Task-aware Performance Prediction for Efficient Architecture Search

Efi Kokiopoulou and **Anja Hauth** and **Luciano Sbaiz** and **Andrea Gesmundo** and **Gábor Bartók** and **Jesse Berent** [1]

Abstract. Neural architecture search has been shown to hold great promise towards the automation of deep learning. However, in spite of its potential, neural architecture search remains quite costly. To this point, we propose a novel gradient-based framework for efficient architecture search by sharing information across several tasks. We start by training many model architectures on several related (training) tasks. When a new unseen task is presented, the framework performs architecture inference in order to quickly identify a good candidate architecture, *before* any model is trained on the new task. At the core of our framework lies a performance prediction network that can estimate the performance of input architectures on a task by utilizing task meta-features and the previous model training experiments performed on related tasks. We adopt a continuous parametrization of the model architecture, which allows for efficient gradient-based optimization. Given a new task, an effective architecture is quickly identified by maximizing the estimated performance with respect to the model architecture parameters with efficient gradient ascent. It is key to point out that our goal is to achieve reasonable performance at the lowest cost. We provide experimental results showing the effectiveness of the framework achieving directly comparable results with state-of-the-art methods albeit at a much more reduced computational cost.

1 INTRODUCTION

Designing high performing neural networks is a time consuming task that typically requires substantial human effort. In the past few years, neural architecture search and algorithmic solutions to model building have received growing research interest as they can automate the manual process of model design. Although they offer impressive results that compete with human-designed models Zoph and Le [2017], neural architecture search requires large amount of computational resources for each new task. For this reason, recent methods have been proposed that focus on reducing its cost (see e.g., Liu et al. [2019], Pham et al. [2018], Bender et al. [2018], Zhang et al. [2019]). This very fact becomes a major limitation in those setups that impose strict resource constraints for model design. For example, in cloud machine learning services, the client uploads a new data set and an effective model should ideally be auto-designed in minutes (or seconds). In such settings architecture search has to be very efficient, which is the main motivation for this work.

At the same time, applying independently automated model building methods to each new task requires a lot of models to be trained as well as learning how to generate high performing models from

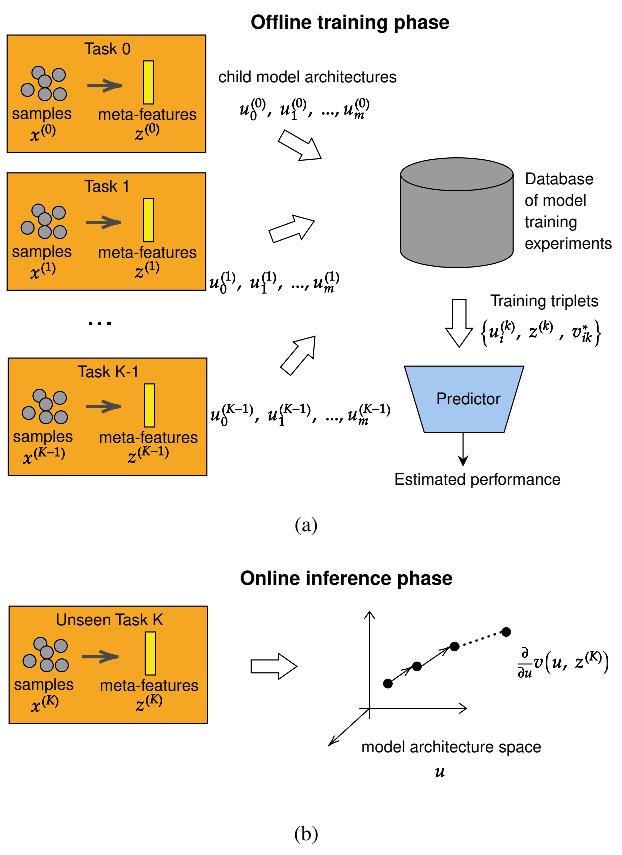

Figure 1. The proposed framework. (a) *Offline phase*. Several architectures are trained and their performances are stored in a database. The performances along with meta-features about the task are used to train a network which estimates the performance. (b) *Online phase*. Given a new task and its meta-features, the system applies gradient ascent on the output of the prediction network.

scratch. Such an approach requires a formidable amount of computational resources and is far from being scalable. On the other hand, human experts can design state-of-the-art models using prior knowledge about how existing architectures perform across different data sets. Similar to human experts, we aim to cross learn from several task data sets and leverage prior knowledge.

In this work, we present a framework that amortizes the cost of architecture search across several tasks and remains effective thanks to the knowledge transfer between tasks. We propose to learn a performance prediction network that estimates the performance of a

[1] Google AI. Emails: {efi, ahauth, sbaiz}@google.com and {agesmundo, bartok, jberent}@google.com

candidate architecture on a certain task. Given a candidate model architecture and meta-features about the task, the prediction network provides an estimate of the performance of the input architecture on the task data set via a *differentiable* mapping. At the same time we adopt a continuous parametrization of the model architecture, which allows for efficient gradient-based optimization of the estimated performance. Adding to the state-of-the-art, we learn crucially important task data set features directly from the task samples while training the performance prediction network.

