Searching Efficient Neural Architectures using Reinforcement Learning

Tharun Mohandoss (tm35848)* and **Surya S Dwivedi (ssd982)***

Abstract

Designing good neural architectures for a particular task has proven to be challenging task even for domain experts but existing research show that manual effort can be eliminated by automating this process using reinforcement learning techniques. (Zoph and Le, 2016) propose a method of learning neural architectures, where they optimize for the accuracy of prediction of the architecture. In this work, we try to extend their approach to optimize for a combination of objectives such as memory usage and latency along with accuracy so that the resulting neural architectures are both, accurate and 'efficient'. Such architectures would be suitable for deployment in embedded and mobile devices with memory and compute constraints. We use the CIFAR-10 dataset for our experiments, and compare the architectures learned by using just accuracy and by using a combination of multiple objectives, as mentioned above. We trained a RNN to predict neural architectures with our modified reward and also a baseline with just accuracy as the reward(they way (Zoph and Le, 2016) do it). The modified RNN produced an architecture that had 33% less no of parameters as compared to the baseline's architecture with negligible difference in accuracy!

1 Introduction

Deep Neural Networks have been successfully applied in many fields like speech recognition, machine translation, and image recognition. Earlier, the challenge in solving these tasks was hand designing good features for machine learning but now it has been replaced by the simpler task of designing neural architectures. Though simpler, it still takes a lot of time and expert knowledge to design good performing neural architectures. (Zoph and Le, 2016) propose a reinforcement learning based method for searching good performing neural architectures on the CIFAR-10 dataset. The 'good' here is defined just by the accuracy of the resulting model. Designing neural architectures which also optimize other objectives like memory usage and execution latency can be a difficult task even for an expert. Such methods come under the general technique of 'model compression' which is basically used for deploying neural networks on devices with resource constraints. Instead of applying conventional model compression, which relies on hand-crafting features and domain knowledge, we try to extend the work of (Zoph and Le, 2016) to automate this process. Our basic approach remains the same, we use a policy gradient based approach to teach our policy network to produce good architectures. We use an RNN as our policy network and predict the parameters(which are actions really, in the RL sense) layer by layer. The motivation to use an RNN as the policy network is twofold. Firstly, it helps us generate variable length neural architectures. Secondly, what parameters (or actions) we choose in some step depends on what parameters(or actions) we took previously, i.e., our state here at the current time step is determined by the previous actions we took. Finally, the approach is as follows: we sample an architecture from our policy network(the RNN controller), train it on a dataset, obtain a reward and then apply a policy gradient update on our policy network using this sampled roll out. More details on how we sample and actually update the policy-network are in the method section.
[1] [2] [3] [4]

2 Related Work

One of the early works in neural architecture search (S Whiteson, 2006) uses neuro evolutionary tech-

[1]* equal contribution

[2]TM: tharun96@utexas.edu SSD: surya1997@utexas.edu

[3]video: https://youtu.be/YKvmAgdzBg8

[4]code: github.com/TharunMohandoss/EfficientNeuralArchitectureSearch

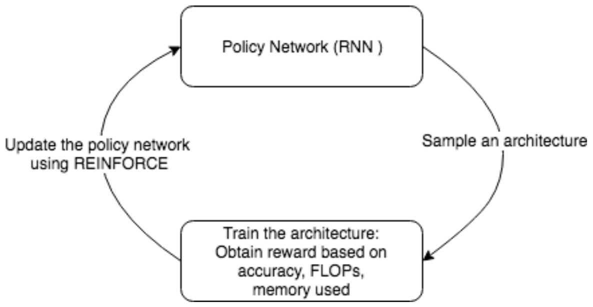

Policy Network (RNN)
Sample an architecture
Train the architecture: Obtain reward based on accuracy, FLOPs, memory used
Update the policy network using REINFORCE

Figure 1: Neural Architecture Search using RL

niques to search for good function approximators for simple tasks such as mountain car and server job scheduling. Such methods rely on clever genetic encoding of neural networks that can facilitate crossovers which is necessary for genetic evolution. The advantage of this method is that along with architectures that perform well, this method also provides initial weight which performs well on this task. But designing such an encoding function for today's neural networks such as Convolutional neural networks will be extremely difficult. Moreover evolutionary algorithms rely heavily on high number of trial and errors and training even one CNN is computationally expensive even for simple tasks such as CIFAR and MNIST. For example, training a 5 layer convolutional network for CIFAR-10 completely takes 30 minutes on a state of the art GPU!.

