LAMANAS: Loss Agnostic and Model Agnostic Meta Neural Architecture Search for Few-Shot Learning

Stanford CS229 Project: Theory and Reinforcement Learning

Sergio Charles
Department of Computer Science
Stanford University
sergioc1@stanford.edu

Gordon Chi
Department of Computer Science
Stanford University
gsychi@stanford.edu

Gil Kornberg
Department of Computer Science
Stanford University
gilk@stanford.edu

Abstract

We present *LAMANAS*[1], a novel neural architecture search method that is loss and model agnostic. The algorithm searches for and trains a network architecture that generalizes well and adapts quickly to unseen tasks. This is achieved by finding high-performing, meta-learned model and architecture parameter initializations using a self-supervised loss. The loss is parameterized as a neural network which allows the neural architecture search to learn an optimal loss landscape for each task without imposing a strong prior. Using a simple long-short term memory (LSTM) recurrent neural network for the loss architecture in tandem with a inner product loss proxy, between the gradients of the self-supervised loss and gold-standard cross entropy, yields state of the art improvements over MetaNAS.

Mentor: Yao Liu & Rachel Gardner

1 Introduction

In recent years, the machine learning community has made several significant strides in the quest toward Artificial General Intelligence (AGI), and away from "narrow" intelligence. In 2015, motivated by the human capacity to generalize concepts successfully after seeing only one or a few examples, a phenomenon known as one-shot learning, [1] introduced the Omniglot and MiniImagenet datasets and their corresponding benchmarks. Models must be equipped to learn robust and flexible representations if data from only a small set of examples if they are to be successful in the few-shot setting, while maintaining speed and efficiency. In parallel, Model Agnostic Meta-Learning (MAML) [2] introduced a model-agnostic gradient-based approach proposed by Finn et al. that optimizes parameters of a model for rapid adaptation to new tasks. MAML finds good model initializations such that adaptation to a new task is efficient and can be achieved in a few-shot setting. Neural architecture search (NAS) was proposed in 2017 [3] to automatically learn network architectures that maximize performance on a specific task. It does so by using an RNN meta-controller to predict a sequence of tokens that specify architectural hyper-parameters of the learned architecture.

In 2020, MetaNAS [4] combined gradient-based neural architecture search (NAS) methods with gradient-based meta-learning methods. They used a flexible model architecture during meta learning, which enabled fast and cheap task adaptation, achieving state-of-the-art performance on the standard few-shot learning benchmarks, Omniglot and MiniImageNet, at the time of publication. Finally, [5] introduced Self-Adaptive Visual Navigation (SAVN) which aims to design robust and flexible learning algorithms for robotics domains. This is accomplished using a self-supervised loss, meaning the agent learns on its own as it interacts with the environment. The authors point out what all humans know: that learning is a continuous process ad infinitum. Inference need not come at the expense of training. Whether learning to learn or learning to learn how to learn, model agnostic or loss agnostic, the aim of these approaches is to achieve high-performance and good model generalizability, and to do so using minimal resources. Hence, we propose *LAMANAS* for loss and model agnostic meta learning of neural architectures for few-shot learning.

2 Related Work

[1]The code implementation is on GitHub at: https://github.com/sergiogcharles/lamanas

35th Conference on Neural Information Processing Systems (NeurIPS 2021), Sydney, Australia.

NAS [3] proposed the serialization of neural network topologies whereby the architecture could be encoded as a sequence of tokens, i.e. operations at each layer. They were able to train an RNN meta controller to produce the sequence of architectural hyperparameters $a_{1:T}$, which is a sequence of actions in the reinforcement learning-theoretic sense [3]. NAS works by using the RNN controller to sample a candidate child model, training it to convergence and then evaluating the reward R, usually measured by the accuracy, of the child model on a held-out validation set, which induces a controller update signal. However, the method uses a REINFORCE [6] proximal policy-based reinforcement learning, which is prohibitively expensive. In light of this, Liu et al. introduced Differentiable Architecture Search (DARTS) [7] that characterizes neural architecture search as finding architectures as sub-network graphs of the directed acyclic super-network graph, which eschewed the need for an RNN controller. It also stacks the learned computation normal and reduction cells to form a convolutional neural network, as introduced by NASNet [8].

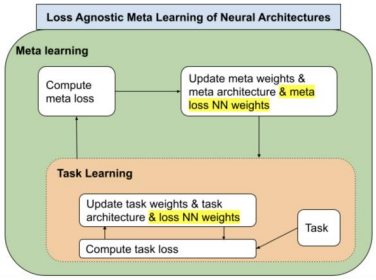

Figure 1: An overview of the LAMANAS algorithm.

MAML [2] is a meta-learning algorithm, proposed by Finn et al., that optimizes parameters for rapid adaptation to new tasks in the few-shot setting. Each task $\mathcal{T}_i$ in the large training set $\mathcal{T}_{\text{train}}$ has a small meta training $\mathcal{D}_{\text{train}}^{\mathcal{T}_i}$ dataset and meta validation dataset $\mathcal{D}_{\text{val}}^{\mathcal{T}_i}$. In the case of n-way, k-shot image classification, $\mathcal{T}_i$ consists of the n image classes, $\mathcal{D}_{\text{train}}^{\mathcal{T}_i}$ has k examples for each of the classes, the objective is to correctly assign class labels to images in $\mathcal{D}_{\text{val}}^{\mathcal{T}_i}$ [5], and we evaluate on the test task $\mathcal{T}_{\text{test}}$ of unseen classes. The MAML training objective is given by:

$$\min_{\theta} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} \mathcal{L}(\theta - \lambda \nabla_{\theta} \mathcal{L}(\theta, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}), \mathcal{D}_{\text{val}}^{\mathcal{T}_i}). \tag{1}$$

where θ is optimized to provide good initializations, which allows for fast adaptation to unseen test tasks. We adapt the parameters for the task on the training set $\mathcal{D}_{\text{train}}^{\mathcal{T}_i}$ by performing the task learner update $\theta - \alpha \nabla_{\theta} \mathcal{L}(\theta, \mathcal{D}_{\text{train}}^{\mathcal{T}_i})$ iteratively and then optimize θ on the task validation set $\mathcal{D}_{\text{val}}^{\mathcal{T}_i}$.

Given model parameter initializations θ, let $W_{\mathcal{T}_i}$ denote the manifold of optimal parameters for the task $\mathcal{T}_i$. Then Reptile [9] is a gradient-based meta-learning method that finds the parameters θ^* close to all of the manifolds of optimal parameters for all tasks. Namely, for a metric on parameters space $d(\theta, \mathcal{W}_{\mathcal{T}_i})$, it will optimize $\min_{\theta} \mathbb{E}_{\mathcal{T}_i} \left[\frac{1}{2} d(\phi, \mathcal{W}_{\mathcal{T}_i})^2\right]$ by performing SGD such that the distance between θ and the optimal manifold $\mathcal{W}_{\mathcal{T}_i}$ is small for all tasks. MetaNAS [4] is a method, proposed by Elsken et al., that combines gradient-based neural architecture search (NAS) methods, such as DARTS [7], with gradient-based meta-learning methods, such as MAML [2]. It optimizes meta-architecture parameters α_{meta} in tandem with meta-model parameters θ_{meta} during meta-training. The meta-parameters α_{meta} and θ_{meta} are able to adapt quickly to a new task $\mathcal{T}_i$ with only a few labeled data points, i.e. for n-way, k-shot tasks. That is, it can adapt the meta-architecture to a *task dependent architecture* [4]. While MetaNAS presents a model-agnostic neural architecture search, it still introduces strong priors with hand-crafted task objectives $\mathcal{L}_{\text{train}}^{\mathcal{T}_i}$. Finally, SAVN [5] introduces a loss-agnostic approach to MAML by learning a self-supervised task interaction objective, which proves useful in "learning how to learn" based on an agent's trajectory.