The framework consists of an *offline training phase* and *an online inference phase* (see Fig. 1 for a conceptual illustration). In the offline phase several architectures are trained on several task data sets and their performances are stored in a database. Adding to the state-of-the-art however, the offline phase is designed to be scalable, *i.e.*, when new data sets are added, the offline training involves only these. Assuming that we have trained several model architectures on several related training tasks, when a new unseen task is presented in the online phase, the framework performs fast architecture optimization in order to quickly identify a good candidate architecture, *before* any actual model training is performed. For the online phase we rely on a combination of i) a very cheap performance prediction of a candidate architecture instead of actual model training and ii) continuous relaxation and thus gradient-based architecture optimization. Note that the user of our framework only sees the cost of the online phase, which is very fast thanks to the gradient ascent.

In summary, the paper contributions are the following:

- Efficient single-shot architecture search on a new task using gradient-based architecture optimization. Different from existing gradient-based methods Liu et al. [2019], Shin et al. [2018], Cai et al. [2019], Xie et al. [2019], our method optimizes *directly* for the architecture parameters *without* any intermediate model training or intermediate model weight updates.
- Ability to learn the task meta-features *directly* from the raw task data samples together with the performance prediction network weights. This is different from previous work that uses pre-computed task meta-features Feurer et al. [2015a] or implicitly learns task embeddings from their corresponding task ids Wong et al. [2018], Fusi et al. [2018].
- Task-aware performance predictor that predicts the performance of a candidate architecture on a certain task. Unlike previous performance predictors Liu et al. [2018], Istrate et al. [2019], Deng et al. [2017], our predictor takes into account not only the candidate architecture, but also meta-features about the task derived directly from the task data samples themselves.

We provide experimental results illustrating that our methodology achieves results that are quite close to expensive online methods such as Reinforcement Learning Zoph and Le [2017]. In addition, our comparison with state-of-the-art efficient methods based on transfer learning between tasks Wong et al. [2018] demonstrates that our results compare favorably with these while at a significantly lower cost.

2 PROBLEM FORMULATION

We are interested in task-aware *efficient* neural architecture search. Given a new (unseen) task data set, we would like to identify quickly an effective model architecture *before* any model is trained. We want to learn across datasets in order to amortize the cost of neural architecture search. In particular, we want to *collectively* learn from all the model training experiments and leverage this wealth of information. Instead

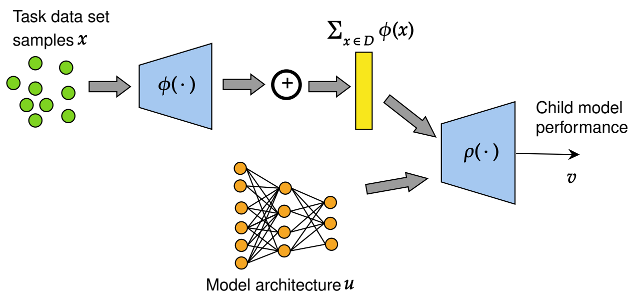

Figure 2. The architecture of the performance predictor, which consists of $\phi(\cdot)$ and $\rho(\cdot)$.

of performing architecture search independently for each new data set, we would like to transfer the knowledge obtained from past training experiments on related tasks. In summary, the proposed framework should have the following properties:

- High scalability in terms of computing resources.
- Ability to scale across task data sets and learn collectively from them.
- Ability to propose a good architecture for a new related task without training any model.

In the next section we propose a general framework that has these desired properties.

3 PROPOSED FRAMEWORK

We want to automatically discover the model architecture that achieves the best quality for a given data set. Essentially we are looking for learning a mapping from an input data set to a high performing model architecture. We propose to formalize the architecture search problem as a *structured output prediction* problem Gygli et al. [2017]. The key intuition is that learning to criticize candidate architectures is easier than learning to directly predict the optimal architecture. In particular, the proposed framework lies on an auxiliary model that predicts the performance of a candidate architecture on a certain task. In our context, the performance predictor acts as a meta-model that helps in tuning the architecture of a child model. We use small neural networks as performance predictors because they readily provide a differentiability mapping from their inputs to output and also because they are able to learn complex feature interactions (that typically cannot be matched by other approaches). We consider child model families parametrized by u, assuming for now that u is a vector of continuous variables.

In its simplest form, a performance prediction network takes as input: (i) descriptive meta-features z derived from a certain task data set and (ii) the child model architecture parameters u, and predicts how well the architecture u performs on the task data set described by z. The network is shown conceptually in Fig. 2. The performance metric v can take various forms (e.g., accuracy, AUC) but the framework is agnostic to it. In this work, we use the validation accuracy as performance metric.

When training the prediction network, our hope is that it learns which type of child model architectures work well on certain types of data. This tries to mimic the human expert during manual architecture design. Human experts rely on intuition and prior knowledge when developing new candidate architectures. Here, our hope is that such an 'intuition' is encoded in the weights of the performance prediction network and that it is generally applicable and transferable across data sets. In the following sections, we provide more details about the proposed framework. We discuss the meta-features of a task in Section 3.1. The framework has two phases: an *offline training* phase and an *online inference* phase that are detailed below in Sections 3.2 and 3.3 respectively. Section 3.4 discusses the child model architecture parameters u.

