AutoML on Keywords Spotting

Yundong Zhang [1] **Chang Yue** [1] **Mengwei Liu** [1]

Abstract

In this project, we explored theoretical backgrounds and strategies of the design and implementation of a system that can automatically achieve efficient and stable hyper-parameters searching. We want to have the structure of a good Neural Network (NN) model for keyword spotting. After fixing the number of layers of the NN, we pick a set of hyper-parameters that the NN is allowed to have per layer. The point of this project is to accelerate hyper-parameter tuning: we implemented an Recurrent Neural Network (RNN) Controller that can output hyper-parameters of its child NN (basically the NN we are interested in for our keyword spotting task) that can yield very high test accuracy under certain constraints, only after seeing a very small fraction of all possible models.

1. Introduction

Machine learning models typically involves a large number of hyper-parameters. In (van Rijn & Hutter, 2017; Ashrafi et al., 2015), it is stated that selections of hyper-parameter is very important for the success of ML algorithms. Those hyper-parameters can both lie in the process of feature engineering and in optimizer training. For decades, the choice of hyper-parameters are heuristically decided by human with domain knowledge through many experiments. With the emerge of Deep Neural Network models in various applications, such as image recognition (Krizhevsky et al., 2012), speech recognition (Hinton et al., 2012) and machine translation (Cho et al., 2014), the paradigm for ML designer has shifted from feature engineering to architecture designing (Zoph & Le, 2016). Although the efforts required in feature engineering have decreased significantly, time needed in hyper-parameter tuning grows significantly, since they are everywhere in deep models. Another major challenge for hyper-parameter tuning in deep models is that each training trail can take many hours and days, requiring considerably amount of GPU resources.

With all these challenges in hand, specifically for deep neural nets, researchers start to look for a more elegant, time-efficient and resource-friendly way to design deep models. Recently, researchers in Google (Zoph & Le, 2016) presented a novel method for hyper-parameter tuning using reinforcement learning, named Neural Architecture Search (NAS). NAS is a gradient-based method, in which a trainable parametrized controller (typically an RNN network) is employed to predict the child neural network (NN that we are actually interested in). The child network is then evaluated on a validation set, from which a reward signal based on the evaluation metrics is generated and fed back to the controller. The controller then updates its parameter and learns to predict good architecture gradually. Based on the results from (Zoph & Le, 2016), NAS is able to predict models from scratch, which is an exciting property that all the previous methods do not have. However, knowing that NAS requires a huge amount of computation resources, we will only focus on utilizing NAS for hyper-parameter tuning in a specific task.

Problem Statement. Based on the scope of this course and computation resources in hand, we define the scope / goal of this project as follows: given a network topology (e.g. layers, connectivity), find the best combination of NN parameters per layer, such as filter size, number of filters per layer, stride size from a fixed pre-defined pool of choices. Additionally, we would like to be able to adjust the reward signal based on different requirements, for example, put constraints on size or computation cost per inference of the NN.

The main challenge of this project lies in three aspects: firstly, sampling child NN can take significantly amount of time, depending on the given network topology; secondly, the way to properly frame the task as a reinforcement learning problem is unclear; lastly, the reward signals from neural network typically contains many saddle points, as a result, the controller can be misled and stuck in a local optima.

We make a number of contributions in this project, including:

[1]Department of Electrical Engineering, Stanford University. Correspondence to: Yundong Zhang <yundong@stanford.edu>, Chang Yue <changyue@stanford.edu>, Mengwei Liu <mengweil@stanford.edu>.

- We construct a simple and visualizable reward function as a toy model for NAS, on which we can conveniently test our implementation and the design of the controller.
- We unveil and detail the implementation tricks of NAS, which is the key to train a stable controller.
- We present a successful application of NAS on a real-world dataset: Google Speech Commands Recognition (Warden, 2017), searching on a Convolution Neural Network.

The paper is organized as following: we first introduce related works that attempt to solve similar problems, then explain our technical approach including RL methods and validation. After that, we present experimental results and conclusion/discussion.

