A Meta-Reinforcement Learning Approach to Optimize Parameters and Hyper-parameters Simultaneously

Abbas Raza Ali[1][0000−0003−0505−2638], Marcin Budka[1][0000−0003−0158−6309], and Bogdan Gabrys[2][0000−0002−0790−2846]

[1] Bournemouth University, Poole BH12 5BB, UK
{aali,mbudka}@bournemouth.ac.uk
[2] University Technology Sydney, Ultimo NSW 2007, Australia
bogdan.gabrys@uts.edu.au

Abstract. In the last few years, we have witnessed a resurgence of interest in neural networks. The state-of-the-art deep neural network architectures are however challenging to design from scratch and requiring computationally costly empirical evaluations. Hence, there has been a lot of research effort dedicated to effective utilisation and adaptation of previously proposed architectures either by using transfer learning or by modifying the original architecture. The ultimate goal of designing a network architecture is to achieve the best possible accuracy for a given task or group of related tasks. Although there have been some efforts to automate network architecture design process, most of the existing solutions are still very computationally intensive. This work presents a framework to automatically find a good set of hyper-parameters resulting in reasonably good accuracy, which at the same time is less computationally expensive than the existing approaches. The idea presented here is to frame the hyper-parameter selection and tuning within the reinforcement learning regime. Thus, the parameters of a meta-learner, RNN, and hyper-parameters of the target network are tuned simultaneously. Our meta-learner is being updated using policy network and simultaneously generates a tuple of hyper-parameters which are utilized by another network. The network is trained on a given task for a number of steps and produces validation accuracy whose delta is used as reward. The reward along with the state of the network, comprising statistics of network's final layer outcome and training loss, are fed back to the meta-learner which in turn generates a tuned tuple of hyper-parameters for the next time-step. Therefore, the effectiveness of a recommended tuple can be tested very quickly rather than waiting for the network to converge. This approach produces accuracy close to the state-of-the-art approach and is found to be comparatively less computationally intensive.

Keywords: Convolutional Neural Networks · Meta-Learning · Reinforcement Learning · Policy Gradients · Hyper-parameter optimization

1 Introduction

Deep neural networks (DNN) have attained tremendous success by consistently outperforming the shallow learning techniques. However, solving complex tasks need deeper and wider networks which are considered hard to design. Transfer learning, often, works well on simple and more general tasks whereas complex tasks require effort to design a customized network. The network designing process requires specialized skills and numerous trials which is a time consuming and computationally expensive task. The state-of-the-art networks require well-tuned hyper-parameters which often demand numerous computationally intensive trials.

In recent years, Meta-Reinforcement Learning (Meta-RL) has become a defacto standard to automatically search for optimal hyper-parameters. Therefore, the proposed framework uses Meta-RL to efficiently explore the optimal hyper-parameters of a deep network from the given search space. The exploration happens simultaneously for both the policy network and the DNN. Given a tuple of hyper-parameters that is generated by a policy network, a network is built and trained for a number of steps. The network computes accuracy on hold-out validation-set whose delta is used as a reward. Furthermore, this reward along with the state of the network comprising statistics of probability distribution over the number of classes and training loss, are back-propagated to the policy network which generates a tuned tuple for the next time-step. The network is initialized once where different tuples of hyper-parameters are tested on the go without resetting the network. Therefore, a tuple of hyper-parameters is not required to train till convergence of the network which saves a significant amount of computation.

There are a number of recent studies around hyper-parameter optimization using Reinforcement learning. The earliest effort of Meta-RL was made by [3] where a recurrent neural network (RNN) based agent is used to learn the behavior of the environment. The goal of the agent is to learn a policy for learning new policies. The Meta-RL is defined in this work in a way that the agent gets trained once on a problem and transfer learned on similar kind of tasks. Moreover, the idea is a learning policy to learn another policy in a family of similar Markov Decision Processes (MDPs). A Meta-agent adjusts its policy after training for a few episodes and validates on an unseen environment. This approach worked well on both small- and large-scale problems. Another simple, yet powerful Meta-RL approach is Model-Agnostic Meta-learning (MAML) [4]. MAML does not initialize model parameters randomly but rather it provides a good initialization to achieve optimal and efficient learning on a new task. The fine-tuning requires a small number of gradient steps. The key aspect of the MAML is that the model can be trained using a gradient descent including convolutional neural networks (CNNs) with a variety of potential loss functions. Additionally, it is equally effective for regression, classification, and reinforcement learning, where it outperformed a number of previous approaches.

[18] proposed a long short-term memory (LSTM) [8] based approach to train a meta-classifier. The few-shot learning method finds the optimal set of parame-

ters. However, [5] claims that the MAML initialization of the model parameters is more resilient to over-fitting, particularly, for smaller datasets. Also, it is more effective when the model is dealing with new unseen tasks. Similarly, [1] proposed an effective and efficient domain adaption approach by fine-tuning the final layers of a CNN for both small- and large-scale problems.