3.1 The meta-features of a task

The meta-features z of a task describe its characteristics and they are typically derived from the task data set itself Vanschoren [2018]. The meta-features may include things such as: number of samples, number of classes and their distribution, label entropy and so on. Although such pre-computed task meta-features could be combined with the proposed framework, they are typically rather expensive to compute for each new task. Hence, they are out-of-scope, since our framework has very tight run-time requirements. For this reason, we do not consider them further in this work and we instead introduce below a method to learn the meta-features directly from the task data samples. We learn the meta-features jointly with the weights of the performance prediction network.

In order to learn the meta-features directly from the task data set D, the data set (or a large fraction of it) is given as input to the performance predictor, and a task embedding is learned directly from the raw samples of the task. Note that we use *both* the features and the labels of the task data set samples towards learning the task embedding. This task embedding plays the role of the meta-features and is learned jointly together with the rest of the weights of the performance prediction network.

The task embedding should be invariant to the order of the samples in the task data set. According to Zaheer et al. [2017], such a function can be decomposed in the form $\rho(\sum_{x\in D}\phi(x))$ for suitable transformations ϕ and ρ. The latter transformations are typically implemented by a few layers (e.g., fully connected, non-linearities etc.). The main idea is to transform each sample from the task data set using $\phi(\cdot)$ and then aggregate the transformed samples such that the task embedding becomes permutation invariant before it is fed into $\rho(\cdot)$. This process is shown conceptually in Fig. 2, where the performance prediction network essentially consists of $\phi(\cdot)$ and $\rho(\cdot)$ that are jointly learned, *i.e.*,

$$v(u,z) := \rho\left(u, \sum_{x\in D}\phi(x)\right), \tag{1}$$

where $z = \sum_{x\in D}\phi(x)$. We assume here that the data samples of different tasks are expressed in a common feature space that can be ingested by $\phi(\cdot)$.

3.2 Offline training phase

Assume we have K tasks with corresponding data sets denoted as:

$$D_k = \left\{(x_i^{(k)}, y_i^{(k)})\right\}_{i=0}^{N_k-1},\ k = 0,\dots,K-1, \tag{2}$$

where N_k is the number of data samples in the k-th task. $(x_i^{(k)}, y_i^{(k)})$ is the i-th sample and its corresponding label in the k-th task data set.

For each task data set, we generate m child model architectures, train them and collect the model performances on the validation set in a database of model training experiments; see Fig. 1(a). This database

Algorithm 1 Offline training phase

```
Inputs:
  Task datasets $D_k$, see Eq. (2)
  Prediction network training set $T$, see Eq. (3)
  kInnerIters, kOuterIters
repeat
  Sample randomly a task $k$.
  for $i = 1$ to kOuterIters do
    Pick a mini-batch from $T$ with samples only from task $k$.
    for $j = 1$ to kInnerIters do
      Pick a large batch from the task dataset $D_k$.
      Perform one step of Stochastic Gradient Descent on the
      weights of the prediction network.
    end for
  end for
until Convergence
```

is used to generate the training set for the performance prediction network, which consists of M triplets of the form:

$$T = \{(z_i, u_i, v_i^*)\}_{i=0}^{M-1}, \tag{3}$$

where the value v_i^* holds the child model performance obtained when training with the model architecture u_i on the task data set with meta-features z_i. In this work, the model performance metric used is the validation accuracy. As more tasks are ingested in the database and more models get trained, the network improves its predictions, as we have also verified experimentally. Once the child model training experiments have been collected in the database, we can start training the performance prediction network. Algorithm 1 shows the main steps of this offline training phase.

3.3 Online inference phase

After training the performance prediction network $v(u, z; w)$, the model weights w are kept fixed. At inference time, given a new task dataset, we first extract its meta-features z. At this point we can employ the network in two ways. First, if we have a candidate architecture u we can evaluate it by simply doing a forward pass on the network and get the estimated child model performance. Alternatively, we can compute the gradient of $v(u, z)$ with respect to u and perform efficient gradient-based optimization to get a good candidate architecture $\hat{u}$ that maximizes the estimated child model performance.

In practice, we noticed that the gradient ascent is sensitive to initialization. Hence, we run the process several times with different initial guesses and at the end pick the one that resulted in the maximum estimated performance. In order to pick the initial guesses we find the closest training tasks in the task embedding space and we collect the top architectures found in the database for these tasks.

Note also that in order to be able to perform gradient-based inference we need to relax the model architecture parameters u to live in a continuous space. Section 3.4 below discusses this parametrization in details. The main steps of the online phase are shown in Algorithm 2. This online process is also illustrated conceptually in Fig. 1(b). Note in passing that the user of the proposed framework only sees the cost of the online phase, which is that of gradient ascent, which is very efficient (please see supporting experiments in Section 5 below).

3.4 Architecture parametrization

We discuss in this section the parametrization of the child model architectures. Previous work Liu et al. [2019], Shin et al. [2018], Cai et al.