2. Related Work

Hyper-parameter searching / optimization is an active research area and widely used technique in machine learning. In (Price, 1983), the authors use constrained random search for hyper-parameter tuning, which is simple yet efficient. However, in the case of neural network, the search space is too huge and it is reported in (Bergstra et al., 2011) that random search usually performs worse than manually tuning. (Hsu et al., 2003) presents a practical guide on using Grid-search for choosing the parameters of Support Vector Machine (SVM). However, the search is exhaustive over the grid and the design of grid is heuristic. In other word, it is both computationally expensive and requires expert knowledge. (Bergstra & Bengio, 2012) design a new pipeline for random search on neural models, which is more reliable and efficient, however, it relies a good initial model. Another drawback of above mentioned methods is that they all search over a fixed space, meaning that they are not able to find novel models.

(Wierstra et al., 2005) and (Floreano et al., 2008) propose to use evolution algorithm for architecture search, which is able to find novel models. The limitation of evolution algorithm is that they are usually not capable of working on large scale. Neural architecture search is much more general, flexible and also able to find new architectures. Although generating a new architecture is not in our scope of this project, it is possible and straightforward to generalize the pipeline in this report to do that. For this part of details, one can also refer to the original paper of NAS (Zoph & Le, 2016).

Since the proposal of NAS, researchers have continuously improved the framework. For example, (Liu et al., 2017) design a new pipeline to search architecture progressively, similar to A* search (Hart et al., 1968), which prunes out unwanted models much faster. (Pham et al., 2018) propose a much more efficient way through parameter sharing of controller, claiming to reduce the search time of NAS by 1000x. Nevertheless, In this work, we focus on reimplementation of original NAS.

3. Approach

3.1. Overview

The NAS model (figure 1) typically consists of two major parts: a recurrent neural network (RNN) as a controller (also called parent AI), and a child neural network that we finally want to have. For each iteration, the controller generates a set of NN hyper-parameters to construct the child NN. The child NN is then trained on the provided dataset and will obtain a reward signal based on the test accuracy and other domain requirements (if any). After child NN feeding back its reward signal, the controller updates its parameter to maximize the probability of generating a new child NN that will obtain high reward for the next iteration. In other words, the controller learns to predict a good neural models over time for the interested task. In the following section, we will explain details in each component.

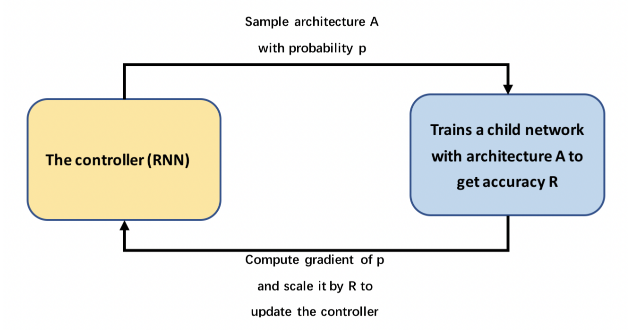

Figure 1. An overview of Neural Architecture Search. It contains a controller (in yellow) and a child NN (in blue).

3.2. Method

In this section we discuss how to frame the task as a reinforcement learning (RL) problem and tackle it. Here, the definition of reward is clear—simply the evaluation metrics of the child NN. However, the definition of actions and states can be vague. For action, one naive choice would be setting it to the prediction of the whole child NN, and treating the task as a multi-arm bandit problem. In this case, though we have a nice and simple definition of the task, it ignores the potential correlation between deep NN hyper-parameters and loses generalization property. For example, if many child NNs who shared a same setting of first layer get low reward, the controller should learn that the hyper-parameter combination of the first layer is bad. The above bandit setting does not exploit the correlation between arms, hence we believe it is not appropriate. Instead, We treat a single prediction (e.g. number of filters at layer N) as one action and use the RNN internal hidden state as RL states. In this way, the controller can generalize across states and actions. In fact, this is exactly the reason we model controller using RNN, in the hope that RNN can capture previous prediction in its hidden state.

