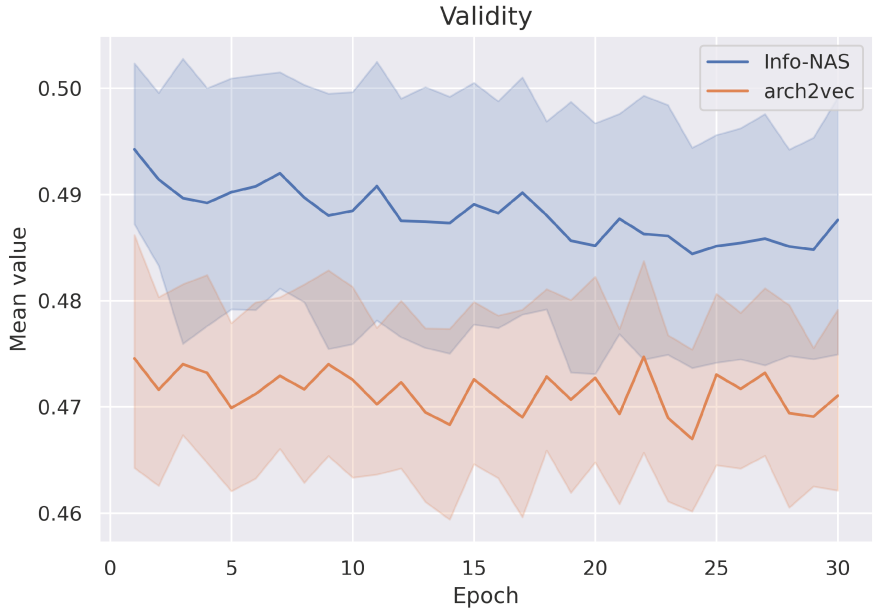

Figure 4.13: Comparison of validity on info-NAS and arch2vec for 10 seeds.

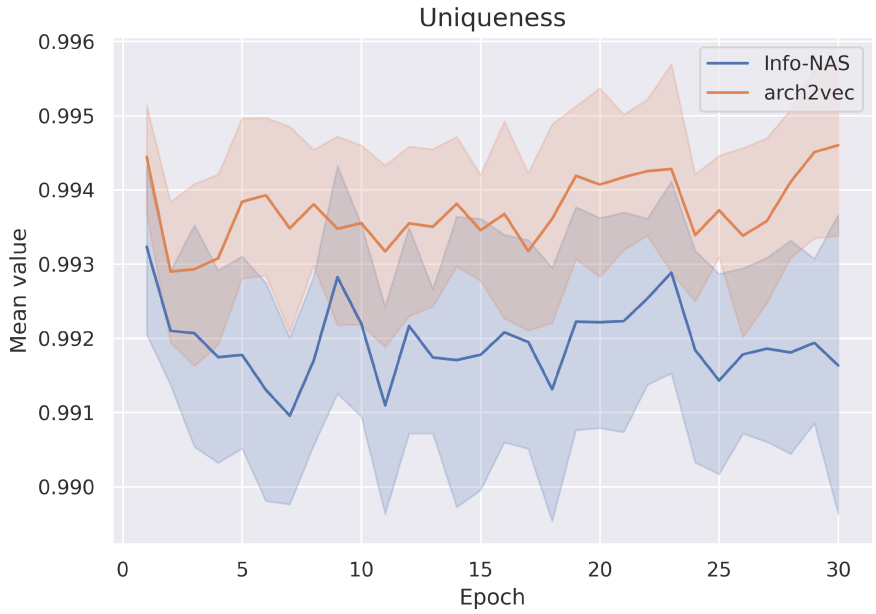

Figure 4.14: Comparison of uniqueness on info-NAS and arch2vec for 10 seeds.