Neural Architectural Search (NAS) is another effort towards Meta-RL based network search [24]. NAS uses an RNN based controller that samples a candidate architecture known as child network. The child network is trained till convergence to obtain accuracy on a hold-out validation-set. The accuracy is used as an immediate reward which further updates the controller. The controller generates better architectures over time where the weights are updated by policy gradient. The approach seems quite simple and powerful but it is tested on very small size tasks. Another observation is that the search space of the child network was limited. The reason behind limiting the experiment to small tasks is the inefficiency of the approach. Progressive Neural Architecture Search (PNAS) proposes a different approach to architecture search known as sequential model-based optimization (SMBO) strategy [15]. In SMBO, instead of randomly recommending and testing out the blocks, they are tested and structures searched in order of increasing complexity. Instead of traversing the entire search space, this approach starts off simple and only gets complex when required. PNAS claims to be significantly less computationally expensive than NAS. Another effort to make architecture search more efficient is known as, Efficient Neural Architecture Search (ENAS), proposed by [17]. ENAS allows sharing of weights across all the models instead of training every model from scratch. The idea is to reuse the weights of a block which are already trained. Thus, the system uses transfer learning to train a new model which makes convergence very fast. It is a very effective method and comparatively less computationally expensive than PNAS. The only observation about this approach is that it keeps a large number of architectures in the memory.

[23] proposed a different approach of learning to do exploration in off-policy RL which is Deep Deterministic Policy Gradients (DDPG). The authors compared two different policy gradient RL approaches: a) On-policy Gradient Algorithms (OPGA) which includes algorithms like Proximal Policy Optimization (PPO), and b) Trust Region Policy Optimization (TRPO) where a stochastic policy is used for exploration of RL environment. A separate policy has been used instead of a simple heuristic for the exploration. This policy is trained using OPGA methods where the reward for training is a relative improvement in the performance of the exploitation policy network. Experimental results show faster convergence of DDPG with higher rewards. [25] further extended NAS where they also replaced REINFORCE with PPO.

The proposed approach is an efficient form of NAS and ENAS to find optimal neural architecture. The shortcomings of NAS is its limitation to small tasks because it is computationally very expensive. On the other hand, ENAS keeps numerous architectures in the memory so that the new architectures can share the weights of the pre-trained blocks. This work further simplifies architecture

search problem which is equally effective for large datasets. The approach tunes the hyper-parameters of the network during training rather than waiting until convergence which saves significant computation time. The effectiveness of a tuple of hyper-parameters is tested by training for a few steps. Further, the feedback of the tuple is used to tune the policy gradient at the same time-step.

This method significantly reduces the computational complexity of the optimal hyper-parameter search problem. Along with minimal computation, the approach requires a substantially smaller amount of memory by optimizing a single instance of the network rather than creating and keeping numerous architectures in the memory. The simplicity of the approach does not affect the accuracy of the network and makes it equally effective for more complex and bigger tasks. This is the key contribution of this study.

The rest of the paper is organized as follows. Section 2 is devoted to discussing the methodology of this study. The formulation of REINFORCE, base-learner and stochastic depth algorithms are outlined in Section 3. Section 4 outlines the data-sources and different configurations that have been used to conduct various experiments. Section 5 reports the experimental results and their analysis. Finally, the paper is concluded in Section 6.

2 Methodology

The primary goal of this study is to efficiently explore the optimal set of hyper-parameters for a given task. This is achieved by optimizing the meta-learner parameters and network hyper-parameters at the same time. Typically, the policy network needs to train for several episodes so that it can start producing an effective outcome. In case of hyper-parameter tuning using Meta-RL, the child network needs to be sequentially trained on a task at hand using all the tuples, recommended by the meta-learner, until convergence in order to conclude their effectiveness. It becomes a time and computationally intensive task. Hence, this challenge has been tackled and addressed in this study.

In order to evaluate the proposed approach, a framework is designed using a typical RL setting which consists of two key components: an agent and an environment [20]. The environment can be in different states (S) which are observed by the agent at different time-steps (t). Given its knowledge of the state and a set of available actions the agent chooses an action (A). These actions affect the state of the environment and in return, generate a reward (R). To find the optimal set of hyper-parameters the agent needs to find the actions that lead to maximizing expected reward, see Equation 1. The γ is a discount factor, which allows the agent to maximize its expected reward on either short- or long-term transitions based on its value. However, the reward is non-differentiable and hence needs a policy gradient method to iteratively update θ as formulated in Equation 2. The stochastic policy $\pi(a|s)$ describes a probability distribution over the set of actions.

$$R_t \leftarrow \sum_{i=0}^{\infty} \gamma^i R_{t+i}, \quad \gamma \in [0,1] \tag{1}$$

$$\theta \leftarrow \theta + \alpha \nabla_\theta log \pi_\theta(s_t, a_t) r_t \tag{2}$$

The agent generates a tuple of hyper-parameters using an RNN which is known as meta-learner. This tuple specifies a neural network architecture known as base-learner in the framework. The base-learner is trained on a task and evaluated on the held-out validation-set. The base-learner provides feedback to the meta-learner to get a well-tuned tuple in the next time-step. Figure 1 shows the setting of the proposed Meta-RL framework.