A more recent work (Liu et al., 2017), shows a simple and much more efficient evolutionary neural architecture search method. They represent an architecture as a single acyclic graph of primitive operations. Such a representation is hierarchical. They use 200 GPUs and show how even simple techniques such as random search when done using this representation can result in state of the art classification architectures. Just like other works in this field, they consider just the accuracy while searching for a good architecture.

(He et al., 2018) propose a method for producing compressible and computer efficient architectures for image-net classification. They use reinforcement learning to sample the neural architecture design space in an efficient manner. Their objective is the most similar to our objective since they optimize not only optimizer for the resulting accuracy but also for the latency of a forward pass and also the total space taken by the model. Yet, they differ from our objective in an important way i.e we do not consider model compression. While model compression is an simple way to reduce the space consumption of a neural network model parameter set, image compression techniques with a good compression rate often take a long time to decompress beating our latency objective. In case the model is decompressed once for being used several times, this can be a useful strategy but it fails to meet the criteria if it requires decompression it is run.

We adopt the strategy introduced in (Zoph and Le, 2016). They generate a neural network layer by layer using a recurrent neural network to predict each layer's hyper-parameter values. For each layer they predict kernel height, kernel width, number of kernels, stride height and stride width. Each choice is considered as an action, and each action takes us to a new state of a partially constructed neural network and the last action is given a reward equal to the accuracy of the resulting neural network on a task. This RNN is trained using policy gradient, specifically using REINFORCE with baseline algorithm. One of the advantages of this method as described in their paper is that this RNN can be trained using techniques similar to federated learning where updates to the neural network are and applied independently to each other. Additionally it is also possible to train this RNN to predict skip layers. We modify their approach by making a small change. Instead of simply using the child network accuracy in the reward function, we use a weighted combination of accuracy and other parameters indicative of forward pass latency and memory consumption of the generated/predicted child neural network.

(Liu et al., 2018a) and (Pham et al., 2018) are extension to previous mentioned work. The second paper shows a more efficient way to sample neural networks with the goal of reducing the total computation cost/time of search without sacrificing too much on the final accuracy. The first one employs a sequential model based optimization strategy to reduce the number of architectures samples by 5 times while still achieving state of the art accuracy.

(Wu et al., 2018) explore a similar goal and show that it is important to optimize a neural network model to a particular device and that neural networks that have low latecy in one device might have much higher latency in other devices. This is a differential architecture search method.

(Liu et al., 2018b) were this first to introduce the differential architecture search technique i.e they they form a continuous representation for a neural architecture and directly apply gradient ascent/evolutionary algorithms on this representation to reach good neural network architectures. This facilitates orders of magnitude faster neural architecture search.

Other related works include (Tan et al., 2019), (Ashok et al., 2017) and (Dai et al., 2019).

3 Method

3.1 Dataset choice

Considering that one might need to sample hundreds or thousands of neural network architectures while searching, most of the works in neural architecture search run on datasets such as MNIST or CIFAR-10. Considering that we have limited time and resources available, we also evaluate our method on the CIFAR-10 dataset.

3.2 Policy Network: RNN

We use an RNN as our policy network. For each layer the network outputs probabilities of all possible values for 3 parameters: kernel size, number of filters and stride and, an additional stop parameter which outputs probability of stopping the sampling after a given layer. The architecture is shown in figure 2. For each value of each of the parameters we have a predefined random embedding of dimension n_h, the dimension of the hidden state of the RNN which we have taken as 35. To sample an architecture from our RNN, we have an initial hidden state h_0, which is used to compute the probabilities of values for the first parameter of the first layer. We sample from this distribution of values, feed the embedding of the sampled value to the next time step , compute probabilities, sample again and continue so on until we sample a '0' from the stop parameter:

