Faster Discovery of Neural Architectures by Searching for Paths in a Large Model

Anonymous authors
Paper under double-blind review

Abstract

We propose an approach for automatic model designing, which is significantly faster and less expensive than previous methods. In our method, which we name *Efficient Neural Architecture Search* (ENAS), a controller learns to discover neural architectures by searching for an optimal path within a larger predetermined model. The parameters of the predetermined model are trained to minimize a canonical loss function, such as the cross entropy, on the training dataset. The controller learns the path with policy gradient to maximize the expected reward on the validation set. In our experiments, ENAS achieves comparable test accuracy while being 10x faster and requiring 100x less resources than NAS. On the CIFAR-10 dataset, ENAS can design novel architectures that achieve the test error of 3.86%, compared to 3.41% by standard NAS (Zoph et al., 2017). On the Penn Treebank dataset, ENAS also discovers a novel architecture, which achieves the test perplexity of 64.6 compared to 62.4 by standard NAS.

1 Introduction

Neural architecture search (NAS) has recently been used successfully to design model architectures for a number of tasks, especially in the areas of image classification and language modeling (Zoph and Le, 2017; Zoph et al., 2017). Despite its success, NAS is computationally expensive and time consuming. For instance, Zoph et al. (2017) use 450 GPUs and train for 3-4 days. On the other hand, using less computing resources often leads to less compelling results (Negrinho and Gordon, 2017; Baker et al., 2017).

In this work, we propose a method to accelerate NAS. First, we observe that it is possible to share parameters among all child models in a particular search space. This observation effectively avoids having to train from scratch each new architecture found during the search process, leading to a faster search time. Though it is appealing, sharing parameters among different architectures can lead to very noisy updates to the parameters, as an update to one architecture may carry negative effects to other architectures. To address this problem, we determine a large model ahead of time, with a careful weight sharing scheme that allows our method to generate a discrete mask within this model, which we describe in Section 4. We then interleave two processes: (1) training the shared parameters to minimize a canonical loss function, such as the cross entropy, on the training data; and (2) searching for good architectures, which are represented by discrete sequences of tokens, using policy gradient on the validation set. This method can quickly discover novel architectures whilst using far less computing resources. Because of this efficiency, we name our method *Efficient Neural Architecture Search* (ENAS).

We give a theoretical justification of the convergence guarantee of ENAS, and then demonstrate empirically its effectiveness on two tasks: image classification on CIFAR-10 and language modeling on Penn Treebank. For both tasks, ENAS is able to discover novel neural architectures that achieve comparable accuracies to best reported results by NAS methods. Specifically, ENAS achieves 3.86% error rate on CIFAR-10 and 64.6 perplexity on Penn Treebank. In comparison, the best results reported by previous NAS approaches are 3.41% error rate on CIFAR-10 and 62.4 perplexity on Penn Treebank. More importantly, ENAS is significantly faster and less computationally intensive than previous methods. In our ex-

periments, ENAS takes less than 15 hours to train, running on a single Nvidia GTX 1080Ti GPU, meaning that ENAS uses approximately 100x less computational resources and is approximately 10x faster than NAS to reach 3.86% test error on CIFAR-10.

2 Neural Architecture Search with Reinforcement Learning

The goal of NAS is to find a model architecture $\mathbf{m}$ that achieves a high reward $R(\mathbf{m})$ on a particular task. This setting is general: $R(\mathbf{m})$ can be the accuracy for an image classifier, or the perplexity of a language model. Recent approaches such as Zoph and Le (2017); Zoph et al. (2017); Baker et al. (2017); Bello et al. (2017); Zhong et al. (2017) used reinforcement learning to search the combinatorial space of architectures, attempting to model a parameterized policy $\pi(\mathbf{m};\theta)$ of $\mathbf{m}$ to maximize the expected reward

$$J(\theta) = \mathbb{E}_{\mathbf{m}\sim\pi(\mathbf{m};\theta)}\left[R(\mathbf{m})\right] \tag{1}$$

The key differences between these NAS approaches are in the design choices for the search space, the parametrization of $\pi(\mathbf{m},\theta)$, and the learning algorithm for optimizing $J(\theta)$.

There are two main approaches for designing the search space. The first approach designs the whole architecture all at once. For instance, if we were to design a convolutional network, we would choose how many filters to use at each layer, the filter sizes, how to set up the skip connections, etc. We call this approach *macro search space design*. Macro design is used by Zoph and Le (2017); Baker et al. (2017); Negrinho and Gordon (2017); Brock et al. (2017); Saxena and Verbeek (2016). The second approach, which we call *micro search space design*, aims to search for smaller modules which are then replicated and assembled to form the final model. Micro design is used by Zoph et al. (2017); Zhong et al. (2017). We note that Zoph and Le (2017) also use micro design when designing the RNN cell.

An architecture $\mathbf{m}$ is often represented as a sequence, with a grammar and meaning specified by the NAS algorithm. Because modeling variable length sequences is necessary, the reinforcement learning policy $\pi(\mathbf{m};\theta)$ is often parametrized with a recurrent neural network. $J(\theta)$ is then learned with various reinforcement learning algorithms. For instance, Zoph and Le (2017) use REINFORCE (Williams, 1992) with a moving average baseline and Zoph et al. (2017) use Proximal Policy Optimization (Schulman et al., 2017) to learn $J(\theta)$. Deep Q-learning approaches have also been explored (Baker et al., 2017; Zhong et al., 2017).

3 The Source of the Computational Expense

NAS methods are typically time consuming and computationally expensive. Most significantly, Zoph and Le (2017) and Zoph et al. (2017) respectively train in 3-4 weeks and 3-4 days, using 800 and 450 GPUs concurrently at any time during their training. Zhong et al. (2017) use 32 GPUs, training in 3 days. This amount of computing resource, albeit reduced compared to the first two studies, is still not widely available. Baker et al. (2017) use 10 GPUs, training in 5 days, but the architectures found on the task of CIFAR-10 do not yield as compelling results, likely because the method did not have an opportunity to explore enough architectures.

The computational expense of NAS methods is due to the fact that they train each child model $\mathbf{m}$ from scratch. To see this, note that in Eqn. 1, the reward $R(\mathbf{m})$ cannot be computed without the optimal parameters $\omega^*_{\mathbf{m}}$ of the model $\mathbf{m}$. To find $\omega^*_{\mathbf{m}}$, we need to train $\mathbf{m}$ to convergence. Due to the high sample complexity of reinforcement learning, many models $\mathbf{m}$ have to be sampled, trained, and evaluated. We posit that this is the main weakness of NAS approaches: after training each model $\mathbf{m}$, only the reward $R(\mathbf{m};\omega^*_{\mathbf{m}})$ is kept, while the learned parameters in $\omega^*_{\mathbf{m}}$, which could be reused, are thrown away.