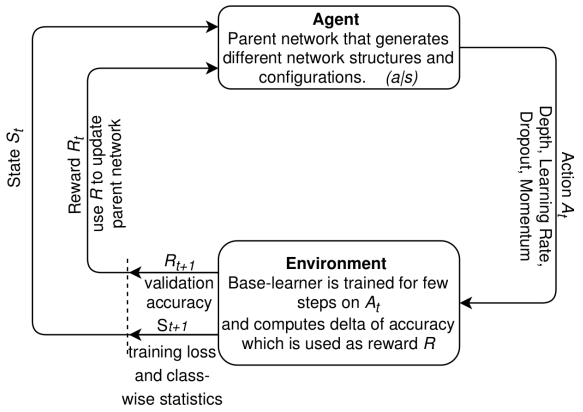

Fig. 1: A typical setting of Meta-RL framework where agent contains a policy gradient and network sits in the environment

2.1 Meta-learner

The Meta-learner consists of a stochastic policy gradient which makes weight adjustments in a direction that lies along the gradient of expected reinforcement. It is a statistical gradient-based approach known as REINFORCE as described by [21]. It makes weights adjustment without explicitly computing gradient estimates with back-propagation. The Meta-learner initializes a base-learner once with the initial values of hyper-parameters from search space except for depth. However, the depth is initialized with the maximum value. For instance, if the maximum depth is 34 in the search space, the network is initialized once with the maximum depth. The meta-learner is a two-layer RNN LSTM with 35 neurons per layer. The network is trained with Adam optimizer [11]. An initial learning rate of 0.0006 has been used. The weights are initialized with Xavier-initialization [6]. A discount factor of 0.97 is used to prevent the total reward from reaching infinity. The meta-learner is updated via a policy gradient method which is computed using an immediate reward.

2.2 Base-learner

The base-learner used in this work is a modified form of Residual Network (ResNet) [7]. It is constructed by stacking a set of residual blocks on top of the input layer and followed by a fully-connected (FC) layer. A block consists of a sequence of two convolutional layers with filter sizes 1x1 and 3x3, respectively,

Algorithm 1 Computing immediate reward of an episode

1: $beta = 0.8$
2: Time-step $= t$
3: $episode = e$
4:
5: $reward_t = (accuracy - moving\_accuracy_{t-1})$
6: $reward_t = clip(reward, -0.1, 0.1)$
7:
8: $moving\_accuracy_t = (1 - beta) * accuracy_e$
9: $moving\_accuracy_t \mathrel{+}= beta * moving\_accuracy_{t-1}$

where a stride of 2 is used by the first convolutional layer to reduce feature map size. Also, there is a bottleneck setting of the block which consists of three convolutional layers with filter sizes of 1x1, 3x3 and 1x1, respectively. The bottleneck block is used for the networks with a depth of 50 or more. The benefit of using ResNet architecture is two-fold: a) residual blocks have repeated units of convolutions with fixed hyper-parameters, namely, kernels and strides, and b) it has a skip-connection feature that provides flexibility to change the depth of the network during the training. The base-leaner has been initiated once and its hyper-parameters are modified during the training cycles.

Table 1: Hyper-parameter search space and parameters covering behaviour of the network that is used as states $_{t+1}$

Parameters	Values (range)
A. Hyper-parameter search space	
Number of layers (D)	2-50
Dropout Rate (DR)	0.5-1.0
Learning Rate (LR)	0.0001-0.9
Momentum (M)	0.6-0.99
B. Representation of the environment (states)	
Network training loss	0-1.0
Mean entropy of class probabilities	0-1.0
Standard deviation entropy of class probabilities	0-1.0

The meta-learner (RNN) suggests a tuple of hyper-parameters from the search space which are listed in Table 1 (A). The table shows the search space range of all the hyper-parameters. Based on the suggested hyper-parameters, the existing CNN architecture is trained for 50 steps with a batch size of 32. Furthermore, delta of validation accuracy has been computed which becomes the immediate reward. The reward that is used to update the meta-learner is the delta of validation accuracy of the recent two episodes. The procedure to compute the immediate reward is formulated in Algorithm 1. Apart from the reward few other parameters of the environment are computed at time-step t comprising of network training loss and entropy of probability distribution over number of classes. The entropy is averaged over an episode, see Equation 3, where x is the output of the softmax layer and N is the size of the episode. Further, the

mean and standard deviation of the entropy has been computed over the number of images, N, processed in an episode, see Table 1 (B). These parameters are utilized by meta-learner as the state information to generate a tuned tuple for time-step $t+1$. The network is trained with Momentum optimizer with Nesterov momentum [19].

$$\text{entropy} = -\sum_{j=1}^{N}(\mathbf{f}_j(x_i) * \log_2(\mathbf{f}_j(x_i))) \tag{3}$$

$$x_{l+1} = ReLU(x_l + \mathbf{f}(x_l, W_l)) \tag{4}$$