Algorithm 2 Online inference phase

```
Inputs:
  New task dataset $D_K = \{(x_i^{(K)}, y_i^{(K)})\}_{i=0}^{N_K-1}$
  Trained performance prediction network $v(u, z; w)$
  kNumStartingPoints, kMaxIters
Compute the meta-features $z = \sum_{x \in D_K} \phi(x)$
Form an empty set $S = \{\}$ of solutions
for $i = 1$ to kNumStartingPoints do
    Pick an initial guess $u_i^{(0)}$, $t = 0$
    repeat
        $u_i^{(t+1)} = u_i^{(t)} + \eta \frac{\partial}{\partial u} v(u_i^{(t)}, z; w)$
        $t := t + 1$
    until Convergence (or $t >$ kMaxIters)
    $S := S \cup \{(\hat{v}_i, \hat{u}_i)\}$ where $\hat{u}_i$ is the found solution and $\hat{v}_i$ its
    corresponding value.
end for
Output: $\arg\max_{(v,u) \in S} v(u)$.
```

[2019], Xie et al. [2019] has shown that relaxing the parametrization from discrete to continuous space allows for efficient gradient-based optimization schemes while still providing competitive model performances. Our approach goes along the lines of this previous work. The main idea is that in order to make the architecture space continuous we move away from the categorical nature of design choices to a parametrized softmax over all possible choices. We provide below a few examples where this is applied.

Continuous parametrization for one layer Assume that we have implemented a basis set consisting of p base layers $o_i(x)$ corresponding to different sizes and different activation functions. We associate a weight α_i with each base layer and we define a new parametrized layer as follows

$$o(x) = \sum_{i=1}^{p} \frac{\exp(\alpha_i)}{\sum_{j=1}^{p} \exp(\alpha_j)} o_i(x). \tag{4}$$

The values α_i allow the final parametrized layer $o(x)$ to 'morph' from one size to another and/or from one activation function to another. We use zero padding whenever needed to resolve the dimension mismatch among different layer sizes.

Continuous parametrization for a child network Leveraging on the continuous parametrization for one layer introduced above, we can put several parametrized layers together. We attach a superscript to the layer parameters to denote the layer where they belong to *i.e.*, $\alpha_i^{(j)}$ is the parameter that multiplies the output of the i-th base layer in the j-th parametrized layer of the final network.

We also add the ability for each layer to be enabled or disabled independently from the other layers. For this, we add extra parameters β_j that control the presence or absence of each layer. This is shown conceptually in Fig. 3.

Putting everything together, we consider child models that are standard Feedforward Neural Networks (FFNNs) composed of an embedding module followed by several parametrized layers and a final softmax classification layer. The reason for using an embedding module is that it speeds up the training time for the child models and improves their quality especially when the training set is small. The embedding module is soft-selected by an input set of pre-trained embedding modules[2] using the same softmax trick analogous to Eq. (4)

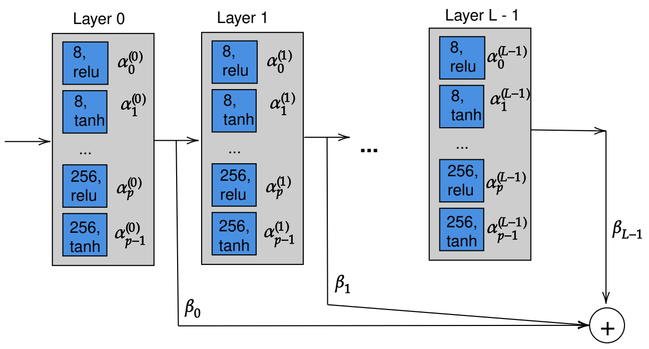

Figure 3. Continuous parametrization of the child models. Each layer is a parametrized combination of base layers. Each layer can also be enabled or disabled independently of the other layers.

where we denote by γ the corresponding parameters of the softmax. After this relaxation, architecture search reduces to learning the continuous variables $u := \{\{\alpha\}, \{\beta\}, \{\gamma\}\}$. We refer to u as the encoding of the model architecture. Finally, we would like to emphasize that this parametrization is just one example among many possible options. Any parametrization should work with the proposed framework as long as it is continuous.

4 RELATED WORK

Automated model building is an important challenging research problem and several related methods have been proposed in the past few years. In general, previous works can be broadly categorized into the following classes:

- Bayesian optimization methods Snoek et al. [2012], Mendoza et al. [2016], Hutter et al. [2011], Fusi et al. [2018], Feurer et al. [2015b] build a probabilistic model of the performance of the network as a function of its hyperparameters and then decide which candidate point in the search space to evaluate next.
- Methods based on Reinforcement Learning (RL) Zoph and Le [2017] evaluate candidate child model architectures on-the-fly, by training and evaluating on a validation set. Using the validation accuracy as the reward signal, these methods use RL to optimize the child model architecture. Recent methods have been focusing on cost reduction using transfer learning Wong et al. [2018] or by introducing weight sharing among candidate architectures (see e.g., Pham et al. [2018], Bender et al. [2018]) or by combining RL together with performance prediction Liu et al. [2018].
- Evolutionary methods such as Real et al. [2017] form a population of model architectures. The population is evolved over time by picking individuals and mutating them (e.g., inserting a new layer). The quality of the population improves over time as the individuals with poor performance are removed.
- Morphing methods Gordon et al. [2018], Cortes et al. [2017] start with an initial model architecture and they iteratively refine the architecture during training until a certain objective is hit (e.g., model size or flops per inference).
- Performance prediction methods Istrate et al. [2019], Deng et al. [2017], Baker et al. [2017], Liu et al. [2018], Luo et al. [2018]. Given a candidate model architecture, these methods forecast its performance without training. In order to train the performance

[2] The pre-trained modules are available via the Tensorflow Hub service (`https://www.tensorflow.org/hub`).