Several methods have been proposed to avoid training each architecture $\mathbf{m}$ from scratch, such as convolutional neural fabrics (ConvFabrics) (Saxena and Verbeek, 2016) and SMASH (Brock et al., 2017). These methods are often more computationally efficient. However, the tradeoff in ConvFabrics is that their search space is not flexible enough to include novel architectures, e.g., architectures with arbitrary skip connection patterns as

in Zoph and Le (2017). In contrast, SMASH can design many interesting architectures, but requires a hypernetwork (Ha et al., 2017) to generate the weights $\omega_{\mathbf{m}}$, conditional on the architecture $\mathbf{m}$. While a hypernetwork can efficiently *rank* different architectures, as shown in the paper, the real performance of each network, i.e., $R(\mathbf{m}, \omega^*_{\mathbf{m}})$, is different from its performance with $\omega_{\mathbf{m}}$ generated by a hypernetwork. Such discrepancy can cause misleading signals, which may make SMASH unsuitable for reinforcement learning.

In this work, we propose a novel approach that combines the strengths of both ConvFabrics and SMASH. Unlike SMASH, we do not use a hypernetwork to generate $\omega_{\mathbf{m}}$ for each $\mathbf{m}$, but instead keep a shared set of parameters ω among all architectures $\mathbf{m}$. Unlike ConvFabrics, we design a different mechanism to specify computing paths in our model, leading to a very flexible search space. In fact, our search space is larger than the search space of SMASH as well as the search space described in Zoph and Le (2017). As a result, our method can efficiently discover novel architectures that achieve better accuracy than SMASH.

4 Efficient Neural Architecture Search

4.1 Parameters Sharing in Convolutional Models

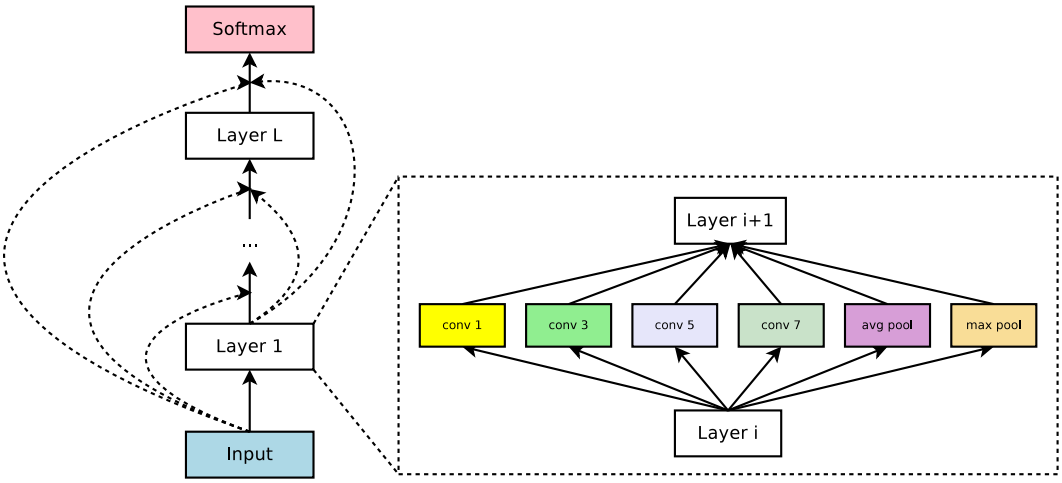

Figure 1: Our search space for convolutional models where all models can share the parameters. Left: the dotted arrows denote the skip connections. Right: At each layer, there are 6 distinct branches, i.e. computational paths.

We propose a scheme to share parameters among different convolutional architectures, which is illustrated in Figure 1. Our method requires a pre-specified number of layers, denoted by L, in all models in our search space.[1] Suppose that at each layer, we have to make a decision of which of B operations to perform. In this paper, we use $B = 6$ operations at each layer: four convolutions with square filters of sizes 1, 3, 5, 7, an average pooling, and a max pooling of filter size 3×3. All these operations have the spatial strides 1×1 and their outputs are centrally padded to preserve the spatial dimensions. Each of these operations has a maximum number of output channels C, set to 256. Parameters are independent to each branch in each layer. We denote by ω the set of all shared parameters in the model.

During each data pass, each of the $L \times B \times C$ channels in the network is controlled by a binary mask, which, if turned off, will remove the corresponding channel from the network and if turned on, will allow information to flow through that channel. These binary masks are specified by a controller network, which we will describe in Section 4.3. Since the number of channels can be large, we group them into blocks of S channels, where S evenly divides C. Consequentially, for each channel, we only predict C/S binary masks.

[1] While this requirement could be relaxed, e.g., by introducing stochastic auxiliary heads (Szegedy et al., 2016), we find that such stochastic heads make our training unstable.

In order to allow architectures with arbitrary patterns of skip connections (He et al., 2016; Zoph and Le, 2017), we insert skip connections between every pair of layers in the network, which the controller can also turn on or turn off. Therefore, each configuration of the channels and the skip connections realizes an architecture in our search space. The resulting search space has $2^{LBC/S+L(L-1)/2}$ configurations.

Structure of Convolutional Layers. Each convolutional operation in our method is followed by a batch normalization (Ioffe and Szegedy, 2015) and then a ReLU layer. We find the alternate setting of batch norm-conv-ReLU (Zoph et al., 2017) to have worse results.

Stabilizing Stochastic Skip Connections. If a layer receives skip connections from multiple layers before it, then these layers' outputs are concatenated in their depth dimension, and then a convolution of filter size 1×1 (followed by a batch normalization layer and a ReLU layer) is performed to ensure the number of output channels is still equal to C. This is necessary because if we were to follow Zoph and Le (2017) in concatenating the skip connected layers, that would result in $\frac{\ell(\ell-1)}{2}$ channels at layer ℓ^{th}, leading to a prohibitively large number of parameters. Using 1×1 convolutions to keep the number of channels at C is more economical.

Global Average Pooling. After the final convolutional layer, we average all the activations of each channel and then pass them to the Softmax layer. This trick was introduced by Lin et al. (2013), with the purpose of reducing the number of parameters in the dense connection to the Softmax layer to avoid overfitting. This trick is important to ENAS, because in ENAS, each configuration of the lower layers can lead to radically different outputs in the final layer, especially in the early training steps. If the Softmax head overfits to any of these configurations, subsequent updates made to model parameters ω become unstable, e.g. parameters in the Softmax layer have much larger gradients than parameters in other layers.

As we shall see in Section 4.4, our training algorithm can lead to noisy updates on ω. These three aforementioned tricks are thus crucial to our scheme of sharing parameters among different convolutional models. We carry out ablation studies which show that removing any of the tricks significantly de-stablizes our ENAS training process and substantially reduces the accuracy of the models found.