Figure 2 shows the structure of the controller for a classical CNN in our NAS. At each time step, RNN takes in the current hidden information, embedding of previous prediction and uses a dense projection layer to generate confidence over the pool of next hyper-parameters. Then we sample the pool according to the confidence to generate final prediction, so as to trade off between exploration and exploitation.

Formally, let reward R be the evaluation results of predicted child NN, action a_t be the prediction of RNN at time step t, $\pi(s_t|a_{(t-1):1};\theta_c)$ = RNN$(s_t|a_{(t-1):1};\theta_c)$ be the current policy, state s_t be the internal hidden state of RNN at time step t, then we want to optimize the following quantity:

$$J(\theta_c) = E_{P(a_{1:T};\theta_c)}[R], \tag{1}$$

where $P(\cdot)$ is probability. Using policy gradient method, we can derive the gradient of (1) as in (Zoph & Le, 2016):

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

where T is the number of hyper-parameters of child NNs. With Monte-Carlo sampling, (2) can be approximated by:

$$\frac{1}{m}\sum_{k=1}^{m}\sum_{t=1}^{T} \nabla_{\theta_c} \log P(a_t|a_{(t-1):1};\theta_c)R_k] \tag{3}$$

where m is the batch size. Equation 4 is an unbiased estimate of 2, however, it has very high variance since we can only use limited batch size. We employ two simple tricks to stabilize the training, the first one is reward normalization, where we simply normalize the reward to have zero mean and unit variance R'; the second one is similar to (Zoph & Le, 2016), in which we use a baseline function b that is a moving average of the previous reward. The objective therefore becomes:

$$\frac{1}{m}\sum_{k=1}^{m}\sum_{t=1}^{T} \nabla_{\theta_c} \log P(a_t|a_{(t-1):1};\theta_c)(R'_k - b)] \tag{4}$$

3.3. Method Validation

To validate our method and formulation, we build a toy model as in figure 3. The toy model is a sum of multiple 2D-multivariate Gaussian centering at different mean, consisting of multiple local optima. It lies on a 1000x1000 x-y grid, with axis-z to be the reward. Hence, we can simply treat the model as a two-layer Multilayer Perceptron (MLP), where each layer can take 1 out of 1000 possible values as the unit number, resulting in a total of 1 million possible combinations. Using a self-built toy model is an extremely efficient way to validate the pipeline, since we don't need to train the child NN to get the reward signal, which usually takes long time.

3.4. Real World Task: Dataset and Preprocessing

We use the Google speech commands dataset (Warden, 2017) for the neural network architecture exploration real-world task. The dataset consists of 65,000 1-second long audio clips of 30 keywords, by thousands of different people, where each clip consisting of only one keyword. For simplicity and fast sampling, we pick two ('yes' and 'no') out of 30 keywords as our target, and classify the remaining 28 keywords as 'unknown' / 'background', together with another 'silence' class. Therefore, our child NN will perform a 4-class classification task. The dataset is split into training, validation and test set in the ratio of 80:10:10 while making sure that the audio clips from the same person stays in the same set.

Following the practice in (Zhang et al., 2017), we first frame each 1-second audio as multiple 10ms windows, with a stride of 20ms. This in total gives us $(1s - 10ms)/20ms =$ 49 windows. Then for each window, we extract 10 MFCC (Muda et al., 2010) features using filtering and FFT. At the end, each 1-second audio is represented as a 49x10 feature matrix, which is fed into neural models for classification. Data argumentation is also used for achieving better performance, including maximum of 100ms time-shift, background mixing and volume adjustment per audio.

