Efficient Neural Architecture Search via Parameter Sharing

Hieu Pham [*1 2] **Melody Y. Guan** [*3] **Barret Zoph** [1] **Quoc V. Le** [1] **Jeff Dean** [1]

Abstract

We propose *Efficient Neural Architecture Search* (ENAS), a fast and inexpensive approach for automatic model design. In ENAS, a controller discovers neural network architectures by searching for an optimal subgraph within a large computational graph. The controller is trained with policy gradient to select a subgraph that maximizes the expected reward on a validation set. Meanwhile the model corresponding to the selected subgraph is trained to minimize a canonical cross entropy loss. Sharing parameters among child models allows ENAS to deliver strong empirical performances, while using much fewer GPU-hours than existing automatic model design approaches, and notably, 1000x less expensive than standard Neural Architecture Search. On the Penn Treebank dataset, ENAS discovers a novel architecture that achieves a test perplexity of 55.8, establishing a new state-of-the-art among all methods without post-training processing. On the CIFAR-10 dataset, ENAS finds a novel architecture that achieves 2.89% test error, which is on par with the 2.65% test error of NASNet (Zoph et al., 2018).

1. Introduction

Neural architecture search (NAS) has been successfully applied to design model architectures for image classification and language models (Zoph & Le, 2017; Zoph et al., 2018; Cai et al., 2018; Liu et al., 2017; 2018). In NAS, an RNN controller is trained in a loop: the controller first samples a candidate architecture, *i.e.* a *child model*, and then trains it to convergence to measure its performance on the task of desire. The controller then uses the performance as a guiding signal to find more promising architectures. This process is repeated for many iterations. Despite its impressive empirical performance, NAS is computationally expensive and time consuming, *e.g.* Zoph et al. (2018) use 450 GPUs for 3-4 days (*i.e.* 32,400-43,200 GPU hours). Meanwhile, using less resources tends to produce less compelling results (Negrinho & Gordon, 2017; Baker et al., 2017a). We observe that the computational bottleneck of NAS is the training of each child model to convergence, only to measure its accuracy whilst throwing away all the trained weights.

The main contribution of this work is to improve the efficiency of NAS by *forcing all child models to share weights* to eschew training each child model from scratch to convergence. The idea has apparent complications, as different child models might utilize their weights differently, but was encouraged by previous work on transfer learning and multitask learning, which established that parameters learned for a particular model on a particular task can be used for other models on other tasks, with little to no modifications (Razavian et al., 2014; Zoph et al., 2016; Luong et al., 2016).

We empirically show that not only is sharing parameters among child models possible, but it also allows for very strong performance. Specifically, on CIFAR-10, our method achieves a test error of 2.89%, compared to 2.65% by NAS. On Penn Treebank, our method achieves a test perplexity of 55.8, which significantly outperforms NAS's test perplexity of 62.4 (Zoph & Le, 2017) and which is a new state-of-the-art among Penn Treebank's approaches that do not utilize post-training processing. Importantly, in all of our experiments, for which we use a single Nvidia GTX 1080Ti GPU, the search for architectures takes less than 16 hours. Compared to NAS, this is a reduction of GPU-hours by more than 1000x. Due to its efficiency, we name our method *Efficient Neural Architecture Search* (ENAS).

2. Methods

Central to the idea of ENAS is the observation that all of the graphs which NAS ends up iterating over can be viewed as sub-graphs of a larger graph. In other words, we can represent NAS's search space using a *single* directed acyclic graph (DAG). Figure 2 illustrates a generic example DAG, where an architecture can be realized by taking a subgraph of the DAG. Intuitively, ENAS's DAG is the su-

[*]Equal contribution [1]Google Brain [2]Language Technology Institute, Carnegie Mellon University [3]Department of Computer Science, Stanford University. Correspondence to: Hieu Pham <hyhieu@cmu.edu>, Melody Y. Guan <mguan@stanford.edu>.

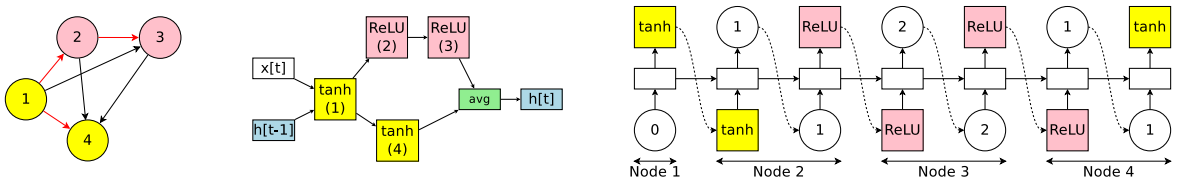

Figure 1. An example of a recurrent cell in our search space with 4 computational nodes. *Left:* The computational DAG that corresponds to the recurrent cell. The red edges represent the flow of information in the graph. *Middle:* The recurrent cell. *Right:* The outputs of the controller RNN that result in the cell in the middle and the DAG on the left. Note that nodes 3 and 4 are never sampled by the RNN, so their results are averaged and are treated as the cell's output.

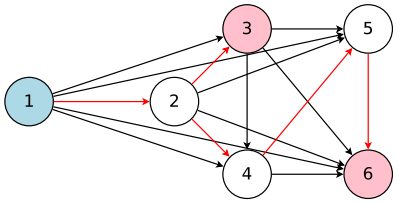

Figure 2. The graph represents the entire search space while the red arrows define a model in the search space, which is decided by a controller. Here, node 1 is the input to the model whereas nodes 3 and 6 are the model's outputs.

perposition of all possible child models in a search space of NAS, where the nodes represent the local computations and the edges represent the flow of information. The local computations at each node have their own parameters, which are used only when the particular computation is activated. Therefore, ENAS's design allows parameters to be shared among all child models, *i.e.* architectures, in the search space.

In the following, we facilitate the discussion of ENAS with an example that illustrates how to design a cell for recurrent neural networks from a specified DAG and a controller (Section 2.1). We will then explain how to train ENAS and how to derive architectures from ENAS's controller (Section 2.2). Finally, we will explain our search space for designing convolutional architectures (Sections 2.3 and 2.4).

2.1. Designing Recurrent Cells

To design recurrent cells, we employ a DAG with N nodes, where the nodes represent local computations, and the edges represent the flow of information between the N nodes. ENAS's controller is an RNN that decides: 1) which edges are activated and 2) which computations are performed at each node in the DAG. This design of our search space for RNN cells is different from the search space for RNN cells in Zoph & Le (2017), where the authors fix the topology of their architectures as a binary tree and only learn the operations at each node of the tree. In contrast, our search space allows ENAS to design both the topology and the operations in RNN cells, and hence is more flexible.