4.2 Parameters Sharing in Recurrent Models

For RNNs, NAS is used to design the architecture for each cell; this cell is then replicated at each time step and at each layer (Zoph and Le, 2017). Rather than searching for a completely novel cell architecture like Zoph and Le (2017), we start with the recurrent highway network (RHN), a recurrent cell architecture that not only allows gradients to propagate asympotically without loss but also allows the cell to model more complex transformations (Zilly et al., 2017). Each RHN cell resembles a dense layer in a feed forward network, augmented with a linear carousel. In He et al. (2016), the authors found that skip connections can improve the gradient flows in deep networks, leading to better performance. We hypothesize whether RHN can also benefit from having skip connections among their highway layers. We thus use ENAS to learn the skip connections pattern in the highway layers. Specifically, if layer ℓ^{th} receives the skip connections from layers $i_1 < i_2 < ... < i_k$, then the output $\mathbf{s}_\ell$ is computed as follows

$$\mathbf{h}_\ell \leftarrow \tanh\left(\mathbf{s}_{\ell-1} \cdot \mathbf{W}_\ell^{(\mathbf{h})}\right); \mathbf{t}_\ell \leftarrow \text{sigmoid}\left(\sum_{j=1}^{k} \mathbf{s}_{i_j} \cdot \mathbf{W}_{\ell,i_j}^{(\mathbf{t})}\right) \tag{2}$$

$$\mathbf{s}_\ell \leftarrow \mathbf{t}_\ell \otimes \mathbf{h}_\ell + (1 - \mathbf{t}_\ell) \otimes \mathbf{s}_{\ell-1}, \tag{3}$$

where $\otimes$ denotes the elementwise product and $\mathbf{W}_\ell^{(\mathbf{h})}$ and $\mathbf{W}_{\ell,j}^{(\mathbf{t})}$ are weight matrices. When $k = 1$ and $i_1 = \ell - 1$ for all layers ℓ, we have the original RHN (Zilly et al., 2017). It is

worth noting that the update rule in Eqn. 3 can be rewritten as

$$\mathbf{s}_\ell \leftarrow \mathbf{s}_{\ell-1} + \underbrace{(\mathbf{h}_\ell - \mathbf{s}_{\ell-1}) \otimes \mathbf{t}_\ell}_{\delta_\ell} \tag{4}$$

If $\mathbb{E}_{\text{data}\sim\mathcal{D}}[\delta_\ell] = 0$, this update rule is consistent with the feature identity and the unrolled iterative estimation interpretations of highway networks (Greff et al., 2017).

Eqn. 2 has a pitfall: if a layer $\mathbf{s}_\ell$ does not receive any connection from all previous layers, the model fails to compile. This phenomenon was noted in Zoph and Le (2017) as compilation failure. We resolve the issue by forcing each layer $\mathbf{s}_\ell$ to be connected to $\mathbf{s}_{\ell-1}$, effectively absorbing the highway connection into the skip connections.

4.3 The Policy Network

Following Zoph and Le (2017), we parameterize the policy $\pi(\mathbf{m};\theta)$ with an RNN. In all experiments, we use a two-layer stacked Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997; Sutskever et al., 2014) with 64 hidden units per layer to model a sequence of tokens. The meaning of the sequence of tokens depends on the class of architectures we want to design, as we discuss below.

General Form of the Controller. Figure 2 illustrates the RNN that models our general controller. In order to design a convolutional architecture with L layers, each with B branches, we let the controller RNN run for L blocks of $B+1$ steps. In each block, the first B steps sample B masks for the corresponding B channels. Each mask is an integer in $(0, 2^{C/S} - 1]$, where C and S are as in Section 4.1. The last step in each block is the anchor step, whose output is used in a content-based attention function (Bahdanau et al., 2015) to sample the binary decisions whether to establish a skip connections between layers. Specifically, for layers i and j if $i < j$ then

$$p(\text{layer } i \text{ is connected to layer } j) \propto \exp\left(\tanh\left(\text{anchor}_i \cdot \mathbf{W}_1 + \text{anchor}_j \cdot \mathbf{W}_2\right) \cdot \mathbf{v}\right), \tag{5}$$

where $\mathbf{W}_1$, $\mathbf{W}_2$, and $\mathbf{v}$ are trainable parameters. If at layer j the controller samples k skip connections to the previous layers i_1, i_2, ..., i_k then the embedding to the next step in the next block of the RNN is the average of the corresponding k anchor steps.

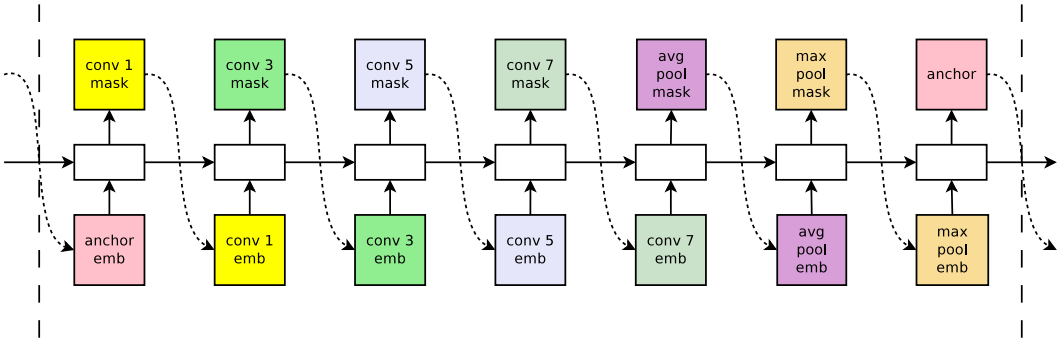

Figure 2: A block of B mask predictions and an anchor prediction. If the desired convolutional network has L layers, then this block is replicated L times.

Practical Restricted Controller. Since the described controller results in a very large search space, we explore two of its restrictions, corresponding to two sub-search spaces. In the first restriction, we remove the anchor step in each block, equivalently removing the controller's ability to design skip connections. Instead, we enforce the network to use the skip connections pattern of DenseNet (Huang et al., 2016).

In the second restriction, we keep only the anchor step in each blocks, remove all the mask predictions, and force the network to use convolutions of filter size 3×3 and C

channels, equivalently focusing the controller to find a good skip connection pattern. Our best convolutional architecture is obtained by this variation, described in more details in Section 5. This is consistent with the observation in (Zoph and Le, 2017) that (1) if the search space is too large, the controller fails to find the good configurations, and (2) fixing a particular dimension in the search space and then using some manual designs lead to the best models. For recurrent models, as we only want to design the skip connection patterns in RHN cells (cf. Section 4.2), we simply employ this second restriction.