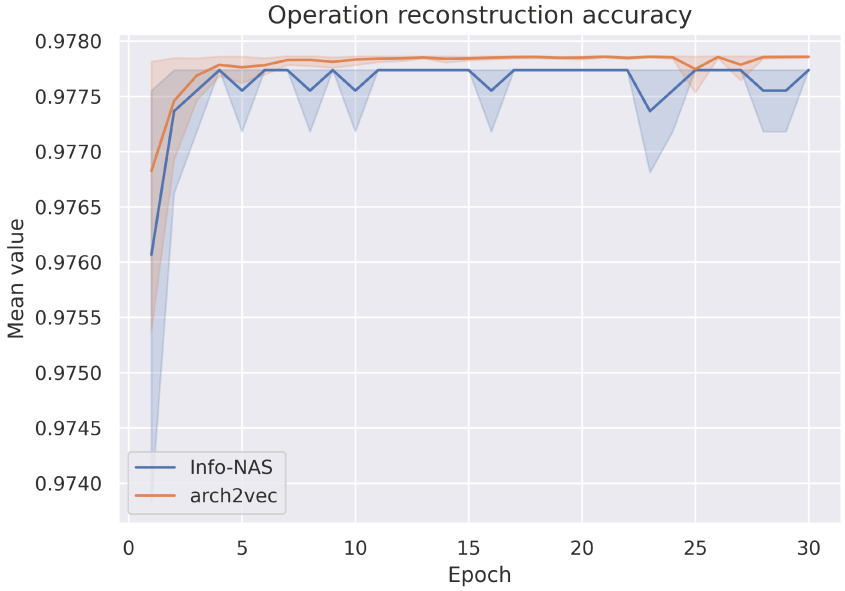

Figure 4.15: Comparison of operations accuracy on info-NAS and arch2vec for 10 seeds.

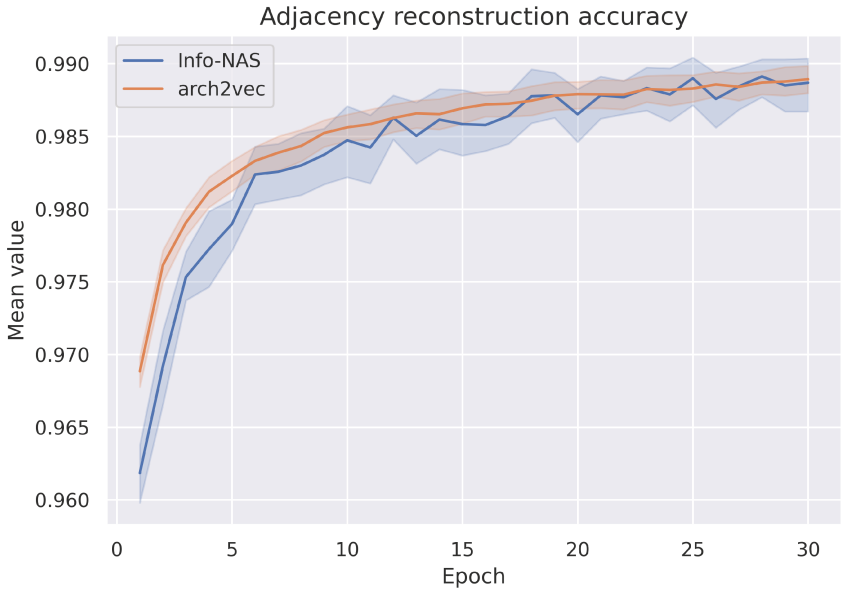

Figure 4.16: Comparison of adjacency accuracy on info-NAS and arch2vec for 10 seeds.

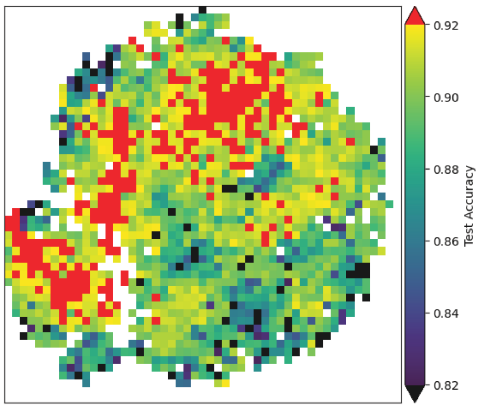

Figure 4.17: Info-NAS after 9 epochs, T-SNE on 10 000 architectures.

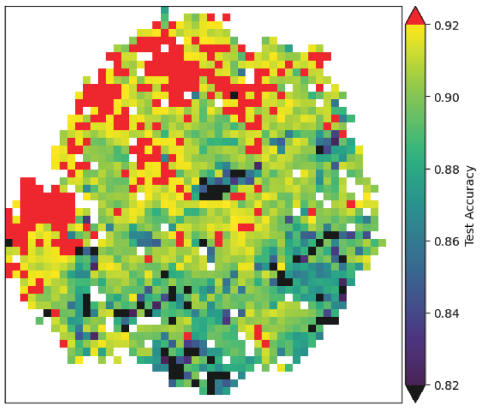

Figure 4.18: Arch2vec after 9 epochs, T-SNE on 10 000 architectures.

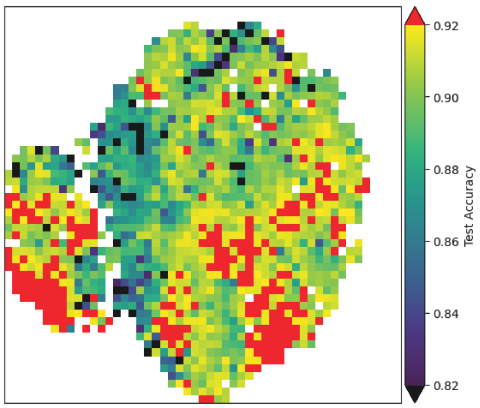

Figure 4.19: Info-NAS after 29 epochs, T-SNE on 10 000 architectures.