To create a recurrent cell, the controller RNN samples N blocks of decisions. Here we illustrate the ENAS mechanism via a simple example recurrent cell with $N = 4$ computational nodes (visualized in Figure 1). Let $\mathbf{x}_t$ be the input signal for a recurrent cell (*e.g.* word embedding), and $\mathbf{h}_{t-1}$ be the output from the previous time step. We sample as follows.

1. At node 1: The controller first samples an activation function. In our example, the controller chooses the $\tanh$ activation function, which means that node 1 of the recurrent cell should compute $h_1 = \tanh\left(\mathbf{x}_t \cdot \mathbf{W}^{(\mathbf{x})} + \mathbf{h}_{t-1} \cdot \mathbf{W}_1^{(\mathbf{h})}\right)$.

2. At node 2: The controller then samples a previous index and an activation function. In our example, it chooses the previous index 1 and the activation function ReLU. Thus, node 2 of the cell computes $h_2 = \text{ReLU}(h_1 \cdot \mathbf{W}_{2,1}^{(\mathbf{h})})$.

3. At node 3: The controller again samples a previous index and an activation function. In our example, it chooses the previous index 2 and the activation function ReLU. Therefore, $h_3 = \text{ReLU}(h_2 \cdot \mathbf{W}_{3,2}^{(\mathbf{h})})$.

4. At node 4: The controller again samples a previous index and an activation function. In our example, it chooses the previous index 1 and the activation function $\tanh$, leading to $h_4 = \tanh\left(h_1 \cdot \mathbf{W}_{4,1}^{(\mathbf{h})}\right)$.

5. For the output, we simply average all the loose ends, *i.e.* the nodes that are not selected as inputs to any other nodes. In our example, since the indices 3 and 4 were never sampled to be the input for any node, the recurrent cell uses their average $(h_3 + h_4)/2$ as its output. In other words, $\mathbf{h}_t = (h_3 + h_4)/2$.

In the example above, we note that for each pair of nodes $j < \ell$, there is an independent parameter matrix $\mathbf{W}_{\ell,j}^{(\mathbf{h})}$. As shown in the example, by choosing the previous indices, the controller also decides which parameter matrices are used. Therefore, in ENAS, all recurrent cells in a search space share the same set of parameters.

Our search space includes an exponential number of configurations. Specifically, if the recurrent cell has N nodes

and we allow 4 activation functions (namely tanh, ReLU, identity, and sigmoid), then the search space has $4^N \times N!$ configurations. In our experiments, $N = 12$, which means there are approximately 10^{15} models in our search space.

2.2. Training ENAS and Deriving Architectures

Our controller network is an LSTM with 100 hidden units (Hochreiter & Schmidhuber, 1997). This LSTM samples decisions via softmax classifiers, in an autoregressive fashion: the decision in the previous step is fed as input embedding into the next step. At the first step, the controller network receives an empty embedding as input.

In ENAS, there are two sets of learnable parameters: the parameters of the controller LSTM, denoted by θ, and the shared parameters of the child models, denoted by ω. The training procedure of ENAS consists of two interleaving phases. The first phase trains ω, the shared parameters of the child models, on a whole pass through the training data set. For our Penn Treebank experiments, ω is trained for about 400 steps, each on a minibatch of 64 examples, where the gradient ∇_ω is computed using back-propagation through time, truncated at 35 time steps. Meanwhile, for CIFAR-10, ω is trained on $45,000$ training images, separated into minibatches of size 128, where ∇_ω is computed using standard back-propagation. The second phase trains θ, the parameters of the controller LSTM, for a fixed number of steps, typically set to 2000 in our experiments. These two phases are alternated during the training of ENAS. More details are as follows.

Training the shared parameters ω of the child models. In this step, we fix the controller's policy $\pi(\mathbf{m}; \theta)$ and perform stochastic gradient descent (SGD) on ω to minimize the expected loss function $\mathbb{E}_{\mathbf{m}\sim\pi}\left[\mathcal{L}(\mathbf{m}; \omega)\right]$. Here, $\mathcal{L}(\mathbf{m}; \omega)$ is the standard cross-entropy loss, computed on a minibatch of training data, with a model $\mathbf{m}$ sampled from $\pi(\mathbf{m}; \theta)$. The gradient is computed using the Monte Carlo estimate

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

where $\mathbf{m}_i$'s are sampled from $\pi(\mathbf{m}; \theta)$ as described above. Eqn 1 provides an unbiased estimate of the gradient $\nabla_\omega \mathbb{E}_{\mathbf{m}\sim\pi(\mathbf{m};\theta)}\left[\mathcal{L}(\mathbf{m}; \omega)\right]$. However, this estimate has a higher variance than the standard SGD gradient, where $\mathbf{m}$ is fixed. Nevertheless – and this is perhaps surprising – 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 $\pi(\mathbf{m}; \theta)$. As mentioned, we train ω during a entire pass through the training data.

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

The reward $\mathcal{R}(\mathbf{m}, \omega)$ is computed on *the validation set*, rather than on the training set, to encourage ENAS to select models that generalize well rather than models that overfit the training set well. In our language model experiment, the reward function is $c/\text{valid\_ppl}$, where the perplexity is computed on a minibatch of validation data. In our image classification experiments, the reward function is the accuracy on a minibatch of validation images.

Deriving Architectures. We discuss how to derive novel architectures from a trained ENAS model. We first sample several models from the trained policy $\pi(\mathbf{m}, \theta)$. For each sampled model, we compute its reward on *a single minibatch* sampled from the validation set. We then take only the model with the highest reward to re-train from scratch. It is possible to improve our experimental results by training all the sampled models from scratch and selecting the model with the highest performance on a separated validation set, as done by other works (Zoph & Le, 2017; Zoph et al., 2018; Liu et al., 2017; 2018). However, our method yields similar performance whilst being much more economical.