4. Experiment and Result

4.1. Toy Model

We first show our results running on the proposed toy model. The pipeline flow is similar to figure 2 except that now each layer only has number of units (filters) to choose as the hyper-parameter. We use one-hot embeddings to embed the predictions into 1000-d vector. The controller is a one-layer unidirectional RNN, with 64-units NAS-cell (Zoph & Le, 2016) to be the RNN cell. At time step $t = 0$, input is a 1000-d zero vector and RNN hidden state is zero-state as well. Figure 4 shows the number of training batch vs. score and figure 5 shows the number of training batch vs. number of sampled models. Each training batch consists of 500 child NNs. We can see that after sampling only less than 60K out of 1M possible combinations, the controller

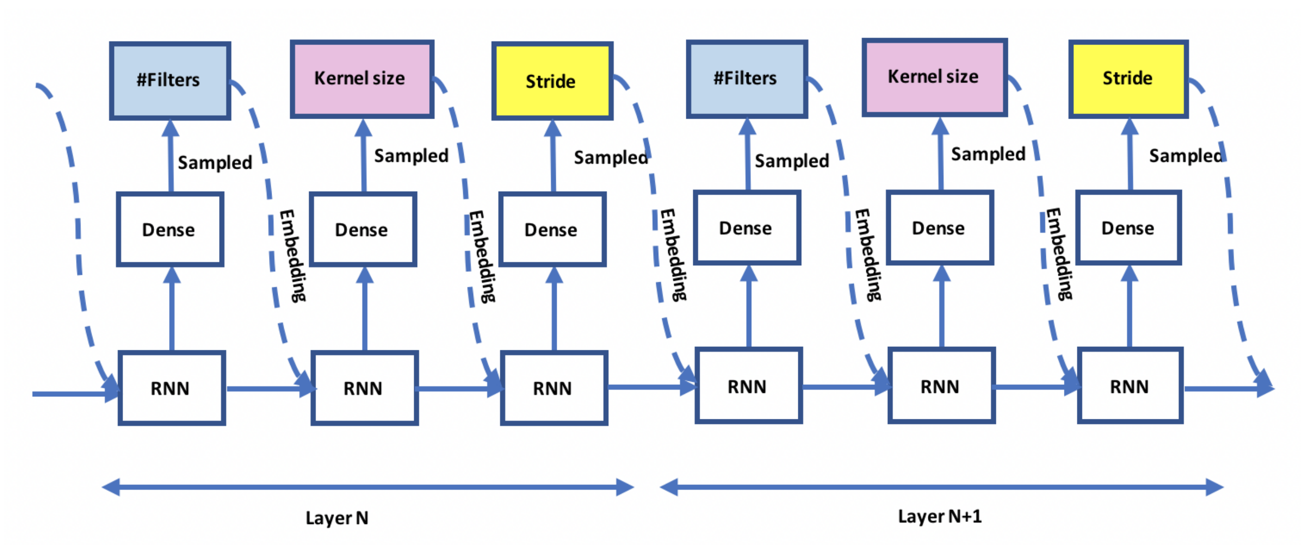

Figure 2. Structure of the RNN controller. It generates hyper-parameters sequentially. At layer N, it outputs that layer's hyper-parameters in the filter number, kernel size and stride order. After done with layer N, it starts generating hyper-parameters for layer $N + 1$.

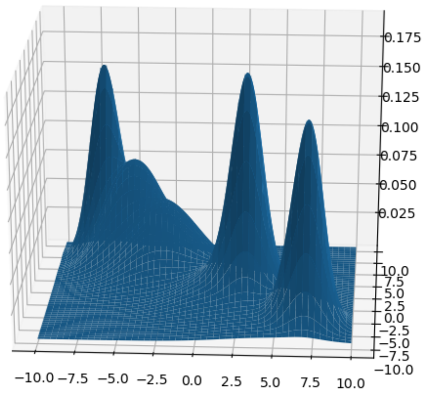

Figure 3. Toy Model for Validation. We assume that hyper-parameters lying in a 1000×1000 grid with a reward distribution as shown. It is a 2D mutli-normal distribution with several local optima, which mimics our child NN and its reward distribution.