4.4 Optimization Algorithm and Convergence Guarantee

With the described specifications on the models and parameters, the training algorithm of ENAS consists of two alternating phases, which we describe below. The first phase trains ω on a whole pass through the training data set, then the second phase trains θ for a fixed number of steps, which we set to 2000.

Training the Shared Parameters ω. In this step, we fix the policy $\pi(\mathbf{m};\theta)$ and perform stochastic gradient descent (SGD) updates on ω to minimize the expected loss function $\mathcal{L}(\mathbf{m};\omega)$. The gradient is computed via the unbiased Monte Carlo estimate

$$\nabla_\omega \mathbb{E}_{\mathbf{m}\sim\pi(\mathbf{m};\theta)}\left[\mathcal{L}(\mathbf{m};\omega)\right] \approx \frac{1}{M}\sum_{i=1}^{M} \nabla_\omega \mathcal{L}(\mathbf{m}_i,\omega) \tag{6}$$

We find that $M = 1$ works just fine, i.e. we can update ω using the gradient from *any single model* $\mathbf{m}$ sampled from $\pi(\mathbf{m};\theta)$. We train ω for a whole pass through the training data set.

Training the Policy $\pi(\mathbf{m};\theta)$. In this step, we fix ω and update the policy parameters θ, aiming to maximize the expected reward $\mathbb{E}_{\mathbf{m}\sim\pi(\mathbf{m};\theta)}\left[R(\mathbf{m},\omega)\right]$. We employ the Adam optimizer (Kingma and Ba, 2015), for which the gradient is computed via the REINFORCE equation (Williams, 1992), with variance reduction using a moving average baseline $\hat{b}$:

$$\nabla_\theta \mathbb{E}_{\mathbf{m}\sim\pi(\mathbf{m};\theta)}\left[R(\mathbf{m},\omega)\right] \approx \frac{1}{M}\sum_{i=1}^{M} \Big(R(\mathbf{m}_i;\omega) - \hat{b}\Big) \cdot \nabla_\theta \log p(\mathbf{m}_i;\theta) \tag{7}$$

The reward $R(\mathbf{m},\omega)$ is computed on a minibatch of examples sampled from *the validation set,* rather than the training set, as we want to encourage ENAS to select the model that generalizes well, rather than the model that overfits the training set well. We try two alternative implementations of this step, where we evaluate $R(\mathbf{m},\omega)$ only on the training set, and on minibatches either from the training set or the validation set. The resulting training algorithms converge faster, but the found architectures generalize poorly.

The reward function for image classification is the rate of correct classification on a minibatch of images sampled from the validation set. The reward function for language model is $c/\text{valid_ppl}$, where the perplexity is computed on a minibatch, also sampled from the validation set. This reward function is different from the reward function proposed by Zoph and Le (2017), which is $c/\text{valid_ppl}^2$. In Appendix A, we derive these choices with theoretical justifications that they stabilize the interleaving updates in Eqn. 6 and 7.

4.5 Inference

We propose two approaches to inference. In the first approach, we sample several models from $\pi(\mathbf{m},\theta)$. We compute each model's reward on a single minibatch sampled from the validation set. We then take the model with the highest reward to re-train from scratch. In the second approach, we also sample models from $\pi(\mathbf{m};\theta)$, but unlike the first approach, we keep ω and evaluate $R(\hat{\mathbf{m}},\omega)$ as the ensemble reward of these models. Specifically, with the image classification task, we average the pre-softmax logits of the sampled models, and with the language model task, we average the post-softmax logits of the sampled models.

In our experiments, the first approach yields better results. To match the performance of an architecture retrained using the first approach, e.g. in terms of accuracy on CIFAR-10, the

second approach needs as many as 171 samples, making its inference more expensive than the inference of a single retrained model. Our interpretation is that ENAS does achieve the same effect of NAS methods, i.e. discovering good architectures, but does not learn a good set of parameters to be shared by all of them.

5 Experiments

5.1 Image Classification on CIFAR-10

Dataset. The CIFAR-10 dataset (Krizhevsky, 2009) consists of 50,000 training images and 10,000 test images. We use the standard data pre-processing and augmentation techniques, i.e. subtracting the mean and dividing the standard deviation from each channel computed on the training images, centrally padding the training images to 40×40 and randomly cropping them back to 32×32, and randomly flipping them horizontally.

Training details. The shared parameters ω is trained with Nesterov momentum (Nesterov, 1983), where the learning rate follows the cosine schedule with $l_{\max} = 0.05$, $l_{\min} = 0.001$, $T_0 = 10$ and $T_{\text{mul}} = 2$ (Loshchilov and Hutter, 2017). Each architecture search is run for $10 + 20 + 40 + 80 + 160 = 310$ epochs. Each weight component in ω is initialized from a scaled Gaussian as described in He et al. (2015). We also apply an ℓ_2 weight decay of 10^{-4}. The same settings are employed to train the architecture recommended by the controller.

The policy parameters θ are initialized uniformly in $[-0.1, 0.1]$, and trained with the Adam optimizer at a learning rate of 10^{-3}. We additionally utilize three techniques to prevent the premature convergence of REINFORCE. First, a temperature $\tau = 5.0$ and a tanh constant $c = 2.5$ are applied to the controller's logits, i.e. every sample s from the controller's logits $\mathbf{u}$ comes from the distribution $s \sim \text{softmax}(c \cdot \tanh(\mathbf{u}/\tau))$. Second, we add to the reward the entropy term of the controller's samples weighted by $\lambda_{\text{ent}} = 0.1$, which discourages convergence (Williams and Peng, 1991). Lastly, we enforce the sparsity in the skip connections by adding to the reward the Kullback-Leibler divergence between the skip connection probability in Eqn. 5 and a chosen probability $\rho = 0.4$, which represents the prior belief of a skip connection being formed. The KL divergence term is weighted by $\lambda_{\text{kl}} = 0.5$ in our reward.

Results. Table 1 summarizes the test errors of ENAS and other approaches. The time taken to discover our models are reported in Table 2. More details are as follows.

In the first experiment, we search for the masks at each branch and each layer in a 12-layer network. Layers 4th and 8th are max pooling layers with a kernel size of 2×2 and a stride of 2, which reduce each spatial dimension of the layers' outputs by a factor of 2. Within each group of 3 layers where the spatial dimensions of the layers remain constant, we connect each layer to all layers before it (Huang et al., 2016). We use the block size of $S = 32$, resulting in $C/S = 256/32 = 8$ blocks per branch per layer. The resulting model, depicted in Figure 3, almost always has 64 or 96 channels at each branch and each layer, indicating that the controller does *not* choose to activate all blocks. This is the desired behavior, as doing so would over-parametrize the model and result in overfitting. The search for this architecture takes only 11.6 hours and yet the resulting model achieves the test error of 4.35%, which is better than all but one model reported by Zoph and Le (2017).