3 Dataset & Features

All experiments were run on the Omniglot dataset, which consists of 1623 unique characters taken from 50 alphabets hand-drawn in pen or pencil. We follow the same evaluation method used in [4], namely, the n-way, k-shot setting as proposed by [10]. A few-shot learning task is constructed by first sampling n classes at random from Omniglot and then sampling k examples for each class. We used $n = 20$ and $k = 5$ for a 20-way, 5-shot model evaluation setting. Each example in Omniglot is $(1, 28, 28)$, each batch is $(20, 1, 28, 28)$, and we use 1 test example per class. Lastly, we use a Vinyals split as in [10]. The architecture always begins with a constant *stem* which is simply a 2-D convolution followed by a 2-D batch normalization layer. The rest of the architecture is determined by a search of the architecture space over a set of candidate operations, e.g., 3 x 3 convolutions, 3 x 3 average pooling, and the zero operation. The learned architecture always performs feature extraction, and we will explore different learned architectures in Section 5.

4 Method

Adopting the notation of MetaNAS and SAVN, $(\mathcal{D}_{\text{train}}^{\mathcal{T}_i}, \mathcal{D}_{\text{val}}^{\mathcal{T}_i})$ is the sampled task, $\mathcal{L}_{\text{val}}^{\mathcal{T}_i}$ is the supervised loss for the task $\mathcal{T}_i$, Φ_K is a K-step task learner algorithm like SGD, and (θ, α) are model and architecture parameters, respectively. For task $\mathcal{T}_i$, $\theta_{\mathcal{T}_i} := \theta_i$ are the model parameters, $\alpha_{\mathcal{T}_i} := \alpha_i$ are the architecture parameters, and $(\theta_i^*, \alpha_i^*) = \Phi_K(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i})$ are the optimized parameters. We denote the meta-parameters by θ and α. Methods like DARTS hold α constant so we optimize the task model parameters θ_i by using the task learner $\Phi_K(\theta, \alpha_{\text{constant}}, \mathcal{D}_{\text{train}}^{\mathcal{T}_i})$ which applies the following one-step SGD update K times: $\theta_i := \Phi_1(\theta_i, \alpha_{\text{constant}}, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}) = \theta_i - \lambda_{\text{task}} \nabla_{\theta} \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta_i, \alpha_{\text{constant}}, \mathcal{D}_{\text{train}}^{\mathcal{T}_i})$ [4].

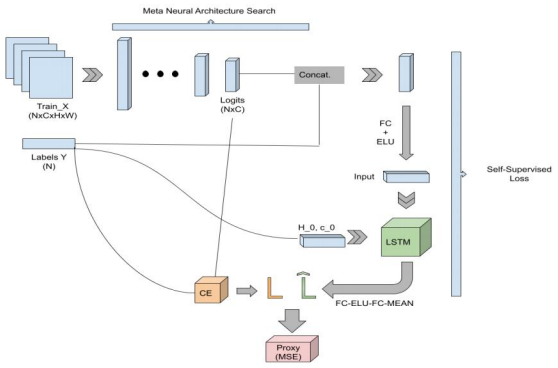

Figure 2: Details of the LAMANAS algorithm.

In constrast to these methods, we construct a K-step task learner Φ_K that, for task $\mathcal{T}_i$, optimizes both the task model parameters θ_i and the task architecture parameters α_i with model learning rate λ_{task} and architecture learning rate ξ_{task}, commensurate to MetaNAS. That is, we write the one-step update as [4]:

$$\begin{bmatrix}\theta_i \\ \alpha_i\end{bmatrix} = \Phi_1(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}) := \begin{bmatrix}\theta_i - \lambda_{\text{task}} \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}) \\ \alpha_i - \xi_{\text{task}} \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i})\end{bmatrix} \tag{2}$$

for $\mathcal{L}_{\text{train}}^{\mathcal{T}_i}$ the task's self-supervised training loss. We repeat this update K times until we converge to the optimized task parameters θ_i^* and α_i^*:

$$\begin{bmatrix}\theta_i^* \\ \alpha_i^*\end{bmatrix} = \Phi_K(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}).$$

We use the following meta-objective to find a meta-architecture with parameters that are fast adapters when given new tasks:

$$\min_{\theta,\alpha} \mathcal{L}_{\text{meta}}(\theta, \alpha, \Phi_K) = \min_{\theta,\alpha} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\Phi_K(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}), \mathcal{D}_{\text{val}}^{\mathcal{T}_i}) = \min_{\theta,\alpha} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta_i^*, \alpha_i^*, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}) \tag{3}$$

Furthermore, we update the meta-objective with a meta-learning algorithm like MAML:

$$\begin{bmatrix}\theta \\ \alpha\end{bmatrix} = \Psi(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}) = \begin{bmatrix}\theta - \lambda_{\text{meta}} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} \nabla_\theta \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta_i^*, \alpha_i^*, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}) \\ \alpha - \xi_{\text{meta}} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} \nabla_\alpha \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta_i^*, \alpha_i^*, \mathcal{D}_{\text{val}}^{\mathcal{T}_i})\end{bmatrix} \tag{4}$$

where θ_i^* and α_i^* denote the parameters optimized by the K-step task learner. For notational simplicity, we will eschew the explicit reference to the K-step task learner Φ_K and, instead, simply write one step of the SGD update inside our meta-learning objective:

$$\min_{\theta,\alpha} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta - \lambda_{\text{task}} \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}), \alpha - \xi_{\text{task}} \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}), \mathcal{D}_{\text{val}}^{\mathcal{T}_i}) \tag{5}$$

where we, implicitly, mean that we perform the K-step task learner updates to θ and α, using the same dataset $\mathcal{D}_{\text{train}}^{\mathcal{T}_i}$. It is not prudent to split the dataset in the small data regime of few-shot learning [5]. We let $\mathcal{L}_{\text{train}}^{\mathcal{T}_i}$ be a feed-forward neural network parameterized by ϕ. The training objective will be a modified version of Equation 5 (see Appendix for a proof of the approximation):

$$\begin{aligned}
&\min_{\theta,\alpha} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta - \lambda_{\text{task}} \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi), \alpha - \xi_{\text{task}} \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi), \mathcal{D}_{\text{val}}) \approx \\
&\min_{\theta,\alpha} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}) - \lambda_{\text{task}} \langle \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi), \nabla_\theta \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}) \rangle - \xi_{\text{task}} \langle \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi), \nabla_\alpha \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}) \rangle
\end{aligned} \tag{6}$$

where ϕ is fixed during inference, according to SAVN [5]. Algorithm 1 shows a detailed implementation of the loss-agnostic MetaNAS approach using MAML for the meta-optimizer, DARTS for neural architecture search, and SGD for the K-step task learner. Algorithm 2 shows a varied implementation using Reptile [9] as the meta-learner, whereby in order to update the self-supervised

loss meta parameter ϕ, we must update it in our task-learner according to the loss of the neural network $\nabla_\phi \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi_i)$.

Algorithm 1 LAMANAS: Loss and Model Agnostic Meta Neural Architecture Search with DARTS and Reptile

Require: Distribution $p(\mathcal{T}_{\text{train}})$ over tasks
  Randomly initialize θ, α, ϕ
  while not converged **do**:
    Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})$
    for all $\mathcal{T}_i$ **do**:
      $\theta_i \leftarrow \theta$
      $\alpha_i \leftarrow \alpha$
      for $j = 1, \ldots, K$ **do**:
        $\theta_i \leftarrow \theta_i - \lambda_{\text{task}} \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi)$
        $\alpha_i \leftarrow \alpha_i - \xi_{\text{task}} \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi)$
      end for
    end for
    $\theta \leftarrow \theta - \lambda_{\text{meta}} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} \nabla_\theta \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta_i^*, \alpha_i^*, \mathcal{D}_{\text{val}}^{\mathcal{T}_i})$
    $\alpha \leftarrow \alpha - \xi_{\text{meta}} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} \nabla_\alpha \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta_i^*, \alpha_i^*, \mathcal{D}_{\text{val}}^{\mathcal{T}_i})$
    $\phi \leftarrow \phi - \chi_{\text{meta}} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} \nabla_\phi \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta_i^*, \alpha_i^*, \mathcal{D}_{\text{val}}^{\mathcal{T}_i})$
  end while