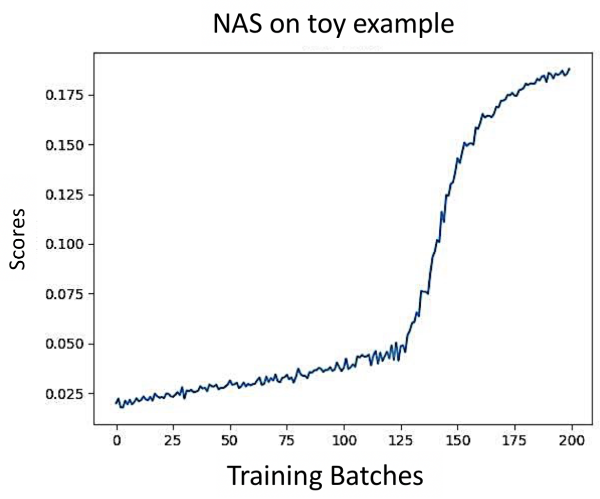

Figure 4. Validation Score of Toy Model VS Training Batch. It shows that the parent AI was learning the way to sample hyper-parameters, and it eventually converged to a point that is a very good optimal.

converges to a very good local optimum. This demonstrates the effectiveness of our method.

4.2. Real World Task: CNN (Discrete)

4.2.1. Set up

After proving the feasibility of the parent RNN - child NN pipeline, we move to solve the real-world task. By treating the input as a 49x10x1 'image', we can use the popular Convolution Neural Network (CNN) to solve the classification problem. Due to the limitations of time and computing resources, we restrict the search space to be three convolution layers, where per layer, we are free to choose the number of filters from a pool of [10, 50, 100, 200], the filter size from a pool of [3, 5], and the stride from a pool of [1, 2]. With this setting, we have $4 \times 2 \times 2 = 16$ parameter combinations for one layer, which gives a total number of $16^3 = 4096$

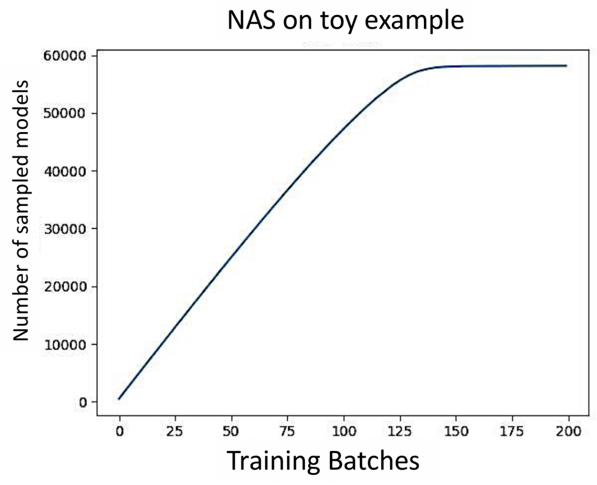

Figure 5. Number of Sampled Models VS Training Batch. While the parent AI was learning, it kept sample lots of models. After it received enough information about the reward distribution, it literally stopped sampling new models. After study 6,000 distinct models out of 1000,000 existing ones, the parent AI converges.

models for the network. If we use a traditional method of parameter searching, such as grid search, we are going to train 4096 CNNs to get the globally best. By statistics, we need to sample about 400 models to get a top 10 CNN, 800 to get the top 5, and 2000 to get one of the best two.

To validate and compare our result, we need information about every CNN model in a global sense. Therefore, we created a buffer to store each CNN's accuracy. We train and test every model and gather the accuracy of all 4096 models, so that we can know the global best accuracy and etc. However, the RNN is supposed to only look at the accuracies of models it samples. We train on 5 GPUs for 4 days.

For RNN controller, the ultimate goal is to find hyper-parameters of the CNN that is best under any pre-stated condition. We test our model in three scenarios: only looking for highest accuracy; searching for highest accuracy with regularization that controls model complexity; searching for model having highest accuracy under hard constraints. Here our regularizations/constraints are total number of parameters/weights and number of computation cost per inference, since we want our CNN applicable on micro-controllers.