In the second experiment, we search for a good pattern of skip connections. We use 3×3 convolutions with 48 output channels at all layers. Our controller is allowed to form skip connections between arbitrary layers, but forming such connections between layers with different spatial dimensions would result in compilation failures. To circumvent, after each max pooling in the network, we centrally pad the output so that its spatial dimensions remain unchanged. In 12.4 hours, this process discovers the pattern of skip connections depicted in Figure 4. This pattern has the property that skip connections are formed much more densely at higher layers than at lower layers, where most connections are only between

[2] We took the authors' published code, ran it for a few epochs and interpolated the running time.

Method	**Depth**	**Parameters** (million)	**Error** (%)
ResNet (He et al., 2016)	110	1.7	6.61
WideResNet (Zagoruyko and Komodakis, 2016)	28	36.5	4.17
DenseNet-BC (Huang et al., 2016)	190	25.6	3.46
Multi-Branch Net (Ahmed and Torresani, 2017)	26	34.3	3.19
Shake-Shake 26 2x96d (Gastaldi, 2016)	26	26.2	2.86
Shake-Shake + CutOut (DeVries and Taylor, 2017)	26	26.2	**2.56**
Budgeted Super Nets (Veniat and Denoyer, 2017)	16	–	9.21
ConvFabrics Dense (Saxena and Verbeek, 2016)	16	21.2	7.43
Macro NAS with Q-Learning (Baker et al., 2017)	11	11.2	6.92
Net Transformation (Cai et al., 2017)	17	19.7	5.70
FractalNet (Larsson et al., 2017)	21	38.6	4.60
SMASH (Brock et al., 2017)	211	16.0	4.03
Micro NAS with Q-Learning (Zhong et al., 2017)	24	–	**3.60**
NAS with stride (Zoph and Le, 2017)	20	2.5	6.01
NAS no stride or pooling	15	4.2	5.50
NAS + max pooling	39	7.1	4.47
NAS + max pooling + more filters	39	37.4	**3.65**
NASNet-B (Zoph et al., 2017)	14	2.6	3.73
NASNet-C	14	3.1	3.59
NASNet-A	20	3.3	**3.41**
ENAS + channels + dense connection	12	12.6	4.35
ENAS + connections + conv 3×3	15	14.1	5.04
ENAS + connections + locally dense connections	39	19.4	4.42
ENAS + connections + multi branches	15	32.0	**3.86**

Table 1: Classification error rates of ENAS and other methods on CIFAR-10. In this table, the first block presents the state-of-the-art models, all of which are designed by human experts. The second block presents various NAS approaches that do not use more than 50 GPUs. All models in this second block have higher test error rates than ENAS except for Zhong et al. (2017), which use 32 GPUs. The next two blocks present the state-of-the-art NAS techniques, which are very computationally expensive. The last block presents the performance of ENAS .

Method	**GPUs**	**Time** (days)
Macro NAS with Q-Learning (Baker et al., 2017)	10	8-10
Net Transformation (Cai et al., 2017)	5	3
SMASH (Brock et al., 2017)	1	1.5[2]
Micro NAS with Q-Learning (Zhong et al., 2017)	32	3
NAS (Zoph and Le, 2017)	800	21-28
Micro NAS (Zoph et al., 2017)	450	3-4
ENAS + channels	1	0.48
ENAS + connections	1	0.52

Table 2: Time taken to search for the desired architectures for CIFAR-10.

consecutive layers. The architecture found in this experiment has the test error of 5.04%, which is slightly worse than the one found in the first experiment.

We manually augment the pattern of skip connections using two approaches. First, similar to Zoph and Le (2017), we replace the 3×3 convolutions at each layer with 3 consecutive layers that are densely connected, each has 256 channels. The resulting model achieves a

better test error of 4.42%. Second, following Xie et al. (2017) and Ahmed and Torresani (2017), we replace blocks of 3 densely connected layers with 4 parallel branches, where each branch performs a 3×3 convolution on the input layer and their outputs are combined using a 1×1 convolution. The resulting architecture achieves the test error of 3.86%.

Sanity Check with Ablation Study. To assert the role of ENAS, we carry out two sanity check experiments. In the first study, we uniformly at random pick a configuration of channels and skip connections and just train a model. As a result, about half of the channels and skip connections are selected, resulting in a model with 47.1M parameters, which has the error rate of 5.86%. This error rate is worse than the models designed by ENAS, which have fewer parameters. In the second study, we only train ω and do not update the controller. The effect is similar to dropout with a rate of 0.5 on both the channels and the skip connections. At convergence, the model has the error rate of 11.92%. On the validation set, the ensemble of 250 Monte Carlo configurations of the trained model could only reach 8.99% test error rate. We therefore conclude that the appropriate training of the ENAS controller is crucial for good performance.

5.2 Language Model with Penn Treebank

Training details. Our controller is trained with the same settings as described in Section 5.1 for CIFAR-10, except that now we anneal the skip connection probability to $\rho = 0.45$ and set learning rate for Adam to 0.005. The shared parameters ω are trained using stochastic gradient descent with a learning rate of 0.2, decayed by a factor of 0.9 after every 3 epochs starting at epoch 15, for a total of 150 epochs. During the architecture search process, following Melis et al. (2017), we randomly reset the starting state with probability of 0.001. We also employ weight tying (Inan et al., 2017). No additional regularization technique is applied. When retraining the architecture recommended by the controller, however, we use variational dropout and an ℓ_2 regularization with weight decay of 10^{-7}.

Results. Table 3 presents our results in comparison with other methods. In 8.2 hours, our controller finds the architecture as depicted in Figure 5 in the Appendix. This architecture has only 8 million parameters and achieves the test perplexity of 71.3, which is better than LSTM trained using the same setting. Finally, if we increase the number of hidden units in our model so that the total number of parameters is 21M, then it achieves the test perplexity of 64.6, which is very close to the test perplexity found by full-scaled NAS (Zoph and Le, 2017), which uses orders of magnitude more computing resources and time.

Method	**Parameters** (million)	**Test Perplexity**
LSTM+Vanilla Dropout (Zaremba et al., 2014)	66	78.4
LSTM+VD (Gal and Ghahramani, 2016)	66	75.2
LSTM+VD+MC (Gal and Ghahramani, 2016)	66	73.4
LSTM+WT (Inan et al., 2017)	51	68.5
Recurrent Highway Network (Zilly et al., 2017)	24	66.0
LSTM+Hyper-parameters Search (Melis et al., 2017)	24	59.5
LSTM+AWD (Merity et al., 2017)	24	52.8
LSTM+AWD+Dynamic Eval (Krause et al., 2017)	24	51.1
NAS (Zoph and Le, 2017)	32	67.9
NAS+VD	25	64.0
NAS+VD+WT (Zoph and Le, 2017)	54	62.4
ENAS (small)	8	71.3
ENAS (large)	21	64.6

Table 3: Test perplexity on Penn Treebank of ENAS and other approaches. VD = Variational Dropout; WT = Weight Tying; MC = Monte Carlo sampling.