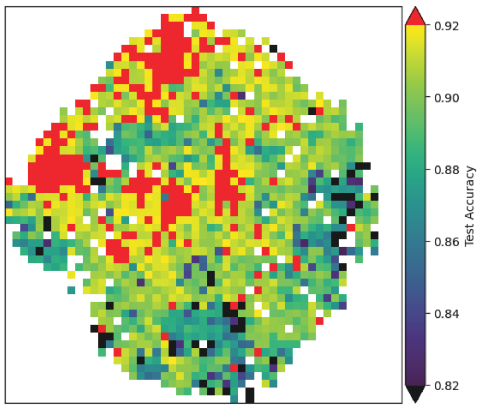

Figure 4.20: Arch2vec after 29 epochs, T-SNE on 10 000 architectures.

training, the second after 30 epochs. For every of the 4 possibilities, we ran the REINFORCE search 500 times. We use the same run settings as in the original arch2vec paper — 10^6 seconds limit per one run. Since NAS-Bench-101 is a tabular benchmark, there is no need to evaluate the networks in the search — instead, we query the benchmark for the validation and test accuracy of a network as well as its running time.

The results are summarized in Figures 4.21 and 4.22[1]. We use the metric called *regret*, defined as the difference between the maximal accuracy of the search space and the accuracy of the best performing network found during the search. The overall performance of the networks was similar, with only small differences at the end of the search. Arch2vec performed better in case of the validation regret, but the test regret was the same for both arch2vec runs and 30 epoch info-NAS.

Although info-NAS did not bring any improvement to the search process, the good side is that a network that learned two different tasks still maintained the same properties in the search process. Also, the NAS-Bench-101 and CIFAR-10 are relatively easy benchmark datasets compared to other existing search spaces or datasets (e.g. DARTS or ImageNet).

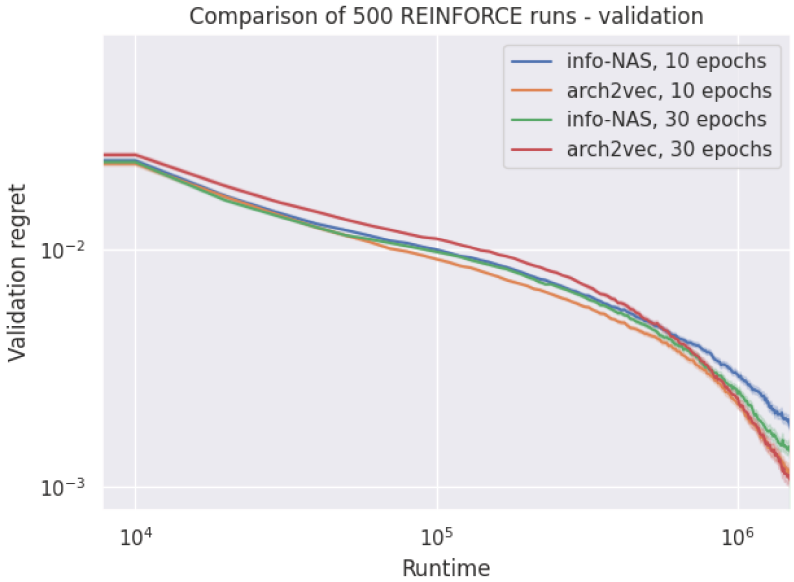

Figure 4.21: Comparison of REINFORCE runs — validation regret.

[1]The x axis was smoothed by separating timestamps into bins of size 10^4.

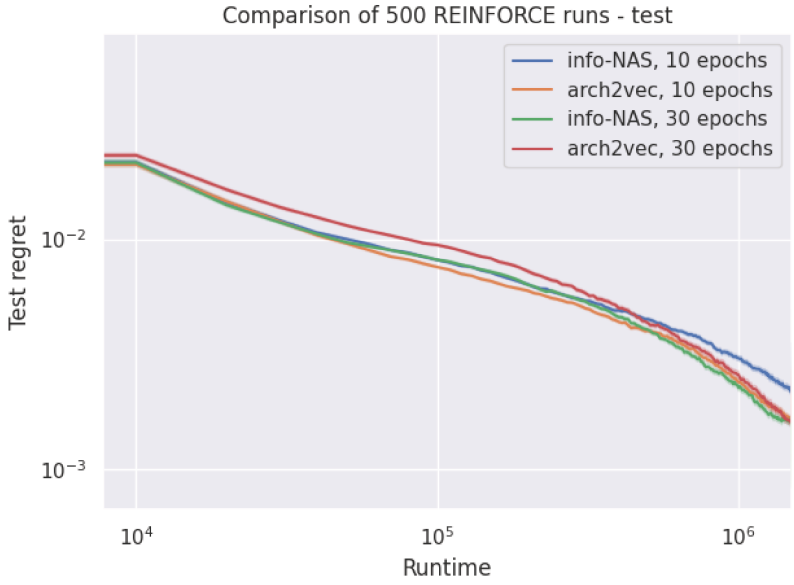

Figure 4.22: Comparison of REINFORCE runs — test regret.