4.2.2. Mechanism

As shown in Figure 2, the RNN controller generates hyper-parameters sequentially. We treat this process as a fixed horizon reinforcement learning problem. Since we have 9 hyper-parameters to sample, $H = 9$. Sampling a hyper-parameter is equivalent to taking an action. The RNN controller will firstly receive an initialization as a starting point, then, it will go through 9 loops where at each loop, it outputs a length-64 vector. Depending on the current action the RNN is sampling (whether it is filter number or kernel size or stride), this 64-length vector is passed in to a dense layer. The dense layer output another vector that has the dimension as the same of the action space (4 for filter number, and 2 for kernel size and stride), and this vector contains information of sampling probability. We then sample a hyper-parameter based on this dense layer output. After gathering samples of all 9 hyper-parameters, we get the CNN model structure. We then go search the accuracy of this sampled CNN in our buffer, compute its reward based on constraints if any. We feed back this reward to RNN controller. For each batch, the RNN samples a number of CNN models same as the batch size. We record the average accuracy in that batch, as well as the total number of distinct models the RNN has sampled so far.

4.2.3. Result

Figure 6 shows the RNN sampled models' accuracy versus number of iterations, and Figure 7 shows the corresponding number of distinct CNN models that the RNN controller has seen. The batch size is 64. Blue line shows the accuracy of scenario 1; yellow line shows scenario 2; green line shows scenario 3. Without any constraint, RNN converged to a model that is globally second accurate (with an accuracy of 0.9844), just after seeing 170 out of 4096 models. This is on average 12 times less than that of randomly choosing hyper-parameters. However, the model it converged seems complex. Therefore, we added two regularization terms: one to penalize number of model parameters, the other to penalize computation cost. This time, the RNN learns slower. After studied 220 models, it yield a smaller model. This model is the forth best among all 4096 models, and it has an accuracy of 0.9828. We further moved to the hard constraint side, because to make sure the CNN really works on small micro-controller, we need to harshly cut the complex ones. We set the threshold to be 8000 parameters and 100 million operations per inference. During training, we set the reward of models that passes this threshold to 0. The RNN learns slightly slower, but it finally converged to a place just as that of scenario 2. Globally, this is the best model under our constraints. It only studied 291 models this time. For this task, we proved both the efficiency and robustness of our RNN controller, and therefore, our auto machine learning pipeline.

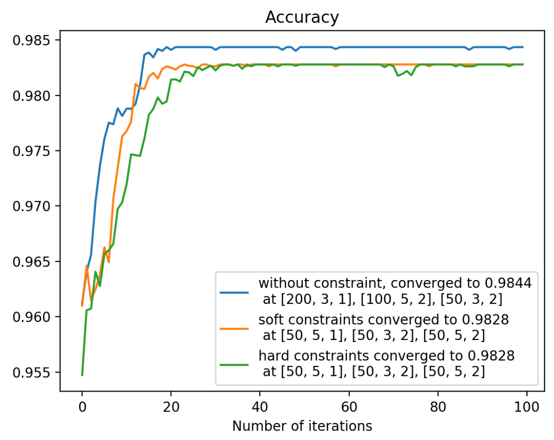

Figure 6. Sample accuracy VS number of iterations. Blue line shows the batch accuracy without regularization; yellow line shows accuracy with soft regularizations; green line shows accuracy with hard constraints. Without constraint, the RNN learns fast and converged the globally second best model. With constraints, it learns slower, converged to a smaller model. This is also the model with the highest accuracy under our limitations.