6 Conclusion

Neural Architecture Search (NAS) is an important advance that allows faster architecture design for neural networks. However, the computational expense of NAS prevents it from being widely adopted. In this paper, we presented ENAS, an alternative method to NAS, that requires less resources and time. The key insight of our method is to share parameters across child models during architecture search. This insight is implemented by having NAS search for a path within a larger model. We demonstrate empirically that the method works well on both CIFAR-10 and Penn Treebank datasets.

References

Ahmed, K. and Torresani, L. (2017). Connectivity learning in multi-branch networks. *Arxiv, 1709.09582*.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *ICLR*.

Baker, B., Gupta, O., Naik, N., and Raskar, R. (2017). Designing neural network architectures using reinforcement learning. In *ICLR*.

Bello, I., Zoph, B., Vasudevan, V., and Le, Q. V. (2017). Neural optimizer search with reinforcement learning. In *ICML*.

Bottou, L. (1991). *Une Approche thÃľorique de lâĂŹApprentissage Connexioniste: Applications Ãă la reconnaissance de la Parole.* PhD thesis.

Brock, A., Lim, T., Ritchie, J. M., and Weston, N. (2017). SMASH: one-shot model architecture search through hypernetworks. *Arxiv, 1708.05344*.

Cai, H., Chen, T., Zhang, W., Yu, Y., and Wang, J. (2017). Reinforcement learning for architecture search by network transformation. *Arxiv, 1707.04873*.

DeVries, T. and Taylor, G. W. (2017). Improved regularization of convolutional neural networks with cutout. *Arxiv, 1708.04552*.

Gal, Y. and Ghahramani, Z. (2016). A theoretically grounded application of dropout in recurrent neural networks. In *NIPS*.

Gastaldi, X. (2016). Shake-shake regularization of 3-branch residual networks. In *ICLR Workshop Track*.

Greff, K., Srivastava, R. K., and Schmidhuber, J. (2017). Highway and residual networks learn unrolled iterative estimation. In *ICLR*.

Ha, D., Dai, A., and Le, Q. V. (2017). Hypernetworks. In *ICLR*.

He, K., Zhang, X., Rein, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *CVPR*.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *CPVR*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. In *Neural Computations*.

Huang, G., Liu, Z., van der Maaten, L., and Weinberger, K. Q. (2016). Densely connected convolutional networks. In *CVPR*.

Inan, H., Khosravi, K., and Socher, R. (2017). Tying word vectors and word classifiers: a loss framework for language modeling. In *ICLR*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*.

Kingma, D. P. and Ba, J. L. (2015). Adam: A method for stochastic optimization. In *ICLR*.

Krause, Ben Kahembwe, E., Murray, I., and Renals, S. (2017). Dynamic evaluation of neural sequence models. *Arxiv, 1709.07432*.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical report.

Larsson, G., Maire, M., and Shakhnarovich, G. (2017). Fractalnet: Ultra-deep neural networks without residuals. In *ICLR*.

Lin, M., Chen, Q., and Yan, S. (2013). Network in network. *Arxiv, 1312.4400*.

Loshchilov, I. and Hutter, F. (2017). Sgdr: Stochastic gradient descent with warm restarts. In *ICLR*.

Melis, G., Dyer, C., and Blunsom, P. (2017). On the state of the art of evaluation in neural language models. *Arxiv, 1707.05589*.

Merity, S., Keskar, N. S., and Socher, R. (2017). Regularizing and optimizing LSTM language models. *Arxiv, 1708.02182*.

Negrinho, R. and Gordon, G. (2017). Deeparchitect: Automatically designing and training deep architectures. In *CPVR*.

Nesterov, Y. E. (1983). A method for solving the convex programming problem with convergence rate $o(1/k^2)$. *Soviet Mathematics Doklady*.

Saxena, S. and Verbeek, J. (2016). Convolutional neural fabrics. In *NIPS*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *Arxiv, 1707.06347*.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *NIPS*.

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., and Wojna, Z. (2016). Rethinking the inception architecture for computer vision. In *CPVR*.

Veniat, T. and Denoyer, L. (2017). Learning time-efficient deep architectures with budgeted super networks. *Arxiv, 1706.00046*.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*.

Williams, R. J. and Peng, J. (1991). Function optimization using connectionist reinforcement learning algorithms. *Connection Science*, 3(3):241–268.

Xie, S., Girshick, R., Dollár, P., Tu, Z., and He, K. (2017). Aggregated residual transformations for deep neural networks. In *CVPR*.

Zagoruyko, S. and Komodakis, N. (2016). Wide residual networks. In *BMVC*.

Zaremba, W., Sutskever, I., and Vinyals, O. (2014). Recurrent neural network regularization. *Arxiv, 1409.2329*.

Zhong, Z., Yan, J., and Liu, C. (2017). Practical network blocks design with q-learning. *Arxiv, 1708.05552*.

Zilly, J. G., Srivastava, R. K., Koutník, J., and Schmidhuber, J. (2017). Recurrent highway networks. In *ICML*.

Zoph, B. and Le, Q. V. (2017). Neural architecture search with reinforcement learning. In *ICLR*.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. (2017). Learning transferable architectures for scalable image recognition. *Arxiv, 1707.07012*.

Appendices

A Derivation of Reward Functions

For model $\mathbf{m}$ with shared parameters ω, the cross-entropy loss $\mathcal{L}(\mathbf{m}, \omega)$ can be computed by integrating over the data space $\mathbf{x} \sim P(\mathbf{x})$

$$\mathcal{L}(\mathbf{m}, \omega) = -\int dP(\mathbf{x}) \log p(\mathbf{x}|\mathbf{m}, \omega). \tag{8}$$

Below, we show that the reinforcement learning signals $R(\mathbf{m}, \omega)$ that we specified in Section 4.4 are the unbiased Monte Carlo approximations of the posterior of the data, i.e. $\int dP(\mathbf{x}) p(\mathbf{x}|\mathbf{m}, \omega)$.