2.3. Designing Convolutional Networks

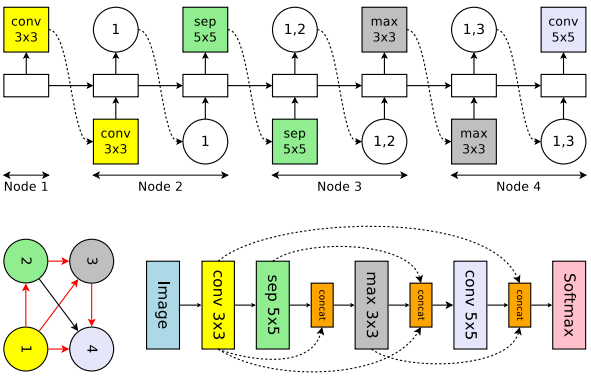

Figure 3. An example run of a recurrent cell in our search space with 4 computational nodes, which represent 4 layers in a convolutional network. *Top:* The output of the controller RNN. *Bottom Left:* The computational DAG corresponding to the network's architecture. Red arrows denote the active computational paths. *Bottom Right:* The complete network. Dotted arrows denote skip connections.

We now discuss the search space for convolutional architectures. Recall that in the search space of the recurrent cell, the controller RNN samples two decisions at each decision block: 1) what previous node to connect to and 2) what activation function to use. In the search space for convolu-

tional models, the controller RNN also samples two sets of decisions at each decision block: 1) what previous nodes to connect to and 2) what computation operation to use. These decisions construct a layer in the convolutional model.

The decision of what previous nodes to connect to allows the model to form skip connections (He et al., 2016a; Zoph & Le, 2017). Specifically, at layer k, up to $k-1$ mutually distinct previous indices are sampled, leading to 2^{k-1} possible decisions at layer k. We provide an illustrative example of sampling a convolutional network in Figure 3. In this example, at layer $k = 4$, the controller samples previous indices $\{1, 3\}$, so the outputs of layers 1 and 3 are concatenated along their depth dimension and sent to layer 4.

Meanwhile, the decision of what computation operation to use sets a particular layer into convolution or average pooling or max pooing. The 6 operations available for the controller are: convolutions with filter sizes 3×3 and 5×5, depthwise-separable convolutions with filter sizes 3×3 and 5×5 (Chollet, 2017), and max pooling and average pooling of kernel size 3×3. As for recurrent cells, each operation at each layer in our ENAS convolutional network has a distinct set of parameters.

Making the described set of decisions for a total of L times, we can sample a network of L layers. Since all decisions are independent, there are $6^L \times 2^{L(L-1)/2}$ networks in the search space. In our experiments, $L = 12$, resulting in 1.6×10^{29} possible networks.

2.4. Designing Convolutional Cells

Rather than designing the entire convolutional network, one can design smaller modules and then connect them together to form a network (Zoph et al., 2018). Figure 4 illustrates this design, where the convolutional cell and reduction cell architectures are to be designed. We now discuss how to use ENAS to search for the architectures of these cells.

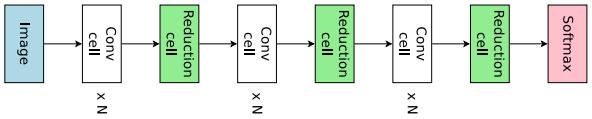

Figure 4. Connecting 3 blocks, each with N convolution cells and 1 reduction cell, to make the final network.

We utilize the ENAS computational DAG with B nodes to represent the computations that happen *locally in a cell*. In this DAG, node 1 and node 2 are treated as the cell's inputs, which are the outputs of the two previous cells in the final network (see Figure 4). For each of the remaining $B - 2$ nodes, we ask the controller RNN to make two sets of decisions: 1) two previous nodes to be used as inputs to the current node and 2) two operations to apply to the two sampled nodes. The 5 available operations are: identity, separable convolution with kernel size 3×3 and 5×5, and average pooling and max pooling with kernel size 3×3. At each node, after the previous nodes and their corresponding operations are sampled, the operations are applied on the previous nodes, and their results are added.

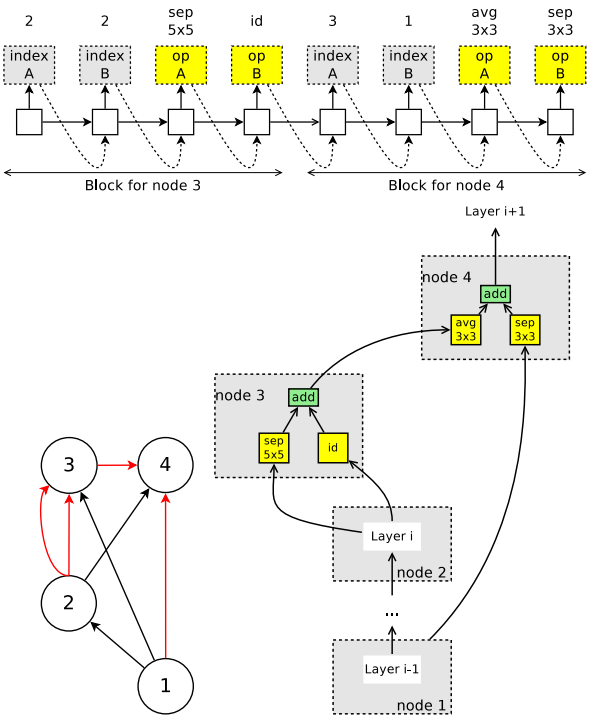

Figure 5. An example run of the controller for our search space over convolutional cells. *Top:* the controller's outputs. In our search space for convolutional cells, node 1 and node 2 are the cell's inputs, so the controller only has to design node 3 and node 4. *Bottom Left:* The corresponding DAG, where red edges represent the activated connections. *Bottom Right:* the convolutional cell according to the controller's sample.

As before, we illustrate the mechanism of our search space with an example, here with $B = 4$ nodes (refer to Figure 5). Details are as follows.

1. Nodes 1, 2 are input nodes, so no decisions are needed for them. Let h_1, h_2 be the outputs of these nodes.
2. At node 3: the controller samples two previous nodes and two operations. In Figure 5 *Top Left*, it samples *node 2*, *node 2*, *separable_conv_5x5*, and *identity*. This means that $h_3 = \text{sep\_conv\_5x5}(h_2) + \text{id}(h_2)$.
3. At node 4: the controller samples *node 3*, *node 1*, *avg_pool_3x3*, and *sep_conv_3x3*. This means that $h_4 = \text{avg\_pool\_3x3}(h_3) + \text{sep\_conv\_3x3}(h_1)$.
4. Since all nodes but h_4 were used as inputs to at least another node, the only loose end, h_4, is treated as the cell's output.

If there are multiple loose ends, they will be concatenated along the depth dimension to form the cell's output.