DATA SET	TRAIN EXAMPLES	VAL. EXAMPLES	TEST EXAMPLES	CLASSES
AIRLINE	11712	1464	1464	3
CORPORATE MESSAGING	2494	312	312	4
EMOTION	32000	4000	4000	13
DISASTERS	8688	1086	1086	2
GLOBAL WARMING	3380	422	423	2
POLITICAL BIAS	4000	500	500	2
POLITICAL MESSAGE	4000	500	500	9
PROGRESSIVE OPINION	927	116	116	3
PROGRESSIVE STANCE	927	116	116	4
US ECONOMY	3961	495	496	2

Table 1. Statistics for the NLP classification data sets. Number of examples in the training set, validation set and test set and number of classes. All data sets are publicly available from `crowdflower.com`.

predictor, a database of previous trainings of various model architectures is typically built.

The proposed framework also belongs to the last category of performance prediction methods. However, our prediction network is task-aware and takes as input not only the architecture but also meta-features about the task, with the extra ability of learning them directly from the raw task samples (see Sec. 3.1 for more details). Hence, the proposed framework in its current form is novel (to the best of our knowledge). However, it shares connections and similarities with existing works that we outline below.

The previously proposed SMAC method Hutter et al. [2011] for general algorithm configuration also uses a history of past configuration experiments as well as descriptive features for the problem instances. However, this method uses an expensive Bayesian optimization process as opposed to the efficient gradient-based architecture search that this framework proposes. The other main difference is that the meta-features in SMAC are pre-computed as opposed to the proposed method that learns the meta-features directly from the data samples (while training the performance prediction network).

Also, relatively few works exist which learns across different data sets if we limit the literature to neural architecture search. We discuss below a few of them. The state-of-the-art method in Wong et al. [2018] proposes a multi-task training of RL-based architecture search methods. For each task, it learns a task embedding which is provided as extra input to the controller at each time step. In contrast to our work where the task embedding is explicitly learned from the data samples of the task, the task embedding in Wong et al. [2018] is implicitly learned from the task id. Also, this method still requires some child model trainings on the test task as opposed to our method that requires no child model trainings. We provide comparisons with this method in Section 5 below.

The TAPAS system proposed in Istrate et al. [2019] also uses a history of past configuration experiments stored in a database of experiments. The paper proposes a performance predictor that takes into account the difficulty of the dataset as well as a candidate network architecture. However, this method uses only pre-computed meta-features and its architecture parametrization is not differentiable.

5 EXPERIMENTS

We implemented the framework in TensorFlow Abadi et al. [2016]. Following the same experimental setup as in Wong et al. [2018] we use publicly available NLP data sets for the experimental validation, whose main characteristics are shown in Table 1. We start by explaining the experimental setup in details below.

5.1 Setup

Child models and search space. The child models have been implemented using the parametrization discussed in Section 3.4. The sizes of the base layers in a single parametrized layer are $\{8, 16, 32, 64, 128, 256\}$ and each one of them is combined with two distinct activation functions (`relu` and `tanh`). Hence a single parametrized layer is composed of twelve base layers and each child model has seven such parametrized layers. In addition, each child model has seven embedding modules. Hence, the resulting architecture search space consists of $7 + 7 \cdot 12 + 2 \cdot 7 = 105$ dimensions, which is rather high-dimensional.

Performance prediction network. The network was trained on the child model training experiments stored in the database, which was populated with about 1500 random child model architectures per task. We used a small network consisting of two fully connected layers of size 50 each for the task meta-features tower (aka $\phi(\cdot)$ in Fig. 2) and two fully connected layers of sizes 50 and 10 for the tower that produces the final prediction (aka $\rho(\cdot)$ in Fig. 2). One thing we found beneficial is to introduce gating weights on the dimensions of the architecture vector u. Before u is fed to the first layer of the performance prediction network, each component u_i is multiplied with a gating weight $1/(1 + \exp(-w_i))$, where w_i is also learned together with the rest of the network parameters. This helps the network focus on the most influential parameters of the architecture space.

The network used standard L2 loss for regression and was trained using Stochastic Gradient Descent with momentum Qian [1999] (using 0.5 as default parameter). The learning rate was set to 10^{-4}. We set `kOuterIters` to 1 and `kInnerIters` to 2 in Algorithm 1. When training the network we normalized the child performances $\tilde{v}_i := (v_i - \mu_k)/\sigma_k$ using the mean μ_k and standard deviation σ_k of the population of child performances for a certain task k. Each task has its own level of difficulty and we noticed that this normalization step factors out the difficulty of the task and improves the performance of the prediction network.