4.4 Performance Estimation on NAS-Bench-101

In this experiment, we tried to reproduce the results of the original paper on perfomance prediction. The authors trained a gaussian process regressor on 250 randomly chosen latent features for 10 different seeds, and then predicted the performance of the other architectures (validation and test accuracy). The prediction was done only for networks with accuracy larger than 0.8. The authors reported two metrics, RMSE (0.018 ± 0.001) and Pearson's correlation coefficient r (0.67 ± 0.02), for the test accuracy prediction task.

For our experiment, we tried variable sample sizes ($[100, 250, 500, 1000]$), however sample sizes different than 250 had worse results in all cases (even for an arch2vec trained using the training function in the original repository). As so, we used sample size 250 for all subsequent experiments. We evaluated both arch2vec and info-NAS on 10 different seeds using 5 different regressors from the scikit-learn library [89] — gradient boosting, gaussian process, random forest, support vector regressor and a random forest with n_estimators = 1 000. Other than the second random forests, the algorithms were used with default hyperparameters. We report the results on validation accuracy prediction in Table 4.13 and on test accuracy prediction in Table 4.14 (values are mean accuracies with standard deviation).

Overall, info-NAS features combined with random forest performed the best

— for test accuracy of random forest, the respective 95 % confidence interval for the correlation coefficient is 0,00008 (using standard deviation 0.05 and sample size 423 374). The interval is approximately the same for both info-NAS and arch2vec, since the rounded standard deviation is the same. We were not able to reproduce the original results — a possible explanation is that the authors fine-tuned the original gaussian process to produce a good result.

Using the arch2vec, we create a plot for comparison of predicted and true accuracies (plot script is used from the original repository). We have chosen the best performin seed for both models. Figures 4.23 and 4.24 visualize the predicted test accuracies using the random forest. We see that in case of arch2vec, a lot of architectures have an underestimated architecture — however, it also predicts a lot of data points accurately.

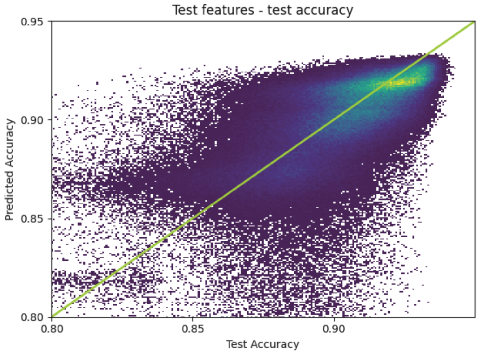

Figure 4.23: Predicted versus true accuracies using random forest — info-NAS.

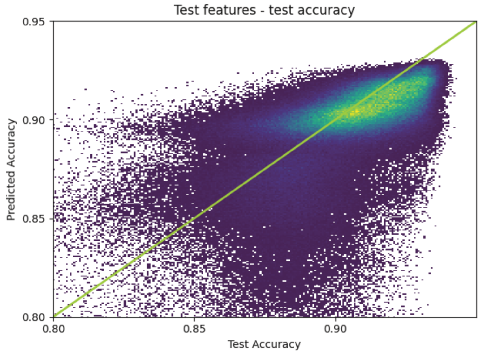

Figure 4.24: Predicted versus true accuracies using random forest — arch2vec.

Table 4.13: Performance prediction results — test set validation accuracy.

Algorithm	Model	RMSE	Pearson's r
Gradient boosting	info-NAS	0.03 ± 0.01	0.56 ± 0.05
	arch2vec	0.03 ± 0.01	0.52 ± 0.05
Gaussian process	info-NAS	0.07 ± 0.02	0.30 ± 0.09
	arch2vec	0.06 ± 0.02	0.29 ± 0.09
Random forest	**info-NAS**	0.03 ± 0.01	$\mathbf{0.61 \pm 0.05}$
	arch2vec	$\mathbf{0.02 \pm 0.01}$	0.58 ± 0.06
SVR	info-NAS	0.06 ± 0.01	0.36 ± 0.13
	arch2vec	0.06 ± 0.01	0.40 ± 0.05

Table 4.14: Performance prediction results — test set test accuracy.