A reduction cell can also be realized from the search space we discussed, simply by: 1) sampling a computational graph from the search space, and 2) applying all operations with a stride of 2. A reduction cell thus reduces the spatial dimensions of its input by a factor of 2. Following Zoph et al. (2018), we sample the reduction cell conditioned on the convolutional cell, hence making the controller RNN run for a total of $2(B-2)$ blocks.

Finally, we estimate the complexity of this search space. At node i ($3 \leq i \leq B$), the controller can select any two nodes from the $i-1$ previous nodes, and any two operations from 5 operations. As all decisions are independent, there are $(5 \times (B-2)!)^2$ possible cells. Since we independently sample for a convolutional cell and a reduction cell, the final size of the search space is $(5 \times (B-2)!)^4$. With $B = 7$ as in our experiments, the search space can realize 1.3×10^{11} final networks, making it significantly smaller than the search space for entire convolutional networks (Section 2.3).

3. Experiments

We first present our experimental results from employing ENAS to design recurrent cells on the Penn Treebank dataset and convolutional architectures on the CIFAR-10 dataset. We then present an ablation study which asserts the role of ENAS in discovering novel architectures.

3.1. Language Model with Penn Treebank

Dataset and Settings. Penn Treebank (Marcus et al., 1994) is a well-studied benchmark for language model. We use the standard pre-processed version of the dataset, which is also used by previous works, *e.g.* Zaremba et al. (2014).

Since the goal of our work is to discover cell architectures, we only employ the standard training and test process on Penn Treebank, and do not utilize post-training techniques such as neural cache (Grave et al., 2017) and dynamic evaluation (Krause et al., 2017). Additionally, as Collins et al. (2017) have established that RNN models with more parameters can learn to store more information, we limit the size of our ENAS cell to $24M$ parameters. We also do not tune our hyper-parameters extensively like Melis et al. (2017), nor do we train multiple architectures and select the best one based on their validation perplexities like Zoph & Le (2017). Therefore, ENAS is not at any advantage, compared to Zoph & Le (2017); Yang et al. (2018); Melis et al. (2017), and its improved performance is only due to the cell's architecture.

Training details. Our controller is trained using Adam, with a learning rate of 0.00035. To prevent premature convergence, we also use a tanh constant of 2.5 and a temperature of 5.0 for the sampling logits (Bello et al., 2017a;b), and add the controller's sample entropy to the reward, weighted by 0.0001. Additionally, we augment the simple transformations between nodes in the constructed recurrent cell with highway connections (Zilly et al., 2017). For instance, instead of having $h_2 = \text{ReLU}(h_1 \cdot \mathbf{W}_{2,1}^{(\mathbf{h})})$ as shown in the example from Section 2.1, we have $h_2 = c_2 \otimes \text{ReLU}(h_1 \cdot \mathbf{W}_{2,1}^{(\mathbf{h})}) + (1 - c_2) \otimes h_1$, where $c_2 = \text{sigmoid}(h_1 \cdot \mathbf{W}_{2,1}^{(\mathbf{c})})$ and $\otimes$ denotes elementwise multiplication.

The shared parameters of the child models ω are trained using SGD with a learning rate of 20.0, decayed by a factor of 0.96 after every epoch starting at epoch 15, for a total of 150 epochs. We clip the norm of the gradient ∇_ω at 0.25. We find that using a large learning rate whilst clipping the gradient norm at a small threshold makes the updates on ω more stable. We utilize three regularization techniques on ω: an ℓ_2 regularization weighted by 10^{-7}; variational dropout (Gal & Ghahramani, 2016); and tying word embeddings and softmax weights (Inan et al., 2017). More details are in Appendix A.

Results. Running on a single Nvidia GTX 1080Ti GPU, ENAS finds a recurrent cell in about 10 hours. In Table 1, we present the performance of the ENAS cell as well as other baselines that do not employ post-training processing. The ENAS cell achieves a test perplexity of 55.8, which is on par with the existing state-of-the-art of 56.0 achieved by *Mixture of Softmaxes* (MoS) (Yang et al., 2018). Note that we do not apply MoS to the ENAS cell. Importantly, ENAS cell outperforms NAS (Zoph & Le, 2017) by more than 6 perplexity points, whilst the search process of ENAS, in terms of GPU hours, is more than 1000x faster.

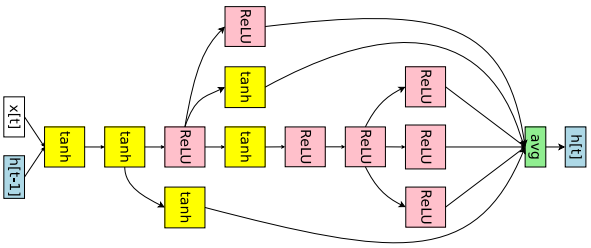

Figure 6. The RNN cell ENAS discovered for Penn Treebank.

Our ENAS cell, visualized in Figure 6, has a few interesting properties. First, all non-linearities in the cell are either ReLU or tanh, even though the search space also has two other functions: identity and sigmoid. Second, we suspect this cell is a local optimum, similar to the observations made by Zoph & Le (2017). When we randomly pick some

Architecture	Additional Techniques	Params (million)	Test PPL
LSTM (Zaremba et al., 2014)	Vanilla Dropout	66	78.4
LSTM (Gal & Ghahramani, 2016)	VD	66	75.2
LSTM (Inan et al., 2017)	VD, WT	51	68.5
LSTM (Melis et al., 2017)	Hyper-parameters Search	24	59.5
LSTM (Yang et al., 2018)	VD, WT, ℓ_2, AWD, MoC	22	57.6
LSTM (Merity et al., 2017)	VD, WT, ℓ_2, AWD	24	57.3
LSTM (Yang et al., 2018)	VD, WT, ℓ_2, AWD, MoS	**22**	**56.0**
RHN (Zilly et al., 2017)	VD, WT	24	66.0
NAS (Zoph & Le, 2017)	VD, WT	54	62.4
ENAS	VD, WT, ℓ_2	**24**	**55.8**

Table 1. Test perplexity on Penn Treebank of ENAS and other baselines. Abbreviations: RHN is *Recurrent Highway Network*, VD is *Variational Dropout*; WT is *Weight Tying*; ℓ_2 is *Weight Penalty*; AWD is *Averaged Weight Drop*; MoC is *Mixture of Contexts*; MoS is *Mixture of Softmaxes*.

nodes and switch the non-linearity into identity or sigmoid, the perplexity increases up to 8 points. Similarly, when we randomly switch some ReLU nodes into tanh or vice versa, the perplexity also increases, but only up to 3 points. Third, as shown in Figure 6, the output of our ENAS cell is an average of 6 nodes. This behavior is similar to that of *Mixture of Contexts* (MoC) (Yang et al., 2018). Not only does ENAS independently discover MoC, but it also learns to balance between i) the number of contexts to mix, which increases the model's expressiveness, and ii) the depth of the recurrent cell, which learns more complex transformations (Zilly et al., 2017).