5.2 Predicting the model architecture performance

We have performed several leave-one-out experiments, where each task in our set is considered to be a test task and the rest of the tasks being used as training tasks. Then for each such leave-one-out experiment, we train a performance prediction network and study its predictive performance. In particular, given the predicted performances and their corresponding actual performances, we quantify the predictive performance in terms of the Spearman's rank correlation

TASK NAME	SPEARMAN'S RANK CORRELATION		PEARSON CORRELATION	
	WITHOUT	WITH	WITHOUT	WITH
AIRLINE	0.6925 ± 0.0137	0.7454 ± 0.0237	0.8703 ± 0.0078	0.9037 ± 0.0108
EMOTION	0.6785 ± 0.0172	0.7126 ± 0.0189	0.7996 ± 0.0094	0.8230 ± 0.0077
GLOBAL WARMING	0.6357 ± 0.0177	0.6633 ± 0.0204	0.7420 ± 0.0097	0.7539 ± 0.0100
CORP MESSAGING	0.5980 ± 0.0144	0.6316 ± 0.0157	0.6235 ± 0.0121	0.6299 ± 0.0101
DISASTERS	0.6209 ± 0.0131	0.6613 ± 0.0272	0.8145 ± 0.0152	0.8443 ± 0.0168
POLITICAL MESSAGE	0.3144 ± 0.0140	0.3403 ± 0.0178	0.7369 ± 0.0231	0.7820 ± 0.0211
POLITICAL BIAS	0.3421 ± 0.0140	0.3643 ± 0.0100	0.3198 ± 0.0106	0.3338 ± 0.0105
PROGRESSIVE OPINION	0.6183 ± 0.0202	0.6626 ± 0.0239	0.5511 ± 0.0133	0.5588 ± 0.0119
PROGRESSIVE STANCE	0.5713 ± 0.0123	0.5969 ± 0.0122	0.5295 ± 0.0098	0.5276 ± 0.0137
US ECONOMY	0.1731 ± 0.0193	0.1817 ± 0.0091	0.6809 ± 0.0170	0.7181 ± 0.0238

Table 2. The effect of the meta-features: correlations between the actual performances and the predicted performances provided by the performance prediction network; breakdown by task. The correlation is measured by the Spearman's rank correlation (left) and the Pearson correlation (right). The higher the better. The meta-features help the performance prediction network to make better predictions.

TASK NAME	FIRST10	NAS	CHILD MODELS	PROPOSED
AIRLINE	0.7904 ± 0.0366	0.83197	751	0.8260 ± 0.0043
GLOBAL WARMING	0.7806 ± 0.0249	0.79196	1927	0.8066 ± 0.0159
DISASTERS	0.8193 ± 0.0105	0.83425	1283	0.8242 ± 0.0082
POLITICAL BIAS	0.7770 ± 0.0151	0.778	1989	0.7728 ± 0.0161
PROGRESSIVE OPINION	0.6750 ± 0.0428	0.73276	1505	0.7250 ± 0.0191
PROGRESSIVE STANCE	0.4181 ± 0.0645	0.57759	1635	0.5162 ± 0.0222
US ECONOMY	0.7494 ± 0.0112	0.76411	1966	0.7509 ± 0.0140
CORPORATE MESSAGING	0.8006 ± 0.0492	0.85897	968	0.8519 ± 0.0262
EMOTION	0.2998 ± 0.0278	0.35425	1779	0.3480 ± 0.0238
POLITICAL MESSAGE	0.4230 ± 0.0075	0.414	1974	0.4264 ± 0.0070

Table 3. Comparison with NAS. The table shows the test accuracy achieved by the top model according to the validation accuracy that NAS found. The number of child models that NAS trained in order to achieve this test accuracy is also reported. For the sake of completeness, we report the statistics of the test accuracy obtained by the first 10 models that NAS produced. Notice that the proposed method (without any child model training on the test task) achieves test accuracy which is close to that of NAS in the majority of cases.

coefficient. In order to get more accurate results we repeat this process ten times (*i.e.*, train the prediction network ten times) and we report the statistics of the obtained performances. The left column in Table 2 shows the obtained Spearman's rank correlations for each task. For the sake of completeness we also report the Pearson correlation values. Notice that in most cases, the Spearman's rank correlations are higher than 0.6, which seems rather satisfactory for a method that does not use any child model trainings on the test task.

We have also studied experimentally the effect of the meta-features and report the predictive performances with and without meta-features in Table 2 as well. The results in the table suggest that the meta-features are helpful, as expected. The meta-features provide task-specific important information to the prediction network that helps towards estimating more accurately the relative performance of various architectures across various tasks.

5.3 Architecture search

In this section we look into the performance of the child model architectures suggested by our method and we report their test accuracy. When we apply our algorithm we pick the initial guesses using the top five architectures from the two closest training tasks in the task embedding space. Hence we set `kNumStartingPoints` to 10 and `kMaxIters` to 1000 in Algorithm 2. Each experiment is repeated ten times (including training the prediction network from scratch ten times) in order to get more accurate statistics on the performances. We perform architecture search in the continuous space; we leave the discretization of the found architectures for future work.

Comparison with NAS. We compare against the NAS method for neural architecture search using Reinforcement Learning Zoph and Le [2017]. In order to have a fair comparison, we applied NAS on the same child models as our method. Hence, both methods have the same search space and the same child model training and evaluation process. Table 3 shows for each test task, the test accuracy of the child model that NAS found as having the best validation accuracy on the test task. We report also the number of trained child models that were needed for achieving this accuracy. For the sake of completeness, the table also includes the performances of the first 10 models that NAS produced. Notice that the performance of the proposed method is not too far from that of NAS. This is very promising given that the proposed method requires no child model training in its online phase and is very efficient.