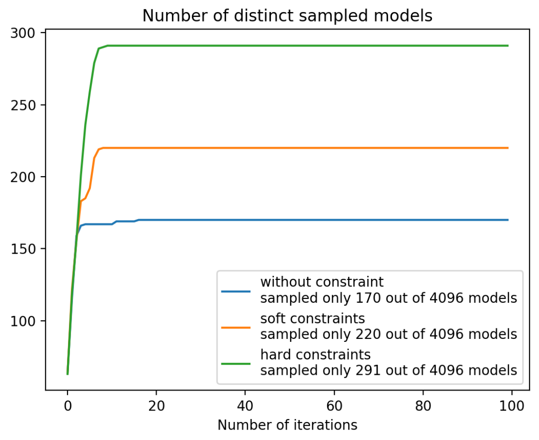

Figure 7. Number of distinct CNN models the RNN controller has seen VS number of iterations. The number of models needed to learn increases as the level of constraints increase, but it is still very efficient compared to random sample.

4.3. Real World Task: CNN (General)

4.3.1. Set up

In section 4.2, we conduct our search within a fixed and discrete space. In this section, we will expand this work to more general search space, where the number of filters per layer can be any integer within a given range. To do this, we slightly change the scheme of predicting number of filters D_N for layer N. That is, instead of sampling from a predefined fixed-length vector based on the logits, now we approximate the distribution of D_N using Normal distribution. The dense projection output in section 4.2 now is simply a scalar, representing the means of the Gaussian. The log standard deviation of Gaussian is modeled by a trainable variable. All the other mechanism remain unchanged.

Since we don't have enough resources for sampling models other than the previous 4096 models in the buffer, we use these models to interpolate the accuracy for other models. We also restrict the search space of $0 \leq D \leq 100$ for simplicity. The interpolation steps are as follows: starting from the last layer, we use $(l1, k1, s1, l2, k2, s2, 10, k3, s3)$, $(l1, k1, s1, l2, k2, s2, 50, k3, s3)$, $(l1, k1, s1, l2, k2, s2, 100, k3, s3)$ to fit a second-order polynomial and interpolate all the values of the above sequence for $l3 \in [0, 100]$. We choose 2nd order because given 3-points we can uniquely determine a quadratic function. Then similarly we interpolate values for $l2$ and $l1$. Note that this is a very naive way and as a result the final reward function can be very inaccurate. Some model will even have an accuracy larger than 100%. Therefore for below discussion we use the reward, which is the normalized accuracy for performance evaluation.

There are some key elements in this experiment for successful training, they are:

- For the dense projection layer, we change it to 2-layer MLP to increase the capability and initialize the last layer to have bias of 50 (mid of $[0, 100]$)
- Threshold the sampled prediction within the given range, i.e. $[0, 100]$
- Use moving average baseline to reduce variance

4.3.2. Result

We show our experiment result for general CNN here. As in figure 8 and figure 9, the controller learns to predict an architecture of reward 4.06 with only sampling 1000 out of $(100 * 2 * 2)^3 = 64$ millions possible combinations. Note that 4.06 is greater than the global maximum in the discrete pool (i.e. filter: (10, 50, 100), kernel: (3, 5) and stride: (1,2)), and is the top 0.07% values of all the interpolated reward. This matches our results in section 4.2.3, where the model ([50, 5, 1], [50, 3, 2], [50 5, 2]) is a local optimum.

5. Conclusion and Discussion

We proved that our re-implementation of Neural Architecture Search using REINFORCE policy gradient is accurate, viable and most importantly, efficient. We list the important