Algorithm	Model	RMSE	Pearson's r
Gradient boosting	info-NAS	0.03 ± 0.01	0.55 ± 0.06
	arch2vec	0.03 ± 0.01	0.52 ± 0.05
Gaussian process	info-NAS	0.06 ± 0.02	0.29 ± 0.09
	arch2vec	0.06 ± 0.02	0.29 ± 0.09
Random forest	**info-NAS**	$\mathbf{0.02 \pm 0.01}$	$\mathbf{0.61 \pm 0.05}$
	arch2vec	0.02 ± 0.00	0.58 ± 0.05
SVR	info-NAS	0.06 ± 0.01	0.36 ± 0.13
	arch2vec	0.05 ± 0.01	0.40 ± 0.05

Conclusion

In this work, we have designed a novel model for neural architecture embedding — info-NAS — a model, that learns the outputs of neural networks given input data. We created the input-output dataset using networks from the NAS-Bench-101 search space pretrained on the image dataset CIFAR-10, and we extended the original variational graph autoencoder — we jointly trained it with a semi-supervised regressor on unlabeled data, that contained only neural architecture data, and labeled data — neural architectures combined with the input-output dataset.

We analyzed the behavior of our model by running it with different hyperparameter settings, and we found that one data scaling approach leads to better results than the other ones. We analyzed the predicted output features and confirmed that it is possible for the model to learn from this type of data, but also identified some problems that need to be solved in subsequent work. We analyzed the results on validation and test datasets created from unseen architectures and images, and the results were better than the baseline except for one test set, where the results were close to the baseline. We compared info-NAS training to arch2vec and concluded that the joint training does not significantly reduce the performance on original metrics like reconstruction accuracy. Lastly, we compared the latent spaces of both models and noticed some interesting changes of the latent space structure of info-NAS.

Finally, we evaluated both info-NAS and the reference arch2vec model by running the REINFORCE search on NAS-Bench-101, and by training performance predictors on the extracted latent features. In the search experiment, both models performed the same — although info-NAS did not bring any improvement, the good result is that the semi-supervised training did not damage the performance of the original model. On the task of performance prediction, we were not able to reproduce the results in the original paper, as the regressor they used (gaussian process) had poor performance for both models. The authors did not provide the specific hyperparameters of the regressor, and as so, they may have used a specific fine-tuned model. In our experiments, the best performing model was random forest, and info-NAS outperformed the reference

arch2vec.

The main task of future work on info-NAS is to improve the training process. On some networks from the train test, info-NAS did not perform well, similarly for the validation and test set. This issue could be solved by developing a more advanced model, increasing the train set size, or by using different output features (labels) or metrics. Furthermore, some experiments indicated possible overfitting on the overrepresented labeled network architectures. Since loss weighting — a possible solution of this problem — lead to worse results, some other strategies need to be employed. A simple solution is to repeat the unlabeled dataset, this however increases the overall train size and slows down the training process. A better possibility is to explore some batch sampling strategies. Lastly, since neither of the used models matched the performance prediction results, more experiments should be done. For example, a larger dimension of the latent space could improve the performance of performance estimators.

The task of predicting the outputs of neural networks is a novel, unexplored approach, and there are many possible directions of research one could take. The main outcome of our work is that we have shown that it is possible to train a model on this kind of data, and that it could possibly lead to improvement in some tasks like performance estimation.

Bibliography

[1] Thomas M. Mitchell. *Machine Learning*. 1st ed. USA: McGraw-Hill, Inc., 1997. ISBN: 0070428077.

[2] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012. ISBN: 0262018020.

[3] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[5] Tomáš Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2013. URL: http://arxiv.org/abs/1301.3781.

[6] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[7] Olivier Chapelle, Bernhard Schlkopf, and Alexander Zien. *Semi-Supervised Learning*. 1st. The MIT Press, 2010. ISBN: 0262514125.

[8] Xin He, Kaiyong Zhao, and Xiaowen Chu. "AutoML: A survey of the state-of-the-art". In: *Knowledge-Based Systems* 212 (2021), p. 106622. ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2020.106622. URL: https://www.sciencedirect.com/science/article/pii/S0950705120307516.

[9] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep Learning". In: *Nature* 521.7553 (2015), pp. 436–444. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539.

[10] Raúl Rojas. *Neural Networks: A Systematic Introduction*. Berlin, Heidelberg: Springer-Verlag, 1996. ISBN: 3540605053.

[11] Francois Chollet. *Deep Learning with Python*. 1st. USA: Manning Publications Co., 2017. ISBN: 1617294438.

[12] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. cite arxiv:1609.04747Comment: Added derivations of AdaMax and Nadam. 2016. URL: http://arxiv.org/abs/1609.04747.

[13] Ning Qian. "On the momentum term in gradient descent learning algorithms". In: *Neural Networks* 12.1 (1999), pp. 145–151. ISSN: 0893-6080. DOI: https://doi.org/10.1016/S0893-6080(98)00116-6. URL: https://www.sciencedirect.com/science/article/pii/S0893608098001166.

[14] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization". In: *J. Mach. Learn. Res.* 12.null (July 2011), 2121–2159. ISSN: 1532-4435.