3.2. 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 channel mean and dividing the channel standard deviation, centrally padding the training images to 40×40 and randomly cropping them back to 32×32, and randomly flipping them horizontally.

Search spaces. We apply ENAS to two search spaces: 1) the *macro search space* over entire convolutional models (Section 2.3); and 2) the *micro search space* over convolutional cells (Section 2.4).

Training details. The shared parameters ω are trained with Nesterov momentum (Nesterov, 1983), where the learning rate follows the cosine schedule with $l_{\text{max}} = 0.05$, $l_{\text{min}} = 0.001$, $T_0 = 10$, and $T_{\text{mul}} = 2$ (Loshchilov & Hutter, 2017). Each architecture search is run for 310 epochs. We initialize ω with He initialization (He et al., 2015). We also apply an ℓ_2 weight decay of 10^{-4}. We train the architectures recommended by the controller using the same settings.

The policy parameters θ are initialized uniformly in $[-0.1, 0.1]$, and trained with Adam at a learning rate of 0.00035. Similar to the procedure in Section 3.1, we apply a tanh constant of 2.5 and a temperature of 5.0 to the controller's logits, and add the controller entropy to the reward, weighted by 0.1. Additionally, in the macro search space, we enforce the sparsity in the skip connections by adding to the reward the KL divergence between: 1) the skip connection probability between any two layers and 2) our chosen probability $\rho = 0.4$, which represents the prior belief of a skip connection being formed. This KL divergence term is weighted by 0.8. More training details are in Appendix B.

Results. Table 2 summarizes the test errors of ENAS and other approaches. In this table, the first block presents the results of DenseNet (Huang et al., 2016), one of the highest-performing architectures that are designed by human experts. When trained with a strong regularization technique, such as Shake-Shake (Gastaldi, 2016), and a data augmentation technique, such as CutOut (DeVries & Taylor, 2017), DenseNet impressively achieves the test error of 2.56%.

The second block of Table 2 presents the performances of approaches that attempt to design an entire convolutional network, along with the the number of GPUs and the time these methods take to discover their final models. As shown, ENAS finds a network architecture, which we visualize in Figure 7, and which achieves 4.23% test error. This test error is better than the error of 4.47%, achieved by the second best NAS model (Zoph & Le, 2017). If we keep the architecture, but increase the number of filters in the network's highest layer to 512, then the test error decreases to 3.87%, which is not far away from NAS's best model, whose test error is 3.65%. Impressively, ENAS takes about 7 hours to find this architecture, reducing the number of

Method	GPUs	Times (days)	Params (million)	Error (%)
DenseNet-BC (Huang et al., 2016)	–	–	25.6	3.46
DenseNet + Shake-Shake (Gastaldi, 2016)	–	–	26.2	2.86
DenseNet + CutOut (DeVries & Taylor, 2017)	–	–	26.2	**2.56**
Budgeted Super Nets (Veniat & Denoyer, 2017)	–	–	–	9.21
ConvFabrics (Saxena & Verbeek, 2016)	–	–	21.2	7.43
Macro NAS + Q-Learning (Baker et al., 2017a)	10	8-10	11.2	6.92
Net Transformation (Cai et al., 2018)	5	2	19.7	5.70
FractalNet (Larsson et al., 2017)	–	–	38.6	4.60
SMASH (Brock et al., 2018)	1	1.5	16.0	4.03
NAS (Zoph & Le, 2017)	800	21-28	7.1	4.47
NAS + more filters (Zoph & Le, 2017)	800	21-28	37.4	**3.65**
ENAS + macro search space	1	0.32	21.3	4.23
ENAS + macro search space + more channels	1	0.32	38.0	**3.87**
Hierarchical NAS (Liu et al., 2018)	200	1.5	61.3	3.63
Micro NAS + Q-Learning (Zhong et al., 2018)	32	3	–	3.60
Progressive NAS (Liu et al., 2017)	100	1.5	3.2	3.63
NASNet-A (Zoph et al., 2018)	450	3-4	3.3	3.41
NASNet-A + CutOut (Zoph et al., 2018)	450	3-4	3.3	**2.65**
ENAS + micro search space	1	0.45	4.6	3.54
ENAS + micro search space + CutOut	1	0.45	4.6	**2.89**

Table 2. Classification errors of ENAS and baselines on CIFAR-10. In this table, the first block presents DenseNet, one of the state-of-the-art architectures designed by human experts. The second block presents approaches that design the entire network. The last block presents techniques that design modular cells which are combined to build the final network.

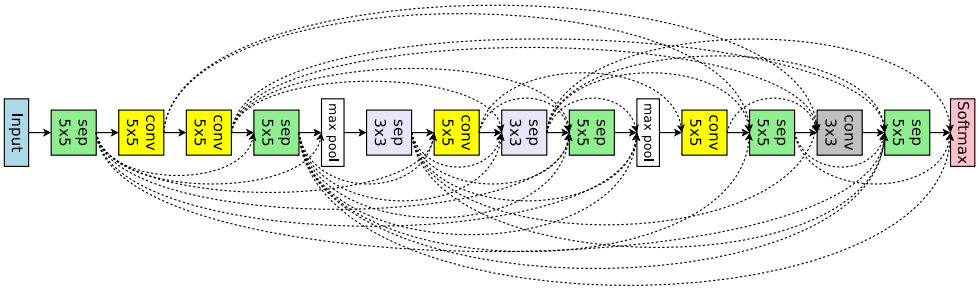

Figure 7. ENAS's discovered network from the macro search space for image classification.

GPU-hours by more than 50,000x compared to NAS.

The third block of Table 2 presents the performances of approaches that attempt to design one more more modules and then connect them together to form the final networks. ENAS takes 11.5 hours to discover the convolution cell and the reduction cell, which are visualized in Figure 8. With the convolutional cell replicated for $N = 6$ times (*c.f.* Figure 4), ENAS achieves 3.54% test error, on par with the 3.41% error of NASNet-A (Zoph et al., 2018). With CutOut (DeVries & Taylor, 2017), ENAS's error decreases to 2.89%, compared to 2.65% by NASNet-A.