Algorithm 2 LAMANAS Variant with DARTS and MAML

Require: Distribution $p(\mathcal{T}_{\text{train}})$ over tasks
  Randomly initialize θ, α, ϕ
  while not converged **do**:
    Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})$
    for all $\mathcal{T}_i$ **do**:
      $\theta_i \leftarrow \theta$
      $\alpha_i \leftarrow \alpha$
      $\phi_i \leftarrow \phi$
      for $j = 1, \ldots, K$ **do**: ▷ Task learner with K update steps. finds θ_i^* and α_i^*
        $\theta_i \leftarrow \theta_i - \lambda_{\text{task}} \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi)$
        $\alpha_i \leftarrow \alpha_i - \xi_{\text{task}} \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi)$
      end for
      $\phi_i \leftarrow \phi_i - \chi_{\text{task}} \nabla_\phi \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi)$
    end for
    ▷ Meta learner update via Reptile, sampling new tasks
    $\theta \leftarrow \theta + \lambda_{\text{meta}} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} (\theta_i^* - \theta)$
    $\alpha \leftarrow \alpha + \xi_{\text{meta}} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} (\alpha_i^* - \alpha)$
    $\phi \leftarrow \phi + \chi_{\text{meta}} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} (\phi_i^* - \phi)$
  end while

5 Experiments & Results

As such, we conducted experiments using the meta neural architecture search method in Algorithm 2 by searching for a CNN that consists of stacked computation cells, like in DARTS [7] and NASNet [8]. We employed two types of cells, namely a *normal cell* which preserves the input dimension and a *reduction cell* which halves the output dimension by using a stride of 2. We designed many variants of the learned self-supervised loss neural network $\mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi_i)$ for each task $\mathcal{T}_i$ by varying its architecture as a feed-forward neural network (FNN) or a long short-term memory (LSTM) RNN. We used DARTS as the task optimizer and Reptile [9] as the meta learner for 200 meta epochs with 5 tasks in each meta batch. Furthermore, we experimented with the use of a cross entropy residual connection in the learned loss. In particular, during the task learner update, we computed the cross entropy between the CNN logits $z^{\mathcal{T}_i}$ and the ground truth labels $y_{\text{train}}^{\mathcal{T}_i}$ and added it to the first layer output of the FNN/LSTM learned loss.

We leveraged a meta loss proxy to guide back propagation of the loss neural network and perform the third SGD task learner update in Algorithm 2. For the first proxy, we measure how well the learned loss approximates the gold-standard loss, i.e. cross entropy loss $\mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}) = CE(z_i, y_{\text{train}}^{\mathcal{T}_i})$ for each task $\mathcal{T}_i$. This was modeled by maximizing the similarity between the output of the cross entropy loss and the learned loss using the following mean squared error difference:

$$\mathcal{L}_{\text{proxy}}^{\mathcal{T}_i} = \text{MSE}(\text{CE}(z^{\mathcal{T}_i}, y_{\text{train}}^{\mathcal{T}_i}), \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi_i)). \tag{7}$$

Another proxy method we used was to maximize the L_2 inner product similarity between the gradients with respect to both meta parameters θ and α of the self-supervised loss $\mathcal{L}_{\text{train}}^{\mathcal{T}_i}$ and the cross-entropy loss, given by Equation 6:

$$\mathcal{L}_{\text{proxy}}^{\mathcal{T}_i} = \text{CE}(z^{\mathcal{T}_i}, y_{\text{train}}^{\mathcal{T}_i}) - \lambda \langle \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}), \nabla_\theta \text{CE}(z^{\mathcal{T}_i}, y_{\text{train}}^{\mathcal{T}_i}) \rangle - \xi \langle \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}), \nabla_\alpha \text{CE}(z^{\mathcal{T}_i}, y_{\text{train}}^{\mathcal{T}_i}) \rangle. \tag{8}$$

In addition, we investigated the marginal benefits of using pre-trained layers in the meta architecture for the meta model CNN to learn and extract stronger visual representations. For pre-training, we used the first four pre-trained layers of ResNet-18. Before using the ResNet-18 feature extraction, for a batch size of N, we transform our input from $(N, 1, 28, 28)$ to $(N, 3, 224, 244)$ by upsampling and applying a 1x1 convolutional filter. This is followed by another convolutional layer after the 4 layer encoding of the ResNet layers. Unfortunately, our experimental results indicates that this harms performance, attaining low accuracy on the held out test set.

The layers of the FNN and LSTM loss neural networks use orthogonal initialization [11] for dynamical stability. The networks take as input the concatenation of the logits produced by the meta model with the ground truth labels for each batch and embeds it using a linear projection transformation, shown in Figure 2. In both cases, we also have the option of adding the output of the cross-entropy loss on the batch of predictions as a residual connection, which is followed by an ELU non-linearity with a tunable hyperparameter

			Accuracy (%)		
Variant	**Residual Connection**	**Loss Proxy**	**Test**	**Train**	**Mean Number of Parameters**
Baseline	No	N/A	82.5	57.1	413,215
FNN	Yes	MSE	41.6	50.0	398,533
	No	MSE	7.5	6.5	327,590
	Yes	Inner product	43.8	49.9	402,351
	No	Inner product	6.0	5.0	299,200
LSTM	Yes	MSE	**89.8**	**61.0**	398,533
	No	MSE	5.5	5.1	327,590
	Yes	Inner product	**91.4**	**61.3**	435,067
	No	Inner product	6.0	4.8	299,200

Table 1: Train and test accuracy across variants and hyperparameters, i.e. residual connections and loss proxy, compared alongside the baseline model. The best variants is bolded, namely an LSTM loss neural network using a inner product loss proxy and residual connection.

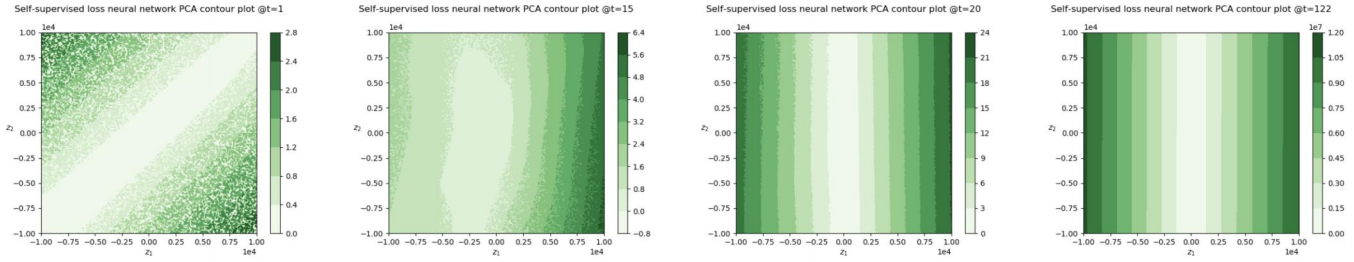

Figure 3: Snapshots of the geometry of self-supervised loss FNN function for the self-supervised loss network over hand-selected meta-epochs 11, 20, 111 and 122. The loss neural network is an FNN with residual connections and MSE loss proxy. *At $t = 1$ epochs, the loss is almost equivalent to cross entropy loss.