$$A_t \sim softmax(\boldsymbol{w_1}.h_{t-1} + \boldsymbol{b_1}) \qquad (1)$$

$$h_t = ReLU(\boldsymbol{w_2}.embedding(A_{t-1}) + \boldsymbol{w_3}.h_{t-1} + \boldsymbol{b_2}) \qquad (2)$$

where w_1,w_2 and w_3 (h x h matrices), and b_1 and b_2 (h x 1 vectors) are the parameters of the RNN. At this point, we construct a CNN with the sampled parameters and train it to obtain the accuracy, memory usage and FLOPs(Floating point operations per second). These are then used for calculating the reward for this sample.

3.3 Child Network Manager

Our child network manager constructs a Convolutional Neural Network from the values sampled using the policy network. As mentioned above, for each layer we sample 3 parameters: kernel size, number of filters and stride in addition to the stop parameter. We took the following possible values for each of the parameters:

Parameters	Possible Values
No of filters	4, 8, 16, 32
Kernel Size	1x1, 3x3, 5x5, 7x7
Stride	1, 2, 4

The number of layers in the resulting CNN is same as the number of layers for which the above 3 parameters are sampled before sampling a '0' for the stop parameter. Each layer has kernel size, no of filters and stride same as the sampled value. The final layer is followed by a fully connected layer and softmax which predicts the probabilities of the 10 classes in CIFAR. We train the resulting CNN on CIFAR-10 dataset for 50 epochs using Adam optimizer with learning rate $= 10e-3$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. We then obtain three values for this architecture: accuracy on validation dataset, FLOPS and number of parameters. We take a weighted average of these three values and use that as our reward, as described in the next section. Also, instead of discarding this architecture, we store its parameters and also the reward obtained, so that we can use it later for making more updates to our policy network.

3.4 Policy Gradient Algorithm

Our goal is to maximize the following objective:

$$J(\theta_p) = E_{p(a_{1:T};\theta_p)}[R] \qquad (3)$$

, where θ_p is the parameter vector for the policy network(RNN in our case), $a_{1:T}$ is the actions predicted by our RNN at each time step and R is the reward obtained after training the child network. For our problem we define the reward as follows:

$$R = accuracy - \lambda_1 * num\_params - \lambda_2 * FLOPs \qquad (4)$$

where we choose $\lambda_1 = 10e-9$ and $\lambda_2 = 2.5 * 10e-7$ so as to bring num_params and $FLOPs$ in the scale of accuracy. We use the REINFORCE rule due to Williams for computing the gradient

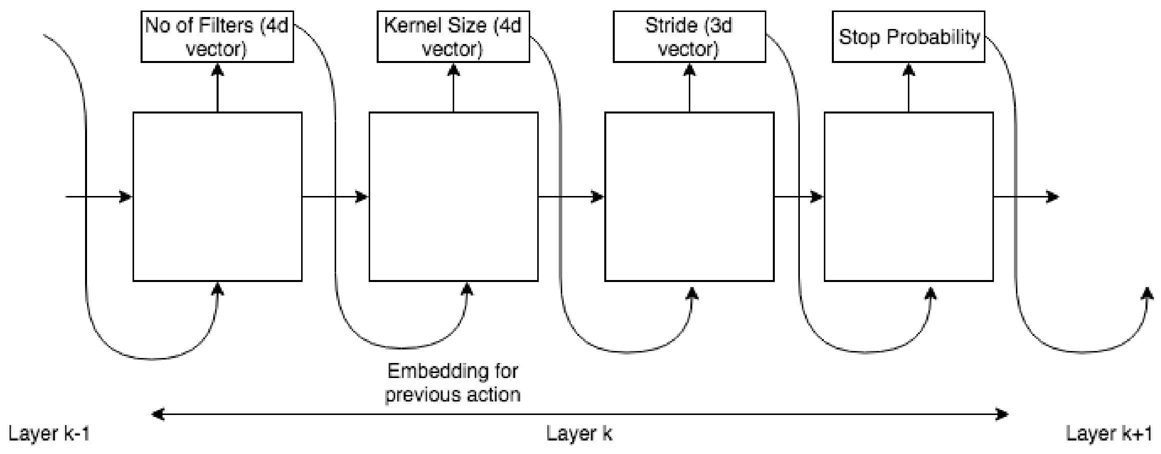


Figure 2: Architecture for our policy network (RNN)

of our objective, i.e $\nabla J(\theta_p)$ can be computed as follows:

$$\nabla J(\theta_p) = \sum_{t=1}^{T} [\nabla \log P(a_t | a_{1:t-1}; \theta_p) R] \quad (5)$$