Before we present the derivation, let us put forth a remark. In Chapter 3.3.2 of Bottou (1991), it was shown that if the estimations of gradients $\hat{\nabla}_\omega$ and $\hat{\nabla}_\theta$ are unbiased, then under reasonable assumptions of the function to optimize, the SGD updates of ω converge almost surely. In other words, if we fix $\pi(\mathbf{m}, \theta)$ and just perform SGD to update ω, it will converge to a local minimum. However, the role of $\pi(\mathbf{m}; \theta)$ is to calibrate the expectation $\mathbb{E}_{\mathbf{m}}[\mathcal{L}(\mathbf{m}, \omega)]$. Specifically, by Jensen's inequality

$$\begin{aligned}
\mathbb{E}_{\mathbf{m}}[\mathcal{L}(\mathbf{m}, \omega)] &= -\mathbb{E}_{\mathbf{m}}\left[\int dP(\mathbf{x}) \log p(\mathbf{x}|\mathbf{m}, \omega)\right] && (9)\\
&\leq -\log \mathbb{E}_{\mathbf{m}}\left[\int dP(\mathbf{x}) p(\mathbf{x}|\mathbf{m}, \omega)\right] && (10)\\
&= -\log \mathbb{E}_{\mathbf{m}}[R(\mathbf{m}, \omega)]. && (11)
\end{aligned}$$

Thus, maximizing $\mathbb{E}_{\mathbf{m}\sim\pi(\mathbf{m},\theta)}[R(\mathbf{m}, \omega)]$ pushes down a surrogate function of $\mathbb{E}_{\mathbf{m}}[\mathcal{L}(\mathbf{m}, \omega)]$, similar to the M-step in the family of Expectation-Maximization algorithms.

Now we present the derivation. On an image classification task, the data $\mathbf{x}$ is actually a pair $(\mathbf{x}, \mathbf{y})$ of an image and its label. If instead of predicting the label by taking argmax of the model's probabilities, we sample the label $\hat{\mathbf{y}} \sim p(\mathbf{y}|\mathbf{x}, \mathbf{m}, \omega)$, as done in Bayesian classifiers, then the expected accuracy is

$$\begin{aligned}
\int dP(\mathbf{x}, \mathbf{y}) \mathbb{E}_{\mathbf{m}}[R(\mathbf{m}, \omega)] &= \int dP(\mathbf{x}, \mathbf{y}) \mathbb{E}_{\mathbf{m}}\left[\mathbb{E}_{\hat{\mathbf{y}}\sim p(\mathbf{y}|\mathbf{x},\mathbf{m},\omega)} \mathbf{1}[\hat{\mathbf{y}} = \mathbf{y}]\right] && (12)\\
&= \int dP(\mathbf{x}, \mathbf{y}) \mathbb{E}_{\mathbf{m}}\left[p(\mathbf{y}|\mathbf{x}, \mathbf{m}, \omega)\right]. && (13)
\end{aligned}$$

The accuracy on a minibatch is thus a Monte Carlo approximation of of $\mathbb{E}_{\mathbf{m}}[p(\mathbf{x}|\mathbf{m}, \omega)]$. On a language model task, the data $\mathbf{x}$ is a sequence of words. Letting $|\mathbf{x}|$ be the number of words in $\mathbf{x}$, the expectation of the proposed reward in Section 4.4 is

$$\begin{aligned}
\int dP(\mathbf{x}) \mathbb{E}_{\mathbf{m}}\left[\frac{c}{\text{valid_ppl}(\mathbf{x}|\mathbf{m}, \omega)}\right] &= -c \int dP(\mathbf{x}) \mathbb{E}_{\mathbf{m}}\left[\exp \frac{\log p(\mathbf{x}|\mathbf{m}, \omega)}{|\mathbf{x}|}\right] && (14)\\
&= -c \int dP(\mathbf{x}) \mathbb{E}_{\mathbf{m}}\left[p(\mathbf{x}|\mathbf{m}, \omega) - \exp|\mathbf{x}|\right] && (15)\\
&\propto -c \int dP(\mathbf{x}) \mathbb{E}_{\mathbf{m}}\left[p(\mathbf{x}|\mathbf{m}, \omega)\right]. && (16)
\end{aligned}$$

Since we use $R(\mathbf{m}, \omega)$ as the signal for reinforcement learning, we can omit the constant $c \int dP(\mathbf{x}) \exp|\mathbf{x}|$ since it will integrate to 0 in the REINFORCE equation, just like the constant baseline (Williams, 1992). This justifies the use of the signal $\frac{c}{\text{valid_ppl}(\mathbf{x})}$ for learning.

B Models found for CIFAR-10

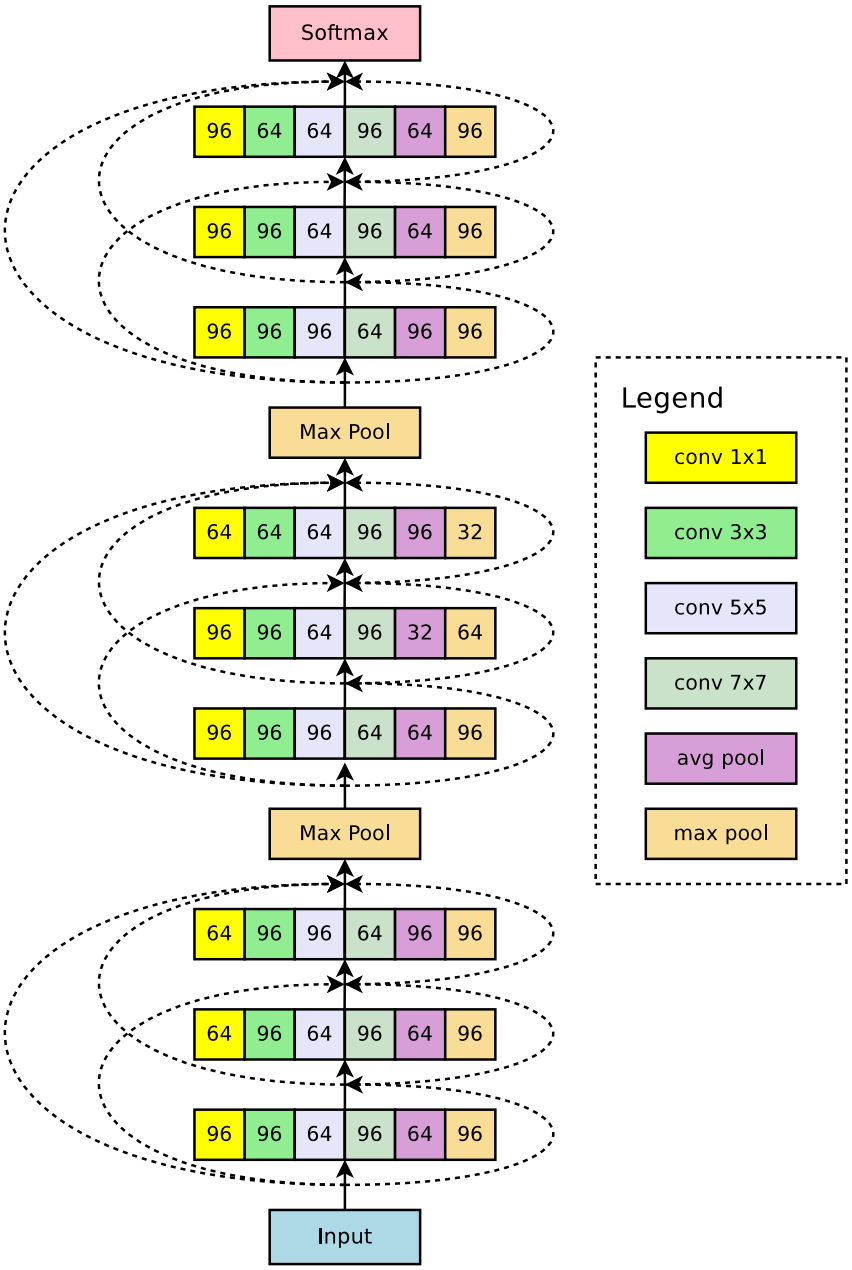

Figure 3: A configuration of channels found by ENAS for CIFAR-10.