Comparison with T-NAML. Next we compare with the state-of-the-art method T-NAML Wong et al. [2018] that also uses transfer learning and cross learns from several tasks. Following a similar experimental methodology to Wong et al. [2018] we split the tasks into five training tasks: GLOBAL WARMING, CORPORATE MESSAGING, POLITICAL MESSAGE, POLITICAL BIAS, PROGRESSIVE STANCE and five test tasks: AIRLINE, EMOTION, DISASTERS, PROGRESSIVE OPINION, US ECONOMY. The controller of T-NAML and the perfor-

TASK NAME	FIRST10 OF T-NAML	PROPOSED
AIRLINE	0.8168 ± 0.0183	0.8275 ± 0.0035
EMOTION	0.3315 ± 0.0236	0.3558 ± 0.0104
DISASTERS	0.8253 ± 0.0137	0.8367 ± 0.0040
PROGRESSIVE OPINION	0.7224 ± 0.0288	0.7276 ± 0.0155
US ECONOMY	0.7558 ± 0.0101	0.7484 ± 0.0104

Table 4. Comparison with the first ten models of T-NAML in terms of test accuracy on the five test tasks. Notice that the proposed method (without any child model training on the test task) achieves higher test accuracy in most cases.

mance prediction network of our method is trained on the same set of training tasks. In order to have a fair comparison, each method uses 1000 child models per training task. Once the controller of T-NAML has been pre-trained on the five training tasks, it is applied on each test task by training child models on the test task and fine-tuning its parameters. In order to keep the resource consumption manageable, we focus our comparison on the first ten models that T-NAML produces.

Table 4 shows the test accuracy results. Notice that the proposed method compares favourably with the first ten models that T-NAML produces in most cases. This might be due to the fact that in T-NAML the task embedding for the test task is randomly initialized (which is also known as the *cold start* problem). On the contrary, our method computes the task embedding directly from the task data samples which seems to generalize better. At the same time the proposed method is ten times more efficient since it does not train any child models on the test task.

TASK NAME	TRAINING TIME (SECS)
AIRLINE	1170
GLOBAL WARMING	696
DISASTERS	830
POLITICAL BIAS	695
PROGRESSIVE OPINION	442
PROGRESSIVE STANCE	445
US ECONOMY	760
CORPORATE MESSAGING	658
EMOTION	2409
POLITICAL MESSAGE	669

Table 5. Median of the time (in secs) required to train a single child model per task. On the contrary our method requires about one minute in its online phase to suggest a good architecture.

Discussion. Table 5 helps us to better understand and quantify the computational advantage of the proposed method. The table reports statistics about the timings required to train one single child model; breakdown by task. The statistics have been obtained from 2000 child model trainings for each task. The baseline methods we are comparing against, typically need multiples of such child model trainings. On the other hand, 1000 steps of gradient ascent used by our method require typically about 6 secs. Repeating the gradient ascent with 10 different initial guesses shows that the proposed method requires only about one minute in its online phase to suggest a good architecture.

5.4 Visualization of the task meta-features

We also looked into the learned task representations in the meta-feature space. In particular, for each task we computed the corresponding task embedding for different random batch realizations (of the task samples) and visualized them with t-SNE Van der Maaten and Hinton [2008]. Fig. 4 show the two-dimensional visualizations obtained with t-SNE with 10 random batches per task and perplexity set to 70. Interestingly, different batch realizations from the same task result in close-by task embeddings in the meta-feature space, which confirms the stability of the method in this respect.

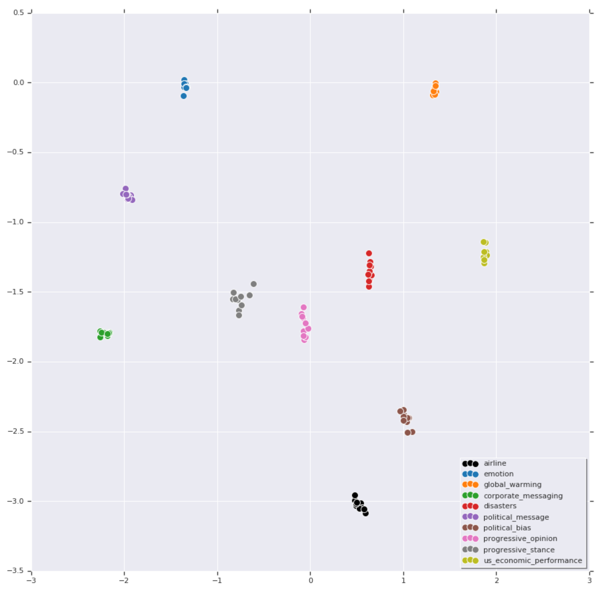

Figure 4. Visualization of the learned meta-features using t-SNE. For each task we show the (two-dimensional projections of the) meta-features computed from 10 random batches of the task samples. Different random batch realizations from the same task result in close-by task embeddings in the meta-feature space.

6 CONCLUSIONS AND FUTURE WORK

We presented a framework for efficient architecture inference that cross learns from several tasks. This is feasible thanks to a prediction network that estimates the performance of a candidate architecture on a certain task based on learned meta-features derived from the raw data samples of the task. Given a new task, the proposed method uses efficient gradient ascent to infer a candidate architecture for it and experimental results confirm that the performance of the found architecture is directly comparable to that of more expensive baselines. In our future work, we plan to study the effect of pruning/discretizing the found architecture and apply the method to other data modalities beyond text (e.g., images). Finally, we are going to look into different strategies for populating the database of experiments.