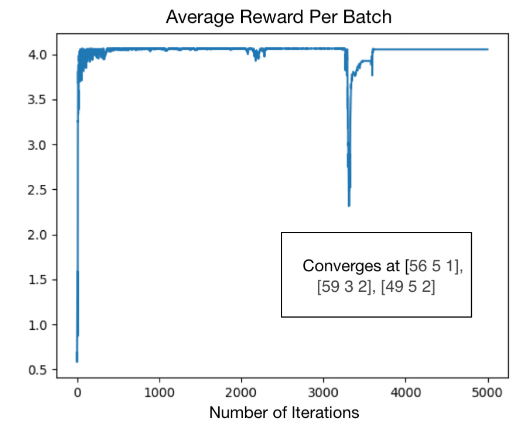

Figure 8. Reward VS. Training Iterations in general CNN

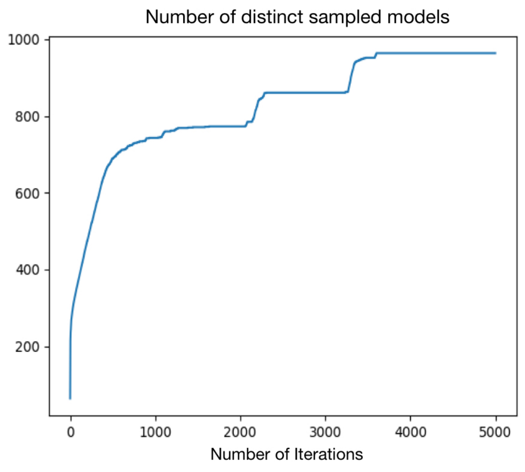

Figure 9. Number of distinct CNN models the RNN controller has seen VS number of iterations in general CNN

tricks concretely for successful deployment. Our method not only works on a given fixed pool, but also works for any given integer within pre-determined range. Hence it is more general and flexible. Nevertheless, the design of controller architecture and training strategy is not so straightforward. So the overall process is still not fully-automatic. However, one can do grid-search on controller and use controller to do NAS. We believe such strategy still significantly reduce the tuning time, since hyper-parameter of controller is usually much smaller.

References

Ashrafi, Parivash, Sun, Yi, Davey, Neil, Adams, Rod, Brown, Marc B, Prapopoulou, Maria, and Moss, Gary. The importance of hyperparameters selection within small datasets. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pp. 1–8. IEEE, 2015.

Bergstra, James and Bengio, Yoshua. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13(Feb):281–305, 2012.

Bergstra, James S, Bardenet, Rémi, Bengio, Yoshua, and Kégl, Balázs. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pp. 2546–2554, 2011.

Cho, Kyunghyun, Van Merriënboer, Bart, Gulcehre, Caglar, Bahdanau, Dzmitry, Bougares, Fethi, Schwenk, Holger, and Bengio, Yoshua. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Floreano, Dario, Dürr, Peter, and Mattiussi, Claudio. Neuroevolution: from architectures to learning. *Evolutionary Intelligence*, 1(1):47–62, 2008.

Hart, Peter E, Nilsson, Nils J, and Raphael, Bertram. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

Hinton, Geoffrey, Deng, Li, Yu, Dong, Dahl, George E, Mohamed, Abdel-rahman, Jaitly, Navdeep, Senior, Andrew, Vanhoucke, Vincent, Nguyen, Patrick, Sainath, Tara N, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

Hsu, Chih-Wei, Chang, Chih-Chung, Lin, Chih-Jen, et al. A practical guide to support vector classification. 2003.

Krizhevsky, Alex, Sutskever, Ilya, and Hinton, Geoffrey E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

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 preprint arXiv:1712.00559*, 2017.

Muda, Lindasalwa, Begam, Mumtaj, and Elamvazuthi, Irraivan. Voice recognition algorithms using mel frequency cepstral coefficient (mfcc) and dynamic time warping (dtw) techniques. *arXiv preprint arXiv:1003.4083*, 2010.

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

Price, WL. Global optimization by controlled random search. *Journal of Optimization Theory and Applications*, 40(3):333–348, 1983.

van Rijn, Jan N and Hutter, Frank. Hyperparameter importance across datasets. *arXiv preprint arXiv:1710.04725*, 2017.

Warden, Pete. Speech commands: A public dataset for single-word speech recognition. `http://download.tensorflow.org/data/speech_commands_v0.01.tar.gz`, 2017.

Wierstra, Daan, Gomez, Faustino J, and Schmidhuber, Jürgen. Modeling systems with internal state using evolino. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pp. 1795–1802. ACM, 2005.

Zhang, Yundong, Suda, Naveen, Lai, Liangzhen, and Chandra, Vikas. Hello edge: Keyword spotting on microcontrollers. *arXiv preprint arXiv:1711.07128*, 2017.

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