defaulted to $\alpha = 1$ which, unlike ReLU, allows negative values to pass. As seen in Table 1, the residual connection significantly improves performance. The FNN architecture includes another linear layer and finally we take the mean over the losses in the minibatch. The LSTM architecture, illustrated in Figure 2, consists of 5 stacked many-to-one LSTM cells which takes as input the outputs of the embedding of the logits in $\mathbb{R}^{N\times H}$, reshaped to lie in $R^{1\times N\times H}$. We initialize the hidden and cell states as the ground truth labels $y^{\mathcal{T}_i}$. The FNN loss neural network achieves an accuracy of 41.6% with a inner product loss proxy, slightly higher than the 43.6% with an MSE loss proxy. As shown in Table 1, the RNN LSTM loss neural network architecture outperforms the benchmark MetaNAS with a train and test accuracy of 61.0% and 91.4%, respectively. We hypothesize that this occurs because loss network is being adapted temporally, which lends itself well to the LSTM mechanism. To plot the loss neural network as a function of two logits z_1 and z_2, as shown in the Figure 3, we used a principal component analysis (PCA) by using a Singular Value Decomposition (SVD) on the model parameter matrix $\Theta \in \mathbb{R}^{m\times}$ where $\Theta = U\Sigma V^\top$ for the first layer of the neural network, $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ are orthogonal, and the diagonal $\Sigma \in \mathbb{R}^{m\times n}$ is the matrix of singular values of Θ. This yields the principal axes of the parameters of the loss neural network. The k-reduced parameter is given by $\hat{\Theta} = \Theta V_k$ where V_k is the first k columns of the orthogonal matrix V. Namely, Figure 3 shows contours plot of the the loss neural network as it adapts over meta epochs to learn the optimal loss landscape for various interesting snapshots. Interestingly, the geometries of the losses being learned are, in certain cases, rotations of the baseline trough-like cross entropy geometry. As we progress in meta training, the loss geometry converges to a trough and absolute height increases with increasingly larger gradients, which could guarantee faster convergence. That is, if the architecture is in state (θ, α), then the amount of work done to increase the self-supervised loss becomes arbitrarily large as we perform meta-training.

6 Conclusion

We have presented a loss and model agnostic meta-learning approach to neural architecture search using a self-supervised loss. We find that the dynamic LSTM self-supervised loss outperformed the constant cross entropy loss used by MetaNAS. In particular, the geometries of the loss function tend have increasingly large curvature which seems to improve training and, consequently, allows the meta architecture search to be a fast adapter. In future work, we intend to analyze the relative merits and limitations of the asymptotic, dynamical stability of such self-supervised loss neural networks, which attain arbitrarily large magnitudes during meta-learning.

7 Appendix

Proof. We invoke a first order Taylor series expansion to prove Equation 6, omitting reference to ϕ. A loss $\mathcal{L}$ is a scalar-valued function $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$, which has second order Taylor series approximation centered around $\boldsymbol{a} \in \mathbb{R}^n$ given by:

$$\mathcal{L}(\boldsymbol{x}) \approx \mathcal{L}(\boldsymbol{a}) + D\mathcal{L}(\boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a}) + \frac{1}{2}(\boldsymbol{x} - \boldsymbol{a})^\top H\mathcal{L}(\boldsymbol{a})(\boldsymbol{x} - \boldsymbol{a}) \tag{9}$$

for $D\mathcal{L}(\boldsymbol{x})$ the matrix of partial derivatives of $\mathcal{L}$ and $H\mathcal{L}(\boldsymbol{x})$ the Hessian of $\mathcal{L}$. We approximate the summand in the following:

$$\min_{\theta,\alpha} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta - \lambda_{\text{task}} \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi), \alpha - \xi_{\text{task}} \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi), \mathcal{D}_{\text{val}}) \tag{10}$$

using the second-order Taylor series expansion:

$$\begin{aligned}\mathcal{L}_{\text{val}}^{\mathcal{T}_i}\left(\begin{bmatrix}\theta - \lambda_{\text{task}} \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i} \\ \alpha - \xi_{\text{task}} \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}\end{bmatrix}, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}\right) \approx \mathcal{L}_{\text{val}}^{\mathcal{T}_i}\left(\begin{bmatrix}\theta \\ \alpha\end{bmatrix}, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}\right) + D\mathcal{L}_{val}^{\mathcal{T}_i}\left(\begin{bmatrix}\theta \\ \alpha\end{bmatrix}, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}\right)\begin{bmatrix}-\lambda_{\text{task}} \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i} \\ -\xi_{\text{task}} \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}\end{bmatrix} \\ + \frac{1}{2}\begin{bmatrix}-\lambda_{\text{task}} \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i} \\ -\xi_{\text{task}} \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}\end{bmatrix}^\top H\mathcal{L}_{\text{val}}^{\mathcal{T}_i}\left(\begin{bmatrix}\theta \\ \alpha\end{bmatrix}, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}\right)\end{aligned} \tag{11}$$

where the matrix of partial derivatives is

$$D\mathcal{L}_{val}^{\mathcal{T}_i}\left(\begin{bmatrix}\theta \\ \alpha\end{bmatrix}, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}\right) = \begin{bmatrix}\nabla_\theta \mathcal{L}_{\text{val}}\left(\begin{bmatrix}\theta \\ \alpha\end{bmatrix}, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}\right)^\top & \nabla_\alpha \mathcal{L}_{\text{val}}\left(\begin{bmatrix}\theta \\ \alpha\end{bmatrix}, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}\right)^\top\end{bmatrix}.$$

Thus, ignoring the second-order Hessian term, we can write Equation 10 as:

$$\begin{aligned}\mathcal{L}_{\text{val}}^{\mathcal{T}_i}\left(\begin{bmatrix}\theta - \lambda_{\text{task}} \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i} \\ \alpha - \xi_{\text{task}} \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}\end{bmatrix}, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}\right) \approx \mathcal{L}_{\text{val}}^{\mathcal{T}_i}\left(\begin{bmatrix}\theta \\ \alpha\end{bmatrix}, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}\right) \\ + \nabla_\theta \mathcal{L}_{\text{val}}^{\mathcal{T}_i}\left(\begin{bmatrix}\theta \\ \alpha\end{bmatrix}, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}\right)^\top \left(-\lambda_{\text{task}} \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i}\left(\begin{bmatrix}\theta \\ \alpha\end{bmatrix}, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}\right)\right) \\ + \nabla_\alpha \mathcal{L}_{\text{val}}^{\mathcal{T}_i}\left(\begin{bmatrix}\theta \\ \alpha\end{bmatrix}, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}\right)^\top \left(-\xi_{\text{task}} \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}\left(\begin{bmatrix}\theta \\ \alpha\end{bmatrix}, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}\right)\right),\end{aligned} \tag{12}$$

implying Equation 6:

$$\begin{aligned}&\min_{\theta,\alpha} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta - \lambda_{\text{task}} \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}), \alpha - \xi_{\text{task}} \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}), \mathcal{D}_{\text{val}}^{\mathcal{T}_i}) \\ &\approx \min_{\theta,\alpha} \sum_{\mathcal{T}_i \sim p(\mathcal{T}_{\text{train}})} \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}) - \lambda_{\text{task}} \langle \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}), \nabla_\theta \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}) \rangle - \xi_{\text{task}} \langle \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}), \nabla_\alpha \mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta, \alpha, \mathcal{D}_{\text{val}}^{\mathcal{T}_i}) \rangle,\end{aligned} \tag{13}$$

which minimizes the supervised validation loss $\mathcal{L}_{\text{val}}^{\mathcal{T}_i}$ and maximizes the similarity between the gradients, with respect to both θ and α, of the self-supervised training loss $\mathcal{L}_{\text{train}}^{\mathcal{T}_i}$ and the supervised validation loss $\mathcal{L}_{\text{val}}^{\mathcal{T}_i}$. Therefore, during inference, when $\mathcal{L}_{\text{val}}^{\mathcal{T}_i}$ is unavailable, we can still perform training if the gradients of the losses are similar [5]. That is, we want the self-supervised loss $\mathcal{L}_{\text{train}}^{\mathcal{T}_i}$ to learn to emulate the supervised loss $\mathcal{L}_{\text{val}}^{\mathcal{T}_i}$. Choosing such a self-supervised $\mathcal{L}_{\text{train}}^{\mathcal{T}_i}$ to guarantee this property is difficult and, thus, it is natural to learn the self-supervised training loss.

□

References

[1] B. M. Lake, R. Salakhutdinov, and J. B. Tenenbaum. Human-level concept learning through probabilistic program induction., 2015.

[2] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks, 2017.

[3] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning, 2017.

[4] Thomas Elsken, Benedikt Staffler, Jan Hendrik Metzen, and Frank Hutter. Meta-learning of neural architectures for few-shot learning, 2020.