In addition to ENAS's strong performance, we also find that the models found by ENAS are, in a sense, the local minimums in their search spaces. In particular, in the model that ENAS finds from the marco search space, if we replace all separable convolutions with normal convolutions, and then adjust the model size so that the number of parameters stay the same, then the test error increases by 1.7%. Similarly, if we randomly change several connections in the cells that ENAS finds in the micro search space, the test error increases by 2.1%. This behavior is also observed when ENAS searches for recurrent cells (*c.f.* Section 3.1), as well as in Zoph & Le (2017). We thus believe that the controller RNN learned by ENAS is as good as the controller RNN learned by NAS, and that the performance gap between NAS and ENAS is due to the fact that we do not sample multiple architectures from our trained controller, train them, and then select the best architecture on the validation data. This extra step benefits NAS's performance.

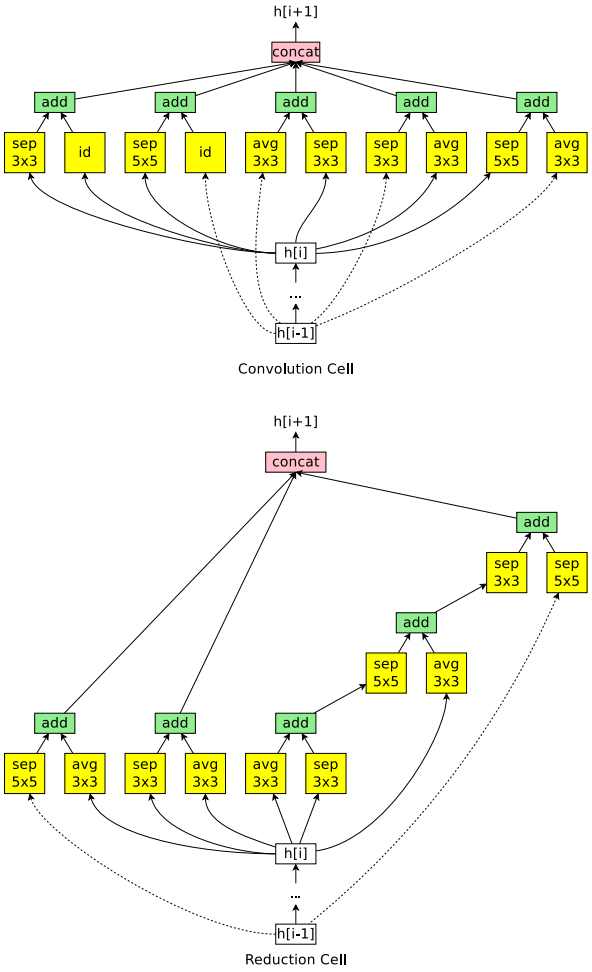

Figure 8. ENAS cells discovered in the micro search space.

3.3. The Importance of ENAS

A question regarding ENAS's importance is whether ENAS is actually capable of finding good architectures, or if it is the design of the search spaces that leads to ENAS's strong empirical performance.

Comparing to Guided Random Search. We uniformly sample a recurrent cell, an entire convolutional network, and a pair of convolutional and reduction cells from their search spaces and train them to convergence using the same settings as the architectures found by ENAS. For the macro space over entire networks, we sample the skip connections with an activation probability of 0.4, effectively balancing ENAS's advantage from the KL divergence term in its reward (see Section 3.2). Our random recurrent cell achieves the test perplexity of 81.2 on Penn Treebank, which is far worse than ENAS's perplexity of 55.8. Our random convolutional network reaches 5.86% test error, and our two random cells reache 6.77% on CIFAR-10, while ENAS achieves 4.23% and 3.54%, respectively.

Disabling ENAS Search. In addition to random search, we attempt to train only the shared parameters ω without updating the controller. We conduct this study for our macro search space (Section 2.3), where the effect of an untrained random controller is similar to dropout with a rate of 0.5 on the skip connections, and to drop-path on the operations (Zoph et al., 2018; Larsson et al., 2017). At convergence, the model has the error rate of 8.92%. On the validation set, an ensemble of 250 Monte Carlo configurations of this trained model can only reach 5.49% test error. We therefore conclude that the appropriate training of the ENAS controller is crucial for good performance.

4. Related Work and Discussions

There is a growing interest in improving the efficiency of NAS. Concurrent to our work are the promising ideas of using performance prediction (Baker et al., 2017b; Deng et al., 2017), using iterative search method for architectures of growing complexity (Liu et al., 2017), and using hierarchical representation of architectures (Liu et al., 2018). Table 2 shows that ENAS is significantly more efficient than these other methods, in GPU hours.

ENAS's design of sharing weights between architectures is inspired by the concept of weight inheritance in neural model evolution (Real et al., 2017; 2018). Additionally, ENAS's choice of representing computations using a DAG is inspired by the concept of stochastic computational graph (Schulman et al., 2015), which introduces nodes with stochastic outputs into a computational graph. ENAS's utilizes such stochastic decisions in a network to make discrete architectural decisions that govern subsequent computations in the network, trains the decision maker, *i.e.* the controller, and finally harvests the decisions to derive architectures.

Closely related to ENAS is SMASH (Brock et al., 2018), which designs an architecture and then uses a hypernetwork (Ha et al., 2017) to generate its weight. Such usage of the hypernetwork in SMASH inherently restricts the weights of SMASH's child architectures to a low-rank space. This is because the hypernetwork generates weights for SMASH's child models via tensor products (Ha et al., 2017), which suffer from a low-rank restriction as for arbitrary matrices $\mathbf{A}$ and $\mathbf{B}$, one always has the inequality: $\text{rank}(\mathbf{A} \cdot \mathbf{B}) \leq \min\{\text{rank}(\mathbf{A}), \text{rank}(\mathbf{B})\}$. Due to this limit, SMASH will find architectures that perform well in the restricted low-rank space of their weights, rather than architectures that perform well in the normal training setups, where the weights are no longer restricted. Meanwhile, ENAS allows the weights of its child models to be arbitrary, effectively avoiding such restriction. We suspect this is the

reason behind ENAS's superior empirical performance to SMASH. In addition, it can be seen from our experiments that ENAS can be flexibly applied to multiple search spaces and disparate domains, *e.g.* the space of RNN cells for the text domain, the macro search space of entire networks, and the micro search space of convolutional cells for the image domain.