[15] Tijmen Tieleman, Geoffrey Hinton, et al. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural networks for machine learning* 4.2 (2012), pp. 26–31.

[16] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[17] *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*. Berlin, Heidelberg: Springer-Verlag, 1998. ISBN: 3540653112.

[18] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 2010, pp. 249–256. URL: http://proceedings.mlr.press/v9/glorot10a.html.

[19] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *J. Mach. Learn. Res.* 15.1 (2014), 1929–1958. ISSN: 1532-4435.

[20] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1 (1989), pp. 541–551.

[21] Yoon Kim. "Convolutional Neural Networks for Sentence Classification". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1746–1751. DOI: 10.3115/v1/D14-1181. URL: https://aclanthology.org/D14-1181.

[22] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[23] Yann LeCun, Yoshua Bengio, et al. "Convolutional networks for images, speech, and time series". In: *The handbook of brain theory and neural networks* 3361.10 (1995), p. 1995.

[24] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. USA: California Technical Publishing, 1997. ISBN: 0966017633.

[25] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791.

[26] Min Lin, Qiang Chen, and Shuicheng Yan. *Network In Network*. 2014. arXiv: 1312.4400 [cs.NE].

[27] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, 2015, pp. 448–456. URL: http://proceedings.mlr.press/v37/ioffe15.html.

[28] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'12. Lake Tahoe, Nevada: Curran Associates Inc., 2012, 1097–1105.

[29] Jost Tobias Springenberg et al. *Striving for Simplicity: The All Convolutional Net*. 2015. arXiv: 1412.6806 [cs.LG].

[30] Diederik P. Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. 2014. arXiv: http://arxiv.org/abs/1312.6114v10 [stat.ML].

[31] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. "Stochastic Backpropagation and Approximate Inference in Deep Generative Models". In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 2. Bejing, China: PMLR, 2014, pp. 1278–1286. URL: http://proceedings.mlr.press/v32/rezende14.html.

[32] Durk P Kingma et al. "Semi-supervised Learning with Deep Generative Models". In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani et al. Vol. 27. Curran Associates, Inc., 2014.

[33] Jonathan L. Gross, Jay Yellen, and Ping Zhang. *Handbook of Graph Theory, Second Edition*. 2nd. Chapman & Hall/CRC, 2013. ISBN: 1439880182.

[34] Kimberly Jordan Burch. "Chapter 8 - Chemical applications of graph theory". In: *Mathematical Physics in Theoretical Chemistry*. Ed. by S.M. Blinder and J.E. House. Developments in Physical & Theoretical Chemistry. Elsevier, 2019, pp. 261–294. ISBN: 978-0-12-813651-5. DOI: https://doi.org/10.1016/B978-0-12-813651-5.00008-5. URL: https://www.sciencedirect.com/science/article/pii/B9780128136515000085.

[35] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. "A linear-time algorithm for testing the truth of certain quantified boolean formulas". In: *Information Processing Letters* 8.3 (1979), pp. 121–123. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(79)90002-4. URL: https://www.sciencedirect.com/science/article/pii/0020019079900024.

[36] Zonghan Wu et al. "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (2021), 4–24. ISSN: 2162-2388. DOI: 10.1109/tnnls.2020.2978386. URL: http://dx.doi.org/10.1109/TNNLS.2020.2978386.

[37] David I Shuman et al. "The emerging field of signal processing on graphs: Extending high-dimensional data analysis to networks and other irregular domains". In: *IEEE Signal Processing Magazine* 30.3 (2013), pp. 83–98. DOI: 10.1109/MSP.2012.2235192.

[38] Thomas N. Kipf and Max Welling. "Semi-Supervised Classification with Graph Convolutional Networks". In: *Proceedings of the 5th International Conference on Learning Representations* (ICLR). ICLR '17. Palais des Congrès Neptune, Toulon, France, 2017. URL: https://openreview.net/forum?id=SJU4ayYgl.

[39] Justin Gilmer et al. "Neural Message Passing for Quantum Chemistry". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia: JMLR.org, 2017, 1263–1272.

[40] Keyulu Xu et al. “How Powerful are Graph Neural Networks?” In: *International Conference on Learning Representations*. 2019. URL: https://openreview.net/forum?id=ryGs6iA5Km.

[41] Thomas Kipf and M. Welling. “Variational Graph Auto-Encoders”. In: *ArXiv* abs/1611.07308 (2016).

[42] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning: From Theory to Algorithms*. USA: Cambridge University Press, 2014. ISBN: 1107057132.

[43] Marc-André Zöller and Marco F. Huber. “Benchmark and Survey of Automated Machine Learning Frameworks”. In: *J. Artif. Int. Res.* 70 (May 2021), 409–472. ISSN: 1076-9757. DOI: 10.1613/jair.1.11854. URL: https://doi.org/10.1613/jair.1.11854.