[5] Mitchell Wortsman, Kiana Ehsani, Mohammad Rastegari, Ali Farhadi, and Roozbeh Mottaghi. Learning to learn how to learn: Self-adaptive visual navigation using meta-learning, 2019.

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

[7] Hanxiao Liu, Karen Simonyan, and Yiming Yang. Darts: Differentiable architecture search, 2019.

[8] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition, 2018.

[9] Alex Nichol, Joshua Achiam, and John Schulman. On first-order meta-learning algorithms, 2018.

[10] Vinyals, C. O., Blundell, T. Lillicrap, and Wierstra. Matching networks for one shot learning, 2016.

[11] Andrew M. Saxe, James L. McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks, 2014.

LNAS: Lottery Ticket Neural Architecture Search

Stanford CS231N Project: Theory and Reinforcement Learning

Sergio Charles
Department of Computer Science
Stanford University
sergioc1@stanford.edu

Gordon Chi
Department of Computer Science
Stanford University
gsychi@stanford.edu

Gil Kornberg
Department of Computer Science
Stanford University
gilk@stanford.edu

Mentors: Yao Liu & Rachel Gardner. Note: Sergio is sharing this project with CS229.

Abstract

Despite the mainstream of usage of neural networks in increasingly difficult tasks in computer vision and natural language processing tasks, optimal network architectures are traditionally attained after countless manual experiments and rounds of redundant training iterations. We propose a different approach that incorporates Neural Architecture Search (NAS) method [10] and the lottery ticket hypothesis [4] – for a randomly initialized neural network, there exists a sub-network architecture that trains and performs at least as efficiently as the super network. Our results show that both variants of this search – the first by pruning model parameters, and the second by pruning architecture parameters – return a model sub-network architecture that achieves state-of-the-art performance without expensive memory or time usage.

1. Introduction

In the past decade, neural networks have been implemented for increasingly difficult tasks in computer vision and natural language processing domains. Yet despite their flexibility and utility, optimal network architectures remain difficult to design; state-of-the-art models are the fruits of countless manual experiments and rounds of trial-and-error training iterations. To bridge this gap, Zoph and Le proposed the use of Neural architecture search (NAS) [10], a method that automatically learns network architectures and maximizes performance on a specific task in tandem. The most obvious drawback of this work: it uses an RNN meta-controller with proximal policy optimization which is too computationally expensive for most ML development tasks. In fact, the original NAS [10] used 450 GPUs for 3-4 days.

Consequently, there have been recent attempts to decrease the runtime of NAS. Efficient NAS [7] (ENAS) introduces the notion of searching for child models as subgraphs over a directed acyclic (DAG) super-graph. Local computation nodes have parameters that are shared among child models in the search space, which yields a 1000× improvement over NAS. NASNet [11] defines the architecture of a ConvNet as stacked modules of two types of cells: a *normal cell* which preserves input dimension and a *reduction cell* which halves output dimension by using a stride of 2. They show that a well-chosen cell yields desirable transfer learning on downstream tasks. The Differentiable architecture search algorithm (DARTS) improves upon both of these methods by making the search space over architectures continuous via a softmax over the set of candidate operations [6].

However, DARTS still has prohibitively expensive memory usage due to redundant candidate architecture paths. Hence, we propose a neural architecture search method based on the lottery ticket hypothesis [4] which states that for a randomly initialized neural network, there exists a sub-network architecture that has commensurate accuracy and trains at least as efficiently as the super-network. More specifically, we propose two variants of this search: the first variation conducts a neural architecture search before pruning model parameters during train time based on our lottery

ticket hypothesis; the second variation conducts a neural architecture search that prunes network architecture parameters before performing a standard training loop. We theorize that regardless of the variant chosen, by performing iterative or one-shot pruning when training the architecture and child models, the sparse sub-network neural architecture *winning ticket* can be found in at most as many iterations, and with at least the same accuracy, of the model architecture found by DARTS.

2. Related Work

Neural architecture search (NAS) was proposed in [10] as a method to automatically learn network architectures to maximize performance on a specific task. The original implementation of NAS serialized CNN and RNN network topology representations as a sequence of operations at each layer. That is, they surmised that the "structure and connectivity of a neural network", namely the architectural hyperparameters, can be encoded by a sequence of tokens. Thus, they train an RNN meta controller to produce the sequence $a_{1:T}$ of hyperparameters used to design the architecture, which can be construed as a sequence of actions in the sense of reinforcement learning [10]. The NAS search space is the set of operations that can be performed between layers of a neural network, which induces a strong prior on network design. The controller samples a candidate model called the child model, trains it to convergence, and evaluates the reward R, measured by the accuracy, of the child model on a held-out validation set. The controller uses the objective of maximizing expected reward as an update signal.

More precisely, the NAS search algorithm iterates over several possible child models in its search space and optimizes expected reward, e.g. model size, accuracy, or latency. The meta-controller RNN optimizes the following objective [10]:

$$J(\theta_C) = \mathbb{E}_{P(a_{1:T};\theta_C)}[R] \tag{1}$$

In the original NAS paper, they optimized the RNN controller parameters θ_C with REINFORCE [9] on the policy gradient [10]:

$$\nabla_{\theta_C} J(\theta_C) = \sum_{t=1}^{T} \mathbb{E}[\nabla_{\theta_C} \log P(a_t | a_{(t-1):1}; \theta_C) R]. \tag{2}$$

because R is non-differentiable.

Furthermore, we can represent a child model as a DAG with nodes as "local computations" and edges as the "flow of information" [7]. Efficient NAS (ENAS) [7] introduces the notion of searching for architectures as sub-graphs of the directed acyclic super-graph. That is, this DAG is represents a superposition of all possible child models. These local computation nodes have parameters and, thus, ENAS shares these parameters among child models in the search space, which yields a 1000x improvement over the original NAS method.

Differentiable Architecture Search (DARTS) [6] eschews the need for a proximal policy-based method. In DARTS, they learn a computation cell which can be stacked or recursively concatenated to create a convolutional neural network or recursive neural network, respectively. A computation cell can be represented as a DAG consisting of a topologically ordered set of n nodes. Let $x^{(i)}$ represent a latent representation and each edge (i, j) represent an operation $o^{(i,j)}$ on $x^{(i)}$. Using the topological ordering, we can compute the node based on the operations of preceding nodes [6]:

$$x^{(j)} = \sum_{i<j} o^{(i,j)}(x^{(i)}). \tag{3}$$

If $\mathcal{O}$ denotes a set of our *candidate operations*, DARTS makes the search space continuous by considering a softmax over possible operations:

$$\hat{o}^{(i,j)}(x) = \sum_{o \in \mathcal{O}} \frac{e^{\alpha_o^{(i,j)}}}{\sum_{o' \in \mathcal{O}} e^{\alpha_{o'}^{(i,j)}}} o(x) \tag{4}$$

where $\alpha^{(i,j)}$ are learned parameters between nodes i and j; finding these variables is equivalent to neural architecture search, which now allows us to employ a standard bi-level optimization. When we have found the neural architecture, we collapse its superposition state by discretizing the mixed operations $\hat{o}^{(i,j)}$ to $o^{(i,j)} = \arg\max_{o \in \mathcal{O}} \alpha_o^{(i,j)}$. Let $\mathcal{L}_{\text{train}}$ denote the training loss and let $\mathcal{L}_{\text{val}}$ denote the validation loss. We use a bi-level objective proposed in DARTS:

$$\begin{aligned} &\min_{\alpha} \mathcal{L}_{\text{val}}(\theta^*(\alpha), \alpha) \\ &\text{subject to } \theta^*(\alpha) = \arg\min_{\theta} \mathcal{L}_{\text{train}}(\theta, \alpha). \end{aligned} \tag{5}$$

which determines the optimal α^* to minimize validation loss $\mathcal{L}_{\text{val}}(\theta^*, \alpha^*)$, where the optimal model parameters θ^* is $\theta^* = \arg\min_{\theta} \mathcal{L}_{\text{train}}(\theta, \alpha^*)$. The inner optimization is very computationally expensive and, thus, DARTS uses an unrolled approximation of [6]:

$$\nabla_{\theta} \mathcal{L}_{\text{val}}(\theta^*(\alpha), \alpha) \approx \nabla_{\alpha} \mathcal{L}_{\text{val}}(\theta - \xi \nabla_{\theta}(\theta, \alpha), \alpha) \tag{6}$$

which approximates $\theta^*(\alpha)$ by only using a single step of SGD to avoid performing the inner optimization until convergence.