5. Conclusion

NAS is an important advance that automatizes the designing process of neural networks. However, NAS's computational expense prevents it from being widely adopted. In this paper, we presented ENAS, a novel method that speeds up NAS by more than 1000x, in terms of GPU hours. ENAS's key contribution is the sharing of parameters across child models during the search for architectures. This insight is implemented by searching for a subgraph within a larger graph that incorporates architectures in a search space. We showed that ENAS works well on both CIFAR-10 and Penn Treebank datasets.

Acknowledgements

The authors want to thank Jaime Carbonell, Zihang Dai, Lukasz Kaiser, Azalia Mirhoseini, Ashwin Paranjape, Daniel Selsam, and Xinyi Wang for their suggestions on improving the paper.

References

Baker, Bowen, Gupta, Otkrist, Naik, Nikhil, and Raskar, Ramesh. Designing neural network architectures using reinforcement learning. In *ICLR*, 2017a.

Baker, Bowen, Otkrist, Gupta, Raskar, Ramesh, and Naik, Nikhil. Accelerating neural architecture search using performance prediction. *Arxiv, 1705.10823*, 2017b.

Bello, Irwan, Pham, Hieu, Le, Quoc V., Norouzi, Mohammad, and Bengio, Samy. Neural combinatorial optimization with reinforcement learning. In *ICLR Workshop*, 2017a.

Bello, Irwan, Zoph, Barret, Vasudevan, Vijay, and Le, Quoc V. Neural optimizer search with reinforcement learning. In *ICML*, 2017b.

Brock, Andrew, Lim, Theodore, Ritchie, James M., and Weston, Nick. SMASH: one-shot model architecture search through hypernetworks. *ICLR*, 2018.

Cai, Han, Chen, Tianyao, Zhang, Weinan, Yu, Yong., and Wang, Jun. Efficient architecture search by network transformation. In *AAAI*, 2018.

Chollet, Francois. Xception: Deep learning with depthwise separable convolutions. In *CVPR*, 2017.

Collins, Jasmine, Sohl-Dickstein, Jascha, and Sussillo, David. Capacity and trainability in recurrent neural networks. In *ICLR*, 2017.

Deng, Boyang, Yan, Junjie, and Lin, Dahua. Peephole: Predicting network performance before training. *Arxiv, 1705.10823*, 2017.

DeVries, Terrance and Taylor, Graham W. Improved regularization of convolutional neural networks with cutout. *Arxiv, 1708.04552*, 2017.

Gal, Yarin and Ghahramani, Zoubin. A theoretically grounded application of dropout in recurrent neural networks. In *NIPS*, 2016.

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

Grave, Edouard, Joulin, Armand, and Usunier, Nicolas. Improving neural language models with a continuous cache. In *ICLR*, 2017.

Ha, David, Dai, Andrew, and Le, Quoc V. Hypernetworks. In *ICLR*, 2017.

He, Kaiming, Zhang, Xiangyu, Rein, Shaoqing, and Sun, Jian. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *CVPR*, 2015.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Deep residual learning for image recognition. In *CPVR*, 2016a.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, and Sun, Jian. Identity mappings in deep residual networks. In *CPVR*, 2016b.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. In *Neural Computations*, 1997.

Huang, Gao, Liu, Zhuang, van der Maaten, Laurens, and Weinberger, Kilian Q. Densely connected convolutional networks. In *CVPR*, 2016.

Inan, Hakan, Khosravi, Khashayar, and Socher, Richard. Tying word vectors and word classifiers: a loss framework for language modeling. In *ICLR*, 2017.

Ioffe, Sergey and Szegedy, Christian. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, 2015.

Kingma, Diederik P. and Ba, Jimmy Lei. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Krause, Ben, Kahembwe, Emmanuel, Murray, Iain, and Renals, Steve. Dynamic evaluation of neural sequence models. *Arxiv, 1709.07432*, 2017.

Krizhevsky, Alex. Learning multiple layers of features from tiny images. Technical report, 2009.

Larsson, Gustav, Maire, Michael, and Shakhnarovich, Gregory. Fractalnet: Ultra-deep neural networks without residuals. In *ICLR*, 2017.

Lin, Min, Chen, Qiang, and Yan, Shuicheng. Network in network. *Arxiv, 1312.4400*, 2013.

Liu, Chenxi, Zoph, Barret, Shlens, Jonathon, Hua, Wei, Li, Li-Jia, Fei-Fei, Li, Yuille, Alan, Huang, Jonathan, and Murphy, Kevin. Progressive neural architecture search. *Arxiv, 1712.00559*, 2017.

Liu, Hanxiao, Simonyan, Karen, Vinyals, Oriol, Fernando, Chrisantha, and Kavukcuoglu, Koray. Hierarchical representations for efficient architecture search. In *ICLR*, 2018.

Loshchilov, Ilya and Hutter, Frank. Sgdr: Stochastic gradient descent with warm restarts. In *ICLR*, 2017.

Luong, Minh-Thang, Le, Quoc V., Sutskever, Ilya, Vinyals, Oriol, and Kaiser, Lukasz. Multi-task sequence to sequence learning. In *ICLR*, 2016.

Marcus, Mitchell, Kim, Grace, Marcinkiewicz, Mary Ann, MacIntyre, Robert, Bies, Ann, Ferguson, Mark, Katz, Karen, and Schasberger, Britta. The penn treebank: Annotating predicate argument structure. In *Proceedings of the Workshop on Human Language Technology*, 1994.

Melis, Gábor, Dyer, Chris, and Blunsom, Phil. On the state of the art of evaluation in neural language models. *Arxiv, 1707.05589*, 2017.

Merity, Stephen, Keskar, Nitish Shirish, and Socher, Richard. Regularizing and optimizing LSTM language models. *Arxiv, 1708.02182*, 2017.

Negrinho, Renato and Gordon, Geoff. Deeparchitect: Automatically designing and training deep architectures. In *CPVR*, 2017.

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

Razavian, Ali Sharif, Azizpour, Hossein, Josephine, Sullivan, and Carlsson, Stefan. Cnn features off-the-shelf: an astounding baseline for recognition. In *CVPR*, 2014.

Real, Esteban, Moore, Sherry, Selle, Andrew, Saxena, Saurabh, Leon, Yutaka Suematsu, Tan, Jie, Le, Quoc, and Kurakin, Alex. Large-scale evolution of image classifiers. In *ICML*, 2017.