For calculating the term at time step t, the input to the current layer is the embedding of the previous action and the hidden state. Then the gradient of the cross-entropy loss between the current prediction and a one hot vector with the label for the sampled action at current time step set as one is exactly what we need, i.e. gradient of the log probability of the sampled action w.r.t RNN parameters. Thus for each sampled architecture, we calculate the gradient at each time step as above, take the sum for all time steps, and update the RNN parameters.

3.5 Experience Replay

We also keep storing all sampled architectures and the reward obtained on training. This serves two purposes, first if our controller RNN predicts an architecture that we've already trained once before, we can just lookup for the reward from the stored dictionary. Second, since we have limited amount of training data since we are working on around 1-4 GPUs, every data point that we sample is precious and hence we reuse data from previous batches by using a modified form a experience replay. Hence in our training algorithm, we first sample and evaluate a new architecture every epoch and train the controller one step on 10 randomly chosen examples from the previously trained list of architectures.

4 Experiments

We conduct a detailed empirical analysis of the method described above.

4.1 Training the RNN

We initiated two trainings simultaneously. We train an RNN that only considers accuracy and another one that also considers latency and other factors. The first RNN sampled 82 architecture over two days of training and the best architecture generated by this RNN was 0.6353. The second RNN also showed similar run statistics and it same time sampled an architecture with extremely similar accuracy, FLOP count and number of parameters. 75% of the architectures generated by these RNNs were less than or equal to two layers in size excluding the final fully connected layer. The architecture that achieved 0.6353 was an architecture with a single convolutional layer with a filter size of 7, no of filters = 32 and stride = 2. There was only one architecture sample with 4 layers.

4.2 Normalization to encourage exploration

We noticed that the policy RNN started over-fitting in the initial stages of training so much so that models that had a below average had near zero chances of being sampled. For example, the probability of sampling the best performing architecture with one layer was orders of magnitude higher than some of the multi conv layer architectures. This hindered exploration and we decided to use the method used in (Haarnoja et al., 2018) to encourage exploration. We added a Hessian term(calculate the entropy of action probabilities at each prediction for each hyper-parameter of each layer including the stop parameter) to the RNN's loss along with the REIN-

FORCE loss. The resulting loss looked as follows. We take c1 = 0.05 here.

$$TotalLoss = -\nabla J(\theta_p) + c1 * entropy \quad (6)$$

This loss turned out be extremely unstable since the entropy value goes to infinity if any of the probability values turn out to be 0.

Hence, we normalized in a different way. To the output of the soft-max layer denoting action probabilities, we add a constant(0.05) to each output and re-normalize them to ensure that none of the values become too close to zero so as to encourage exploration.

4.3 Training a simpler RNN with a baseline

We trained the RNN network for four days using the manual normalization technique explained above but due to the small number of training examples at the initial stages of the training causing the network to get stuck in a local minima always producing architectures with just 1 or 2 layers. At this point, since our final goal is to achieve a neural network architecture on CIFAR-10 that is memory and compute efficient without sacrificing too much on the accuracy, we instead use a much simpler RNN that predicts just the kernel size and no of filters for each layer and the values possible for each of these parameters in the same as described in the method section. We have fixed the stride width and stride height to 1. In this simple implementation we fix the number of layers in the neural network to five. As a baseline, we also repeat the same training with just the accuracy as the reward. We compare the best models generated during the course of the two training regimes i.e one with just accuracy as the reward and the other with reward as a linear combination of accuracy, latency and memory measures. Figure 3 shows the highest accuracy among all the networks sampled by the baseline RNN as a function of the number of models sampled. The baseline RNN sampled a total of 216 unique neural network architectures in 4 days of training. Initially the neural network sampled by the model average at around 0.68 accuracy and then slowly improves in performance. The best architecture sampled by this baseline RNN is 0.7245. The forward propagation latency for this neural network is 90800128 Flops and it has a total of 239642 learn-able parameters. This architecture is detailed in Figure 4. This architecture follows one of the well known heuristics that human beings use to generate neural network architectures i.e increase the number of filters/kernels initially and then decrease them near the end. But it violates one other heuristic that favors a higher kernel_size in the initial layers compared to later layers. We henceforth refer to this architecture as model 1.