ACKNOWLEDGEMENTS

The authors would like to thank Thomas Deselaers, Sergey Ioffe and Daniel Golovin for their valuable comments, fruitful discussions and support. We would also like to thank the referees for their comments, which helped improve this paper considerably.

REFERENCES

M. Abadi, P. Barham, . Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. *USENIX Conference on Operating Systems Design and Implementation*, 2016.

B. Baker, O. Gupta, R. Raskar, and N. Naik. Accelerating Neural Architecture Search using Performance Prediction. *arXiv preprint*, November 2017. URL arXiv:1705.10823.

G. Bender, P.-J. Kindermans, B. Zoph, V. Vasudevan, and Q. Le. Understanding and simplifying one-shot architecture search. *International Conference on Machine Learning (ICML)*, 2018.

H. Cai, L. Zhu, and S. Han. ProxyLessNAS: Direct neural architecture search on target task and hardware. *International Conference on Learning Representations (ICLR)*, 2019.

C. Cortes, X. Gonzalvo, V. Kuznetsov, M. Mohri, and S. Yang. AdaNet: Adaptive Structural Learning of Artificial Neural Networks. *International Conference on Machine Learning (ICML)*, 2017.

B. Deng, J. Yan, and D. Lin. Peephole: Predicting network performance before training. *arXiv preprint*, December 2017. URL arXiv:1712.03351.

M. Feurer, A. Klein, K. Eggensperger, J. Springenberg, M. Blum, and F. Hutter. Efficient and Robust Automated Machine Learning. *Conference on Neural Information Processing Systems (NIPS)*, 2015a.

M. Feurer, J. T. Springenberg, and F. Hutter. Initializing Bayesian Hyperparameter Optimization via Meta-Learning. *AAAI Conference on Artificial Intelligence*, 2015b.

N. Fusi, R. Sheth, and H. M. Elibol. Probabilistic Matrix Factorization for Automated Machine Learning. *Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

A. Gordon, E. Eban, O. Nachum, B. Chen, H. Wu, T.-J. Yang, and E. Choi. MorphNet: Fast & Simple Resource-Constrained Structure Learning of Deep Networks. *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

M. Gygli, M Norouzi, and A. Angelova. Deep value networks learn to evaluate and iteratively refine structured outputs. *International Conference on Machine Learning (ICML)*, 2017.

F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. *5th International Conference on Learning and Intelligent Optimization*, pages 507–523, 2011.

R. Istrate, F. Scheidegger, G. Mariani, D. Nikolopoulos, C. Bekas, and A. C. I. Malossi. TAPAS: Train-less Accuracy Predictor for Architecture Search. *AAAI Conference on Artificial Intelligence*, 2019. URL arXiv:1806.00250.

C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy. Progressive Neural Architecture Search. In *European Conference on Computer Vision (ECCV)*, September 2018.

H. Liu, K. Simonyan, and Y. Yang. DARTS: Differentiable architecture search. In *International Conference on Learning Representations (ICLR)*, 2019.

R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu. Neural architecture optimization. *Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

H. Mendoza, A. Klein, M. Feurer, J. T. Springenberg, and F. Hutter. Towards Automatically-Tuned Neural Networks. *JMLR: Workshop and Conference Proceedings*, 1:1–8, 2016.

H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean. Efficient neural architecture search via parameters sharing. *International Conference on Machine Learning (ICML)*, 2018.

N. Qian. On the momentum term in gradient descent learning algorithms. *Neural Netw.*, 12(1):145–151, January 1999. ISSN 0893-6080. doi: 10.1016/S0893-6080(98)00116-6.

E. Real, S. Moore, A. Selle, S. Saxena, Y. L. Suematsu, J. Tan, Q. V. Le, and A. Kurakin. Large-Scale Evolution of Image Classifiers. *International Conference on Machine Learning (ICML)*, 2017.

R. Shin, C. Packer, and D. Song. Differentiable Neural Network Architecture Search. *Workshop track - ICLR*, 2018.

J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. *Conference on Neural Information Processing Systems (NIPS)*, 2012.

L. Van der Maaten and G. Hinton. Visualizing Data using t-SNE. *Journal Journal of Machine Learning Research*, 9:2579–2605, 2008.

J. Vanschoren. Meta-Learning: A Survey. *https://arxiv.org/abs/1810.03548*, 2018. URL arXiv:1806.00250.

C. Wong, N. Houlsby, Y. Lu, and A. Gesmundo. Transfer Learning with Neural AutoML. *Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

S. Xie, H. Zheng, C. Liu, and L. Lin. SNAS: Stochastic Neural Architecture Search. *International Conference on Learning Representations (ICLR)*, 2019.

M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. Salakhutdinov, and A. Smola. Deep Sets. *Conference on Neural Information Processing Systems (NIPS)*, 2017.

C. Zhang, M. Ren, and R. Urtasun. Graph HyperNetworks for Neural Architecture Search . *International Conference on Learning Representations (ICLR)*, 2019.

B. Zoph and Q. V. Le. Neural Architecture Search with Reinforcement Learning. *International Conference on Learning Representations (ICLR)*, 2017.