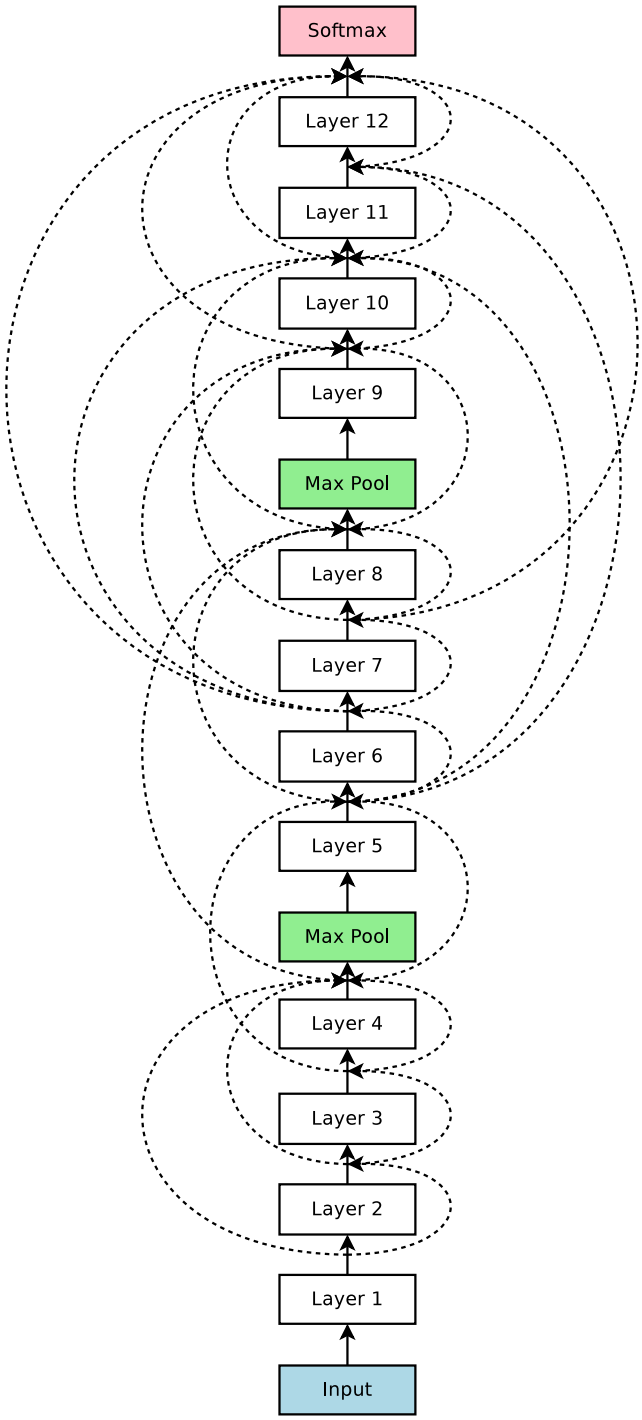

Figure 4: A skip connection pattern found by ENAS for CIFAR-10.

C Models found for Penn Treebank

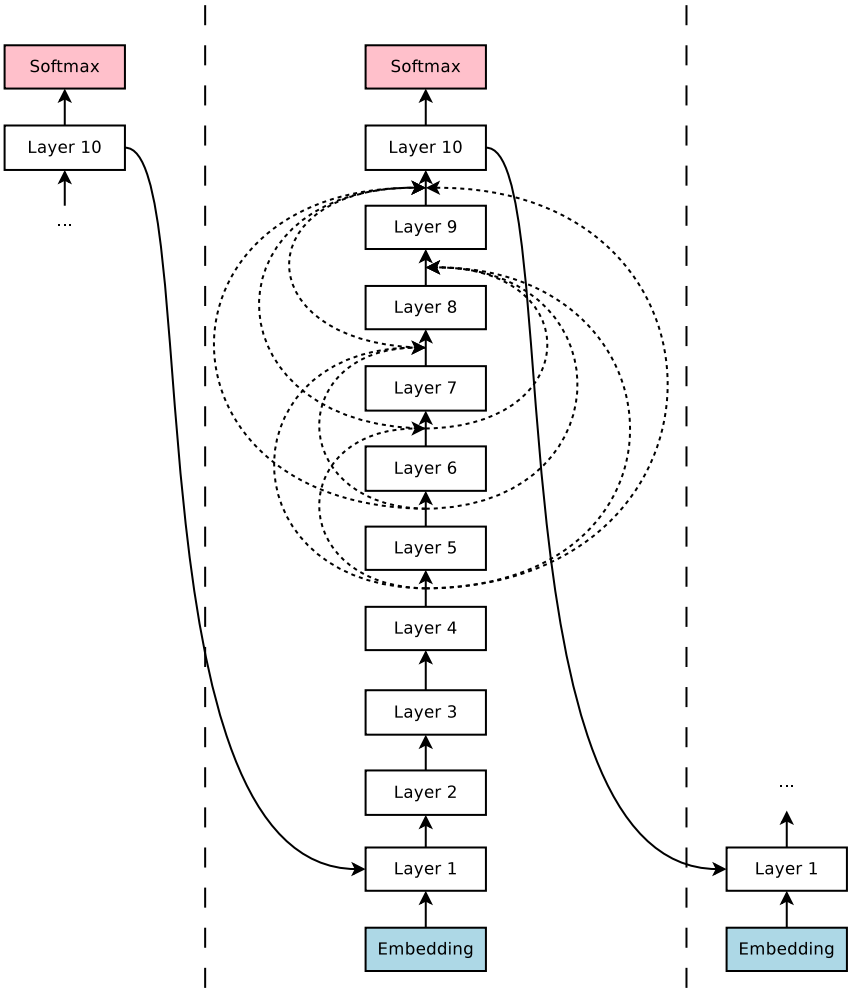

Figure 5: A pattern of skip connections that ENAS found to augment the RHN cell on Penn Treebank.