Figure 5 shows the highest accuracy among all the networks sampled by our modified RNN as a function of number of models sampled. The trend of learning is extremely similar to the baseline RNN wherein initially the model samples somewhat lower accuracy and slowly learns to produce better models. Given the exact same amount of training time, the modified RNN samples 208 unique neural network architectures and the best architectures(accuracy wise) achieved by this model has a validation accuracy of 0.717. The

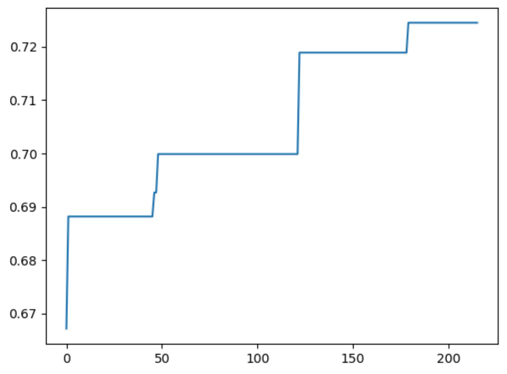
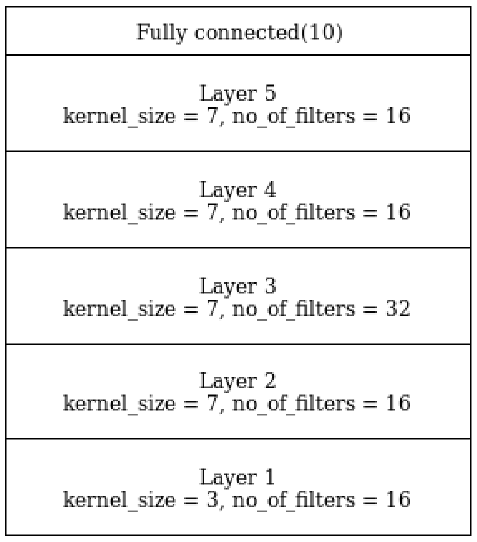


Figure 3: Highest validation accuracy vs number of architectures sampled by the baseline RNN


Fully connected(10)
Layer 5
kernel_size = 7, no_of_filters = 16
Layer 4
kernel_size = 7, no_of_filters = 16
Layer 3
kernel_size = 7, no_of_filters = 32
Layer 2
kernel_size = 7, no_of_filters = 16
Layer 1
kernel_size = 3, no_of_filters = 16
Figure 4: Model 1 : The architecture of the best model sampled by the modified RNN that achieves 0.7245 validation accuracy

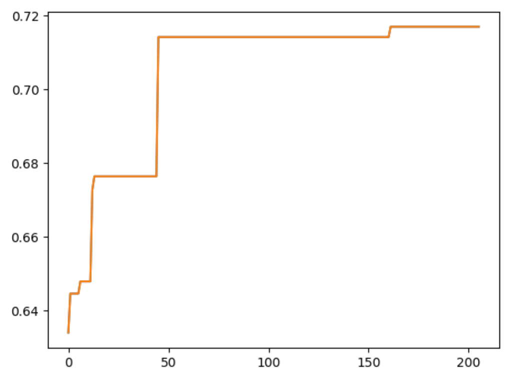


Figure 5: Highest validation accuracy vs number of architectures sampled by the modified RNN.

