# 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].

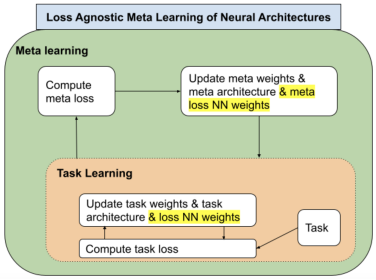


Figure 1: An overview of the LAMANAS algorithm.

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 $\theta$ 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 $\theta$ on the task validation set $\mathcal{D}_{\text{val}}^{\mathcal{T}_i}$.

Given model parameter initializations $\theta$, 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 $\theta^*$ 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 $\theta$ 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 $\alpha_{\text{meta}}$ in tandem with meta-model parameters $\theta_{\text{meta}}$ during meta-training. The meta-parameters $\alpha_{\text{meta}}$ and $\theta_{\text{meta}}$ 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 $\mathcal{T}_i$, $\Phi_K$ is a $K$-step task learner algorithm like SGD, and $(\theta, \alpha)$ 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 $\theta$ and $\alpha$. Methods like DARTS hold $\alpha$ constant so we optimize the task model parameters $\theta_i$ 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].

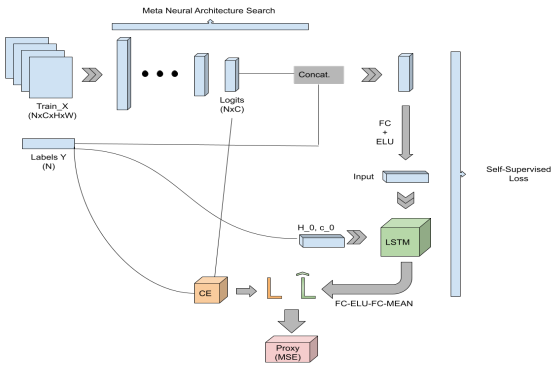


Figure 2: Details of the LAMANAS algorithm.

In constrast to these methods, we construct a $K$-step task learner $\Phi_K$ that, for task $\mathcal{T}_i$, optimizes both the task model parameters $\theta_i$ and the task architecture parameters $\alpha_i$ with model learning rate $\lambda_{\text{task}}$ and architecture learning rate $\xi_{\text{task}}$, 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 $\theta_i^*$ and $\alpha_i^*$:

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

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 $\theta_i^*$ and $\alpha_i^*$ 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 $\Phi_K$ 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 $\theta$ and $\alpha$, 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 $\phi$. 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 $\phi$ 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 $\phi$, 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 $\theta$, $\alpha$, $\phi$
  **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 $\theta$, $\alpha$, $\phi$
  **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 $\theta_i^*$ and $\alpha_i^*$
        $\theta_i \leftarrow \theta_i - \lambda_{\text{task}} \nabla_\theta \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi)$
        $\alpha_i \leftarrow \alpha_i - \xi_{\text{task}} \nabla_\alpha \mathcal{L}_{\text{train}}^{\mathcal{T}_i}(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}; \phi)$
      **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 cross entropy loss $\mathcal{L}_{\text{val}}^{\mathcal{T}_i}(\theta_i, \alpha_i, \mathcal{D}_{\text{train}}^{\mathcal{T}_i}) = \mathcal{H}(z_i, y_{\text{train}}^{\mathcal{T}_i})$ 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:

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

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

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

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.

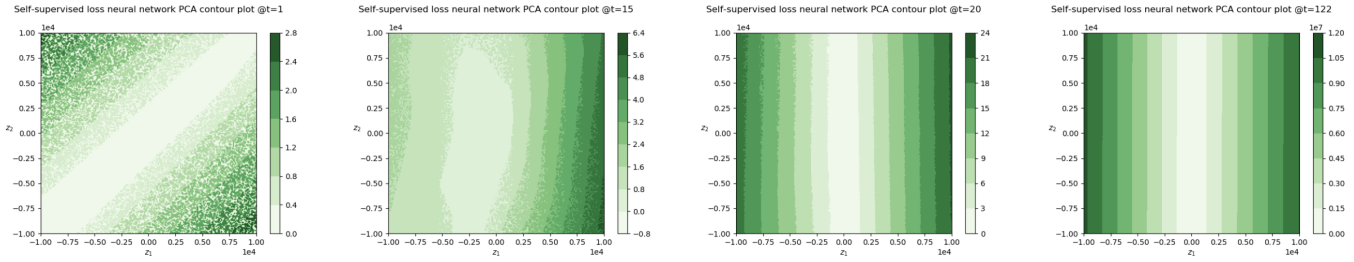


Figure 3: Snapshots of the geometry of self-supervised loss FNN function for the self-supervised loss network over hand-selected meta-epochs 11, 20, 111 and 122. The loss neural network is an FNN with residual connections and MSE loss proxy. *At $t = 1$ epochs, the loss is almost equivalent to cross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