However, the memory requirements of DARTS are still cumbersome because GPU consumption is proportional to $|\mathcal{O}|$. Methods like MnasNet [8] employ optimization with a multi-objective of accuracy on a target task and inference

time latency. Likewise, ProxylessNAS, proposed by Cai et. al [1], redresses the problem of high memory consumption by using path-level pruning which binarizes the associated softmax probabilities of the real-valued architecture parameters α. They also present "hardware-aware neural network specialization", which optimizes the architecture for the specific hardware device [1].

We make use of the results of the lottery ticket hypothesis [4], which states that for a randomly initialized neural network, there exists a sub-network architecture that has at least as good test accuracy and trains at least as efficiently as the super-network. Such a winning ticket sub-network is identified and trained as follows [4]:

1. Initialize a neural network $\mathrm{f}_{\theta_0}(x)$ for parameters θ_0 distributed according to the training data $\theta \sim \mathcal{D}_{\text{train}}$.
2. Train the network for k iterations until we have parameters θ_j.
3. Prune $p\%$ of the parameters in the network, according to magnitude, forming a mask $\mathcal{M}$.
4. Reset the remaining parameters to their respective θ_0 values, which yields the winning ticket $\mathrm{f}_{\mathcal{M}\odot\theta_0}(x)$.

3. Methods

In light of the lottery ticket hypothesis [4], we posit that by performing iterative or one-shot pruning during neural architecture search with DARTS, the sparse sub-network neural architecture *winning ticket* can be found in *commensurate time* (at most as many iterations as the architecture found by search) and with *commensurate accuracy* (at least the same accuracy of the architecture found by search).

This is distinct from NAS methods that optimize on a multi-objective of model size and accuracy on a target task, as seen in [3], because these methods do not leverage the notion of a winning ticket from the lottery ticket hypothesis. That is, they are not searching for sparse child neural architectures, but instead small neural architectures; this would generally improve inference time latency but does not address the computational expensiveness of modern NAS.

We search for an architecture of a ConvNet as stacked computation cells, like in NASNet [11]. We use two types of cells: a *normal cell* which preserves input dimension and a *reduction cell* which halves output dimension by using a stride of 2. NASNet showed a well-chosen cell yields desirable transfer learning on downstream tasks. We use DARTS [6] as the NAS method, which eschews the meta-controller configuration. Operations are limited to convolutions, average pooling, max pooling, and the zero operation.

Following the notation of ProxylessNAS [1], let $\mathcal{N}(e_1, \ldots, e_n)$ denote the over-parameterized convolutional neural network, where e_i is an edge in its DAG representation of a topologically ordered set of its n nodes. Let $x^{(i)}$ represent a latent representation and each edge $e_{i,j}$ represents the operation $o^{(i,j)}$ on $x^{(i)}$. For our N *candidate operations* $\mathcal{O}$, we make the search space continuous by defining a mixed operation as a softmax over possible operations [6]: $\hat{o}^{(i,j)}(x) = \sum_{o\in\mathcal{O}} \frac{e^{\alpha_o^{(i,j)}}}{\sum_{o'\in\mathcal{O}} e^{\alpha_{o'}^{(i,j)}}} o(x)$ where $\alpha^{(i,j)}$ are learned architecture parameters between node i and node j. Since each edge is a mixed operation, i.e. a linear combination of the $N = |\mathcal{O}|$ candidate operations $\hat{o}(x) = \sum_{i=1}^{N} p_i o_i(x)$, we can intuitively represent a mixed operation by N paths where each path is weighted with probability p_i. When performing the neural ticket architecture search to find the winning neural ticket α^*, we apply an architecture-dependent mask $\mathcal{M}_\alpha$:

$$\hat{o}(x) = \mathbb{E}_{o\sim\mathcal{O}\odot\mathcal{M}_\alpha}[o(x)]. \tag{7}$$

to prune the architecture parameters α. When we retrain the model f_θ with optimal architecture α^*, we use a model-dependent mask $\mathcal{M}_\theta$ to prune the parameters θ of the model f_θ. Since we effectively hard-prune the choice of candidate operations when selecting the architecture with highest candidate probability, this will induce significant regressions in performance. Therefore, it is important to retrain the model f_θ with the sparse architecture. Let $\mathcal{L}_{\text{train}}$ and $\mathcal{L}_{\text{val}}$ and denote the training and validation loss, respectively. We use a bi-level objective proposed in DARTS:

$$\begin{aligned} &\min_{\alpha} \mathcal{L}_{\text{val}}(\theta^*(\alpha), \alpha) \\ &\text{subject to } \theta^*(\alpha) = \arg\min_{\theta} \mathcal{L}_{\text{train}}(\theta, \alpha). \end{aligned} \tag{8}$$

Iterative pruning is performed to identify the *neural winning ticket*. In the generalized method, the cell architecture is trained in tandem with the pruning program of the lottery ticket hypothesis until early stopping, and accuracy is measured on a held-out validation set. Then we retrain the model with the architecture found during the search using a similar pruning program. We developed two approaches to performing the lottery ticket pruning, which are as follows:

Variant 1: We perform DARTS neural architecture search to find the optimal architecture α^*. Then, when retraining the model f_θ with architecture α^*, with a sparsity of $p \in (0, 1)$, we prune $p^{1/n}$ of the lowest magnitude parame-

ters for n rounds. The algorithm is enumerated as follows:

Algorithm 1: Variant 1 of LNAS

Randomly initialize $\theta_0 \sim \mathcal{D}_{\text{train}}, \alpha_0 \sim \mathcal{D}_{\text{val}}$;
Search for optimal architecture α^*
while *not converged* **do**
  1. Perform the α gradient update $\alpha \leftarrow \alpha - \eta \nabla_\alpha \mathcal{L}_{\text{val}}(\theta - \xi \nabla_\theta \mathcal{L}_{\text{train}}(\theta, \alpha), \alpha)$ for k iterations;
  2. Perform the model θ gradient update $\theta \leftarrow \theta - \xi \nabla_\theta \mathcal{L}_{\text{train}}(\theta, \alpha)$ for k iterations;
end
Re-train model parameters θ of optimal architecture α^* to find (α^*, θ^*)
for n *pruning iterations* **do**
  1. Perform the θ gradient update $\theta \leftarrow \theta - \xi \nabla_\theta \mathcal{L}_{\text{train}}(\theta, \alpha)$ for k iterations;
  2. Prune $p^{1/n}$ of smallest magnitude model parameters by applying mask: $\theta \odot \mathcal{M}_\theta$;
end
return *winning neural ticket* (α^*, θ^*)

For fixed θ, we optimize the architecture parameters α with SGD on the validation set, using the unrolled approximation of Equation 6. Then during retraining, we optimize the model parameters θ with SGD on the training set until convergence, while pruning $p^{1/n}$ of model parameters every k iterations of SGD for a total of n rounds of training.