Real, Esteban, Aggarwal, Alok, Huang, Yanping, and Le, Quoc V. Peephole: Predicting network performance before training. *Arxiv, 1802.01548*, 2018.

Saxena, Shreyas and Verbeek, Jakob. Convolutional neural fabrics. In *NIPS*, 2016.

Schulman, John, Heess, Nicolas, Weber, Theophane, and Abbeel, Pieter. Gradient estimation using stochastic computation graphs. In *NIPS*, 2015.

Veniat, Tom and Denoyer, Ludovic. Learning time-efficient deep architectures with budgeted super networks. *Arxiv, 1706.00046*, 2017.

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

Yang, Zhilin, Dai, Zihang, Salakhutdinov, Ruslan, and Cohen, William. Breaking the softmax bottleneck: A high-rank rnn language model. In *ICLR*, 2018.

Zaremba, Wojciech, Sutskever, Ilya, and Vinyals, Oriol. Recurrent neural network regularization. *Arxiv, 1409.2329*, 2014.

Zhong, Zhao, Yan, Junjie, and Liu, Cheng-Lin. Practical network blocks design with q-learning. *AAAI*, 2018.

Zilly, Julian Georg, Srivastava, Rupesh Kumar, Koutník, Jan, and Schmidhuber, Jürgen. Recurrent highway networks. In *ICML*, 2017.

Zoph, Barret and Le, Quoc V. Neural architecture search with reinforcement learning. In *ICLR*, 2017.

Zoph, Barret, Yuret, Deniz, May, Jonathan, and Knight, Kevin. Transfer learning for low-resource neural machine translation. In *EMNLP*, 2016.

Zoph, Barret, Vasudevan, Vijay, Shlens, Jonathon, and Le, Quoc V. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018.

Appendix for:
Efficient Neural Architecture Search via Parameter Sharing

A. Details on Penn Treebank Experiments

Computations in an RNN Cell. We think of the cell at time step t as a DAG with N computational nodes, indexed by $\mathbf{h}_1^{(t)}$, $\mathbf{h}_2^{(t)}$, ... $\mathbf{h}_N^{(t)}$. Node $\mathbf{h}_1^{(t)}$ receives two inputs: 1) the RNN signal $\mathbf{x}^{(t)}$ at its current time step; and 2) the output $\mathbf{h}_D^{(t-1)}$ from the cell at the previous time step. The following computations are performed:

$$\mathbf{c}_1^{(t)} \leftarrow \text{sigmoid}\left(\mathbf{x}^{(t)} \cdot \mathbf{W}^{(\mathbf{x},\mathbf{c})} + \mathbf{h}_N^{(t-1)} \cdot, \mathbf{W}_0^{(\mathbf{c})}\right) \quad (2)$$

$$\mathbf{h}_1^{(t)} \leftarrow \mathbf{c}_1^{(t)} \otimes f_1\left(\mathbf{x}^{(t)} \cdot \mathbf{W}^{(\mathbf{x},\mathbf{h})} + \mathbf{h}_N^{(t-1)} \cdot \mathbf{W}_1^{(\mathbf{h})}\right) + (1 - \mathbf{c}_1^{(t)}) \otimes \mathbf{h}_N^{(t-1)}, \quad (3)$$

where f_1 is an activation function that the controller will decide. For $\ell = 2, 3, ..., N$, node $\mathbf{h}_\ell$ receives its input from a layer $j_\ell \in \{\mathbf{h}_1, ..., \mathbf{h}_{\ell-1}\}$, which is specified by the controller, and then performs the following computations:

$$\mathbf{c}_\ell^{(t)} \leftarrow \text{sigmoid}\left(\mathbf{h}_{j_\ell}^{(t)} \cdot \mathbf{W}_{\ell,j_\ell}^{(\mathbf{c})}\right) \quad (4)$$

$$\mathbf{h}_\ell^{(t)} \leftarrow \mathbf{c}_\ell^{(t)} \otimes f_\ell\left(\mathbf{h}_{j_\ell}^{(t)} \cdot \mathbf{W}_{\ell,j_\ell}^{(\mathbf{h})}\right) + (1 - \mathbf{c}_\ell^{(t)}) \otimes \mathbf{h}_{j_\ell}^{(t)}. \quad (5)$$

Therefore, the shared parameters ω among different recurrent cells consist of all the matrices $\mathbf{W}^{(\mathbf{x},\mathbf{c})}$, $\mathbf{W}^{(\mathbf{x},\mathbf{h})}$, $\mathbf{W}_{\ell,j}^{(\mathbf{c})}$, $\mathbf{W}_{\ell,j}^{(\mathbf{h})}$, word embeddings, and the softmax weights if they are not tied with the word embeddings. The controller decides the connection j_ℓ and the activation function f_ℓ for each $\ell \in \{2, 3, ..., N\}$. The layers that are never selected by any subsequent layers are averaged and sent to a softmax head, or to higher recurrent layers.

Parameters Initialization. Our controller's parameters θ are initialized uniformly in $[-0.1, 0.1]$. We find that for Penn Treebank, ENAS quite insensitive to its initialization than for CIFAR-10. Meanwhile, the shared parameters ω are initialized uniformly in $[-0.025, 0.025]$ during architecture search, and $[-0.04, 0.04]$ when we train a fixed architecture recommended by the controller.

Stabilizing the Updates of ω. To stabilize the updates of ω, during the architectures search phase, a layer of batch normalization (Ioffe & Szegedy, 2015) is added immediately after the average of these layers, before the average are sent out of the cell as its output. When a fixed cell is sampled by the controller, we find that we can remove the batch normalization layer without any loss in performance.

B. Details on CIFAR-10 Experiments

We find the following tricks crucial for achieving good performance with ENAS. Standard NAS (Zoph & Le, 2017; Zoph et al., 2018) rely on these and other tricks as well.

Structure of Convolutional Layers. Each convolution in our model is applied in the order of relu-conv-batchnorm (Ioffe & Szegedy, 2015; He et al., 2016b). Additionally, in our micro search space, each depthwise separable convolution is applied twice (Zoph et al., 2018).

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 that the number of output channels does not change between different architectures. When a fixed architecture is sampled, we find that one can remove these batch normalization layers to save computing time and parameters of the final model, without sacrificing significant performance.

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.

The last two tricks are extremely important, since the gradient updates of the shared parameters ω, as described in Eqn 1, have very high variance. In fact, we find that without these two tricks, the training of ENAS is very unstable.