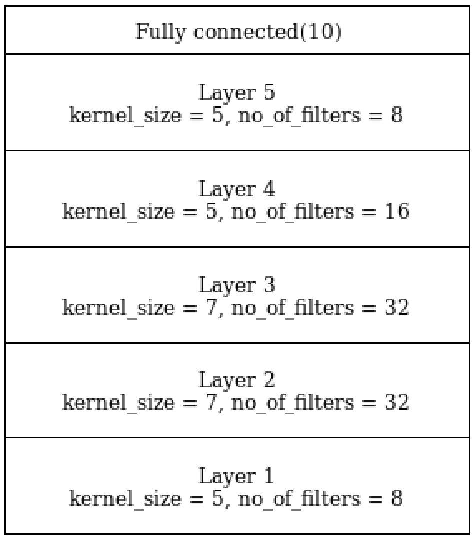

Fully connected(10)
Layer 5
kernel_size = 5, no_of_filters = 8
Layer 4
kernel_size = 5, no_of_filters = 16
Layer 3
kernel_size = 7, no_of_filters = 32
Layer 2
kernel_size = 7, no_of_filters = 32
Layer 1
kernel_size = 5, no_of_filters = 8

Figure 6: Model 2 : The architecture of the best model sampled by the modified RNN that achieves 0.717 validation accuracy

forward propagation latency for this architecture is 84770816 Flops and it has a total of 161346 parameters. This architecture is detailed in Figure 6. This architecture shows a similar trend in following human heuristics to design good architectures. We henceforth refer to this architecture as model 2.

Notice that the best architecture produced by the modified RNN i.e model 2 has almost the same accuracy as the best architecture produced the baseline RNN i.e model 1 inferior only by 0.75%(0.7245 vs 0.717) accuracy without just around 66%(239642 vs 161346) the number of learn-able parameters as the other model. This amounts a direct saving of 33% in memory requirement with negligible loss in accuracy! We have ignore considering the latency time in this comparison since the latency values for both are almost equal with the model 2 being around 10% faster in terms of FLOPS. This saving in memory is mainly a result of the differing size of inputs to the final dense layer. In model 1, the input to the dense layer is of size 32*32*16 whereas in model2 it is of size 32*32*8 which is essentially half the size. Model 2 also saves some parameters on the convolutional layers where it has a smaller kernel size and also uses lesser number of filters compared to model 1 except in the first layer.

4.4 Merits and Demerits of our approach

Our approach makes it possible to search for neural architectures optimizing both accuracy as well as latency. It is simple to understand, implement and modify. The main disadvantage of our approach is that the training over-fits when we allow variable layer size and we haven't been able to solve that issue succesfully.

5 Conclusion

With increasingly fast GPUs and better hardware resources, neural architecture search has become feasible and had gained traction recently. While usually the goal of such methods is simply to produce an architecture with good accuracy, we consider goals such as forward propagation latency and memory requirement as well. We propose a modification of (Zoph and Le, 2016) to achieve our goals and conduct experiements to demonstrate the effectiveness of our methods.

References

Anubhav Ashok, Nicholas Rhinehart, Fares Beainy, and Kris M Kitani. 2017. N2n learning: Network to network compression via policy gradient reinforcement learning. *arXiv preprint arXiv:1709.06030*.

Xiaoliang Dai, Peizhao Zhang, Bichen Wu, Hongxu Yin, Fei Sun, Yanghan Wang, Marat Dukhan, Yunqing Hu, Yiming Wu, Yangqing Jia, et al. 2019. Chamnet: Towards efficient network design through platform-aware model adaptation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 11398–11407.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*.

Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. Amc: Automl for model compression and acceleration on mobile devices. In *The European Conference on Computer Vision (ECCV)*.

Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2018a. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34.

Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. 2017. Hierarchical representations for efficient architecture search. *CoRR*, abs/1711.00436.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018b. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*.

Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. 2018. Efficient neural architecture search via parameter sharing. *arXiv preprint arXiv:1802.03268*.

Peter Stone S Whiteson. 2006. Evolutionary function approximationfor reinforcement learning. *Journal of Machine Learning Research*.

Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V Le. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2820–2828.

Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2018. Fbnet: Hardware-aware efficient convnet design via differentiable neural architecture search.

Barret Zoph and Quoc V Le. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.