[44] Chris Thornton et al. “Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms”. In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’13. Chicago, Illinois, USA: Association for Computing Machinery, 2013, 847–855. ISBN: 9781450321747. DOI: 10.1145/2487575.2487629. URL: https://doi.org/10.1145/2487575.2487629.

[45] Matthias Feurer et al. “Efficient and Robust Automated Machine Learning”. In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., 2015, pp. 2962–2970. URL: http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf.

[46] Randal S. Olson et al. “Applications of Evolutionary Computation: 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 – April 1, 2016, Proceedings, Part I”. In: ed. by Giovanni Squillero and Paolo Burelli. Springer International Publishing, 2016. Chap. Automating Biomedical Data Science Through Tree-Based Pipeline Optimization, pp. 123–137. ISBN: 978-3-319-31204-0. DOI: 10.1007/978-3-319-31204-0_9. URL: http://dx.doi.org/10.1007/978-3-319-31204-0_9.

[47] Aaron Klein and Frank Hutter. *Tabular Benchmarks for Joint Architecture and Hyperparameter Optimization*. 2019. arXiv: 1905.04970 [cs.LG].

[48] Xiaoliang Dai et al. “FBNetV3: Joint Architecture-Recipe Search Using Predictor Pretraining”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2021, pp. 16276–16285.

[49] Difan Deng and Marius Lindauer. *Literature on Neural Architecture Search*. 2021. URL: https://www.automl.org/automl/literature-on-neural-architecture-search/ (visited on 07/14/2021).

[50] Mingxing Tan and Quoc Le. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 6105–6114. URL: http://proceedings.mlr.press/v97/tan19a.html.

[51] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[52] *Image Classification on ImageNet Leaderboard*. 2021. URL: https://paperswithcode.com/sota/image-classification-on-imagenet (visited on 07/14/2021).

[53] Hieu Pham et al. "Meta Pseudo Labels". In: *IEEE Conference on Computer Vision and Pattern Recognition*. 2021. URL: https://arxiv.org/abs/2003.10580.

[54] T. Elsken, J. H. Metzen, and F. Hutter. "Neural Architecture Search: A Survey". In: *ArXiv* abs/1808.05377 (2019).

[55] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. "A Survey on Neural Architecture Search". In: *ArXiv* abs/1905.01392 (2019).

[56] George Kyriakides and Konstantinos G. Margaritis. "Evolving graph convolutional networks for neural architecture search". In: *Neural Computing and Applications* (Apr. 2021), pp. 1–11. DOI: 10.1007/s00521-021-05979-8.

[57] Barret Zoph and Quoc V. Le. "Neural Architecture Search with Reinforcement Learning". In: 2017. URL: https://arxiv.org/abs/1611.01578.

[58] Barret Zoph et al. "Learning Transferable Architectures for Scalable Image Recognition". In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018), pp. 8697–8710.

[59] Chris Ying et al. "NAS-Bench-101: Towards Reproducible Neural Architecture Search". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. Long Beach, California, USA: PMLR, 2019, pp. 7105–7114. URL: http://proceedings.mlr.press/v97/ying19a.html.

[60] Hanxiao Liu, Karen Simonyan, and Yiming Yang. *DARTS: Differentiable Architecture Search*. 2019. arXiv: 1806.09055 [cs.LG].

[61] Xuanyi Dong and Yi Yang. "NAS-Bench-201: Extending the Scope of Reproducible Neural Architecture Search". In: *International Conference on Learning Representations (ICLR)*. 2020. URL: https://openreview.net/forum?id=HJxyZkBKDr.

[62] Frédéric Gruau. *Neural Network Synthesis Using Cellular Encoding And The Genetic Algorithm*. 1994.

[63] Kenneth O. Stanley and Risto Miikkulainen. "Evolving Neural Networks Through Augmenting Topologies". In: *Evolutionary Computation* 10.2 (2002), pp. 99–127. URL: http://nn.cs.utexas.edu/?stanley:ec02.

[64] Ronald J. Williams. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Mach. Learn.* 8.3–4 (May 1992), 229–256. ISSN: 0885-6125. DOI: 10.1007/BF00992696. URL: https://doi.org/10.1007/BF00992696.

[65] Tao Wei et al. "Network Morphism". In: *CoRR* abs/1603.01670 (2016). arXiv: 1603.01670. URL: http://arxiv.org/abs/1603.01670.

[66] "Surrogate-based analysis and optimization". In: *Progress in Aerospace Sciences* 41.1 (2005), pp. 1–28. ISSN: 0376-0421. DOI: https://doi.org/10.1016/j.paerosci.2005.02.001. URL: https://www.sciencedirect.com/science/article/pii/S0376042105000102.

[67] Christoph Molnar. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable*. https://christophm.github.io/interpretable-ml-book/. 2019.