Variant 2: We perform path-level pruning, in the sense of ProxylessNAS [1], whereby with a sparsity of $p \in (0, 1)$, we prune $p^{1/n}$ of the lowest-magnitude architecture parameters α during the neural architecture search. This corresponds to pruning candidate operations or, equivalently, paths for each edge of the computational graph. As before, during this step, we use the unrolled approximation in Equation 8. Then we re-train the model f_θ, with the optimal architecture α^* found during the previous step, via SGD until convergence. The algorithm is as follows:

Algorithm 2: Variant 2 of LNAS

Randomly initialize $\theta_0 \sim \mathcal{D}_{\text{train}}, \alpha_0 \sim \mathcal{D}_{\text{val}}$;
Search for optimal architecture α^*
for n *pruning iterations* **do**
  1. Perform the α gradient update $\alpha \leftarrow \alpha - \eta \nabla_\alpha \mathcal{L}_{\text{val}}(\theta - \xi \nabla_\theta \mathcal{L}_{\text{train}}(\theta, \alpha), \alpha)$ for k iterations;
  2. Perform the model θ gradient update $\theta \leftarrow \theta - \xi \nabla_\theta \mathcal{L}_{\text{train}}(\theta, \alpha)$ for k iterations;
  3. Prune $p^{1/n}$ of smallest magnitude architecture parameters by applying mask: $\alpha \odot \mathcal{M}_\alpha$.
end
Re-train model parameters θ of optimal architecture α^* to find optimal (α^*, θ^*)
while *not converged* **do**
  1. Perform the θ gradient update $\theta \leftarrow \theta - \xi \nabla_\theta \mathcal{L}_{\text{train}}(\theta, \alpha)$;
end
return *winning neural ticket* (α, θ)

Although we did not complete a full grid search of pruning both model parameters and architecture parameters at various sparsity levels, to determine the relative utility of each variant in a wider scope, it is useful to consider the more generalized version of our algorithm:

Algorithm 3: Generalized LNAS

Randomly initialize $\theta_0 \sim \mathcal{D}_{\text{train}}, \alpha_0 \sim \mathcal{D}_{\text{val}}$;
Search for optimal architecture α^*
for n *pruning iterations* **do**
  1. Perform the α gradient update $\alpha \leftarrow \alpha - \eta \nabla_\alpha \mathcal{L}_{\text{val}}(\theta - \xi \nabla_\theta \mathcal{L}_{\text{train}}(\theta, \alpha), \alpha)$ for k iterations;
  2. Perform the model θ gradient update $\theta \leftarrow \theta - \xi \nabla_\theta \mathcal{L}_{\text{train}}(\theta, \alpha)$ for k iterations;
  3. Prune $p^{1/n}$ of smallest magnitude architecture parameters by applying mask: $\alpha \odot \mathcal{M}_\alpha$.
end
Re-train model parameters θ of optimal architecture α^* to find optimal (θ^*, α^*)
for n *pruning iterations* **do**
  1. Perform the θ gradient update $\theta \leftarrow \theta - \xi \nabla_\theta \mathcal{L}_{\text{train}}(\theta, \alpha)$ for k iterations;
  2. Prune $p^{1/n}$ of smallest magnitude model parameters by applying mask: $\theta \odot \mathcal{M}_\theta$.
end
return *winning neural ticket* (α, θ)

That is, we perform k SGD update steps for the DARTS neural architecture search, using the unrolled approximation in Equation 6. Then, with a sparsity of $p \in (0, 1)$, we prune $p^{1/n}$ of of the smallest magnitude architecture pa-

rameters, which is a form of path-level pruning of the N possible operation candidate paths [1]. We do this for n rounds until we have found the optimal neural architecture α^*. Next, we optimize the model parameters θ with SGD on the training set for k steps. Then we prune $p^{1/n}$ of the smallest magnitude model parameters. We repeat for n rounds to converge to an optimal θ^* and α^*, which is the theorized winning neural ticket.

4. Data

For all experiments, we train and validate all experiments on the CIFAR-10 dataset [5], which consists of 60,000 32×32 colored images of 10 different classes. We utilize a 50,000 to 10,000 *train-test* split of the images; all final models were validated on Top-1 accuracy performance on the test dataset. No pre-processing or filtering was applied prior to the training of our networks.

5. Experiments

5.1. Lottery Ticket Neural Architecture Search

For each variant, we train models on different sparsity levels and evaluate their relative performance on the CIFAR-10 dataset to a baseline model constructed via differential architectural search (DARTS). Our goal is determine how the number of parameters remaining affects model accuracy and training time of the optimal *neural winning ticket*.

Baseline DARTS Model: We reimplemented the DARTS search on the CIFAR-10 dataset using the `nni` library as proposed in the original paper [6]. Our search space $\mathcal{O}$ of candidate operations consisted of 3×3 and 5×5 separable convolutions, 3×3 and 5×5 dilated separable convolutions, 3×3 max and average pooling, identity mapping, and a zero mapping. All operations were stride one and padded to preserve spatial resolution. The convolutional cell consisted of 7 nodes, and was placed after a ReLU activation block and before batch normalization.

Our neural network was optimized with respect to cross-entropy loss, a metric known for its strong performance in classification tasks. We perform 10 epochs of neural architecture search on the $\mathcal{O}$ search space described above, followed by 20 epochs of a standard training loop. Both the search and train loops utilized a stochastic gradient descent (SGD) + Momentum optimizer with 0.9 momentum and 3.0×10^{-4} weight decay; the initial learning rate of 0.025 was adjusted according to a cosine annealing scheduler. Batch size was set at 64.

Initial Architectural Search: For **Variant 1**, observe that all experiments focus only on pruning model parameters of the baseline models given different sparsity levels. Hence, our initial model architecture is identical to the baseline model, where we performed 10 epochs of neural architecture search on the $\mathcal{O}$ search space consisting of 3×3 and 5×5 separable convolutions, 3×3 and 5×5 dilated separable convolutions, 3×3 max and average pooling, identity mapping, and a zero mapping. Learning rates and optimizers were also kept constant.

For **Variant 2**, we performed a redesigned DARTS search on the CIFAR-10 dataset that iteratively prunes architecture parameters after each epoch of our neural architecture search, in conjuction with lottery ticket hypothesis. Otherwise, all other hyperparameters of the search were kept constant with the baseline model; our search space $\mathcal{O}$ of candidate operations again consisted only of 3×3 and 5×5 separable convolutions, 3×3 and 5×5 dilated separable convolutions, 3×3 max and average pooling, identity mapping, and a zero mapping. Likewise, all operations are stride one and padded to preserve spatial resolution; the convolutional cell consisted of 7 nodes, and was placed after a ReLU activation block and before batch normalization. Learning rates and optimizers were also kept constant.

Training Loop: For **Variant 1** and a given sparsity level p, we run a modified training loop for 20 epochs, where on the 10th epoch we perform one-shot pruning that zeroes out p parameters of the model, before re-initializing the model parameters to θ_0. For **Variant 2** and a given sparsity level p, a standard training loop for 20 epochs is implemented; there is no need to prune out parameters during train-time since the initial neural-architecture-search leaves us with a sufficiently pruned neural architecture.

Models trained from both variants were optimized with respect to cross-entropy loss, and all other hyperparameters were held constant. Both The training loop utilized a stochastic gradient descent (SGD) + Momentum optimizer with 0.9 momentum and 3.0×10^{-4} weight decay; the initial learning rate of 0.025 was adjusted according to a cosine annealing scheduler. Batch size was set at 64.

Sparsity Levels: For each variant, we experimented with 8 different sparsity levels: 0.125, 0.25, 0.375, 0.5, 0.675, 0.75, 0.875, 0.95. This leaves us with 16 trained models alongside our baseline model for analysis.

Evaluation Metrics: We evaluate our models on three metrics:

- First, we evaluate the prediction accuracy of the models across sparsity levels p and both variants, in order to determine which variant and sparsity levels leave us with a strong-performing model.

- Second, we evaluate test inference time by reporting the floating point operations per second (FLOPS) required; although the relative speed of models may be apparent given the chosen sparsity levels, this analysis presents with a better estimate of runtime.
- Finally, we evaluate train inference time by reporting the total number of hours required for the 10 epochs of neural architecture search and 20 epochs of the concurrent training loop. While we had attempted to keep all hyperparameters regarding batch size, learning rate and number of epochs constant across both variants, it is important to acknowledge that the lottery ticket hypothesis performs fundamentally different operations in the two variants, which may result in drastically different training times.

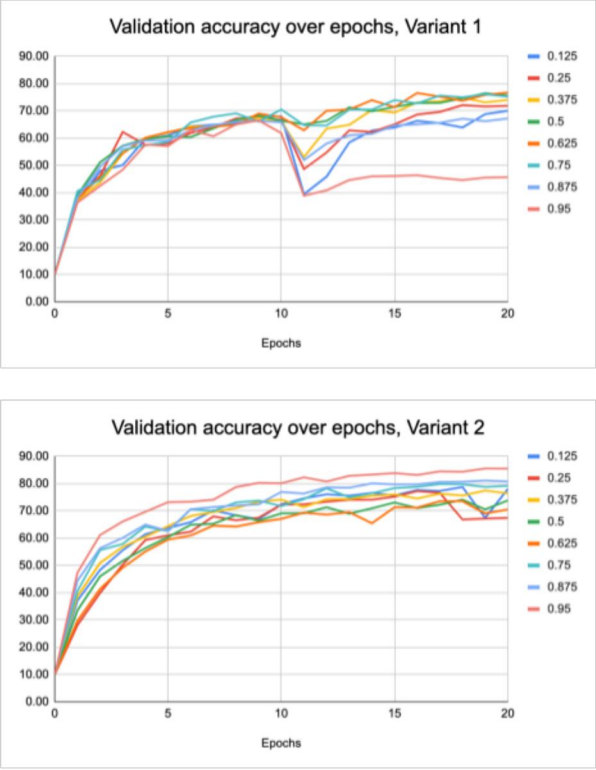

Figure 1. A visualization of model validation accuracy over epochs for both variants. Note that Variant 1 is less robust to sparsity levels; the sparsity levels of 0.95 is unable to reach past 50% accuracy in the aftermath of the one-shot pruning. Meanwhile, Variant 2 performs particularly well with the sparsity level of 0.95, reaching a validation accuracy of near 86.00% after the twenty epochs.

5.2. Prediction Accuracy and FLOPS

Overall, although not all sparsity levels and variant choices were able to obtain the winning lottery ticket after 20 epochs, many of our models returned comparable accuracy on the validation dataset. For Variant 1, the LNAS performed best on medium to high sparsity levels, with the models pruned on 0.5, 0.625, and 0.75 sparsity levels returning a validation accuracy of over 76% compared to just an average of around 70% for models trained on higher or lower sparsity levels. An opposite trend is observed in Variant 2, where the LNAS performed stronger overall and best on high and low sparsity levels. Surprisingly, we were able to obtain the three *winning tickets* under Variant 2, with the sparsity level of 0.95 obtaining a state-of-the-art validation accuracy of 85.6% in 20 epochs. This outperforms even the super baseline model we constructed with DARTS. More specific results across all the sparsity levels can be seen in Table 1 below.

We also present prediction accuracy on the validation set over time in Figure 1. Due to the pruning and re-initializing of weights to θ_0 in Variant 1, the drop in accuracy following Epoch 10 may result in models with a given sparsity level p being unable to attain high validation accuracy in the successive train steps. More importantly, our experiment suggests that if a model's validation accuracy is immediately robust to the re-initializing of weights, then it is likely that the given sparsity level will be closer to a winning lottery ticket.

In contrast, the validation accuracy curves for Variant 2 do not have large jumps in performance, resembling a standard training loop.

5.3. Variant Train Time Comparison

Though all experiments were run with the same number of epochs in both the search and train loops, Variant 1 has the lottery ticket hypothesis implemented in only the training loop while Variant 2 has the same hypothesis implemented in the search loop. This results in large differences in train time: as outlined in the Methods section, the neural architecture search requires a longer runtime when implemented alongside the lottery ticket hypothesis while the training loop only requires a one-shot pruning of model parameters over 20 epochs.

Hence, with all experiments done on a Tesla K80 GPU, the initial neural architecture search took around 2 hours to run for Variant 1 compared to about 5.2 hours for Variant 2; differences in training time between the two variants were negligible.

5.4. Discussion

The analysis on the differences in train time across variants (Section 5.3) suggests that the relative validation accuracy sparsity levels cannot be compared across variants, since one approach has the advantage of having a relatively longer train time. However, general discussions on sparsity levels and training robustness can still give us a strong

		Validation Accuracy (%)		
Variant	**Sparsity Level**	**Epoch 10**	**Epoch 20**	**FLOPS**
Baseline	N/A	74.04	78.90	752.355M
1	0.125	68.03	70.05	549.204M
(model parameter pruning)	0.25	67.50	72.13	473.034M
	0.375	67.43	74.02	391.527M
	0.5	68.32	76.52	315.356M
	0.625	69.03	76.74	233.849M
	0.75	70.54	76.15	157.678M
	0.875	66.32	67.23	76.171M
	0.95	66.58	46.43	84.050M
2	0.125	72.45	78.77	549.204M
(model architecture pruning)	0.25	72.28	78.04	473.034M
	0.375	74.13	77.47	391.527M
	0.5	69.11	74.12	315.356M
	0.625	67.12	73.60	233.849M
	0.75	71.80	**79.86**	157.678M
	0.875	79.86	**81.15**	76.171M
	0.95	80.14	**85.59**	84.050M

Table 1. Test inference time and validation accuracy across variants and multiple sparsity levels for both variants, compared alongside the baseline model. Winning tickets are shown in **bold**. The result reported for validation accuracy at epochs 10 and 20 were based off of highest validation accuracy over the course of training up until the stated epoch. For Variant 1, 20-epoch performance was consistently strongest on sparsity levels between 0.375-0.75, with performance teetering off at the higher sparsity levels. Variant 2 observes an opposite trend, with performance strongest at low and high sparsity levels.

indication on which variant performs better; there is sufficient evidence to hypothesize how the proposed Algorithm 3 (generalized LNAS) would perform on the CIFAR-10 dataset.

Overall, Variant 1 is more sensitive to sparsity level. We theorize that this is because the initial neural architecture search is done just once and is focused on searching for a general model with a certain range of parameters; as a result, only mid to high sparsity levels such as 0.5, 0.625 and 0.75 were close to reaching the lottery ticket hypothesis. This hypothesis is supported by the sharp drop-off once the sparsity level hits 0.875 and 0.95, when the validation accuracy never eclipses the validation accuracy obtained prior to the pruning and re-initializing of weights. In contrast, Variant 2 does not suffer from the same issue, because the architecture outputted from our lottery ticket hypothesis is optimized on performing well with a lower number of parameters.

With this in mind, it is most likely that the generalized LNAS will perform better than variants 1 and 2 individually. By pruning network architecture parameters (Variant 2) before pruning model architecture parameters (Variant 1), our concern with the sensitivity to sparsity level is mitigated; the model architecture may overall be better suited for the pruning process.

6. Conclusion

Our work shows that with the implementation of the lottery ticket hypothesis alongside a differential architectural search (DARTS) algorithm, we are able to obtain sub-network architecture that achieve state-of-the-art performance without expensive memory or time usage. Furthermore, we show that both parts of a generalized LNAS algorithm – the first part of pruning model parameters, and the second part of pruning architecture parameters – have strong performances in isolation and may be combined in future work to create an even better sub-model.

While this work is a good start in the field of neural network optimization, our future experiments aim to further improve upon training efficiency. We aim to explore the intersectionality of DARTS in other machine learning subfields. We hope that by implementing algorithms such as MetaNAS [2] that combine gradient-based neural architecture search (NAS) methods with gradient-based meta-learning methods, we will be able to solve more challenging problems with just a fraction of the computational and time-related resources required when compared to more traditional approaches.

References

[1] H. Cai, L. Zhu, and S. Han. Proxylessnas: Direct neural architecture search on target task and hardware, 2019.

[2] T. Elsken, B. Staffler, J. H. Metzen, and F. Hutter. Meta-learning of neural architectures for few-shot learning, 2020.

[3] I. Fedorov, R. P. Adams, M. Mattina, and P. N. Whatmough. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers, 2019.

[4] J. Frankle and M. Carbin. The lottery ticket hypothesis: Finding sparse, trainable neural networks, 2019.

[5] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-10 (canadian institute for advanced research).

[6] H. Liu, K. Simonyan, and Y. Yang. Darts: Differentiable architecture search, 2019.

[7] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean. Efficient neural architecture search via parameter sharing, 2018.

[8] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile, 2019.

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

[10] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning, 2017.

[11] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition, 2018.