[68] Carl Edward Rasmussen. "Gaussian Processes in Machine Learning". In: *Advanced Lectures on Machine Learning: ML Summer Schools 2003, Canberra, Australia, February 2 - 14, 2003, Tübingen, Germany, August 4 - 16, 2003, Revised Lectures*. Ed. by Olivier Bousquet, Ulrike von Luxburg, and Gunnar Rätsch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 63–71. ISBN: 978-3-540-28650-9. DOI: 10.1007/978-3-540-28650-9_4. URL: https://doi.org/10.1007/978-3-540-28650-9_4.

[69] Yanan Sun et al. "Surrogate-Assisted Evolutionary Deep Learning Using an End-to-End Random Forest-Based Performance Predictor". In: *IEEE Transactions on Evolutionary Computation* 24.2 (2020), pp. 350–364. DOI: 10.1109/TEVC.2019.2924461.

[70] Shen Yan et al. "Does Unsupervised Architecture Representation Learning Help Neural Architecture Search?" In: *NeurIPS*. 2020.

[71] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: https://doi.org/10.1162/neco.1997.9.8.1735.

[72] Chenxi Liu et al. "Progressive Neural Architecture Search". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018.

[73] Renqian Luo et al. "Neural Architecture Optimization". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, 7827–7838.

[74] Renqian Luo et al. "Semi-Supervised Neural Architecture Search". In: *ArXiv* abs/2002.10389 (2020).

[75] Colin White, Willie Neiswanger, and Yash Savani. *{BANANAS}: Bayesian Optimization with Neural Networks for Neural Architecture Search*. 2020. URL: https://openreview.net/forum?id=B1lxV6NFPH.

[76] George Kyriakides and Konstantinos G. Margaritis. "Evolving graph convolutional networks for neural architecture search". In: *Neural Computing and Applications* (Apr. 2021), pp. 1–11. DOI: 10.1007/s00521-021-05979-8.

[77] Han Shi et al. "Bridging the Gap between Sample-based and One-shot Neural Architecture Search with BONAS". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 1808–1819. URL: https://proceedings.neurips.cc/paper/2020/file/13d4635deccc230c944e4ff6e03404b5-Paper.pdf.

[78] Xuefei Ning et al. "A Generic Graph-Based Neural Architecture Encoding Scheme for Predictor-Based NAS". In: *Computer Vision – ECCV 2020*. Ed. by Andrea Vedaldi et al. Cham: Springer International Publishing, 2020, pp. 189–204. ISBN: 978-3-030-58601-0.

[79] Muhan Zhang et al. "D-VAE: A Variational Autoencoder for Directed Acyclic Graphs". In: (2019), pp. 1586–1598.

[80] Stefan Falkner, Aaron Klein, and Frank Hutter. "BOHB: Robust and Efficient Hyperparameter Optimization at Scale". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, 2018, pp. 1437–1446. URL: http://proceedings.mlr.press/v80/falkner18a.html.

[81] James Bergstra and Yoshua Bengio. "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13.10 (2012), pp. 281–305. URL: http://jmlr.org/papers/v13/bergstra12a.html.

[82] Esteban Real et al. “Regularized Evolution for Image Classifier Architecture Search”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 33 (Feb. 2018). DOI: 10.1609/aaai.v33i01.33014780.

[83] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: *NIPS-W*. 2017.

[84] Alex Krizhevsky. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.

[85] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[86] Romulus Hong. *NASBench-PyTorch*. https://github.com/romulus0914/NASBench-PyTorch. 2013.

[87] Ilya Loshchilov and Frank Hutter. “SGDR: Stochastic Gradient Descent with Restarts”. In: *CoRR* abs/1608.03983 (2016). arXiv: 1608.03983. URL: http://arxiv.org/abs/1608.03983.

[88] Laurens van der Maaten and Geoffrey Hinton. “Visualizing Data using t-SNE”. In: *Journal of Machine Learning Research* 9 (2008), pp. 2579–2605. URL: http://www.jmlr.org/papers/v9/vandermaaten08a.html.

[89] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

Appendix A

Using our implementation

The code for this work is divided into four repositories: info-nas, arch2vec, NASBench-PyTorch and nasbench. The first is the code of our implementation, the rest are the supporting libraries:

- arch2vec — the original arch2vec repository with some changes to enable info-NAS easily extend the model and reuse some code (like evaluation)
- NASBench-PyTorch — implementation of NAS-Bench-101 training in PyTorch
- nasbench — the original nasbench repository modified to work with TensorFlow >= 2.0

All three repositories were modified into a local package installable by pip. Although all repositories are included in the thesis attachments, we recommend cloning the info-NAS repository from `https://github.com/gabrielasuchopar/info-nas/`. The installation of the virtual environment and of the other libraries is described in info-nas repository README.md. The url to some of the generated data (datasets and scales) is also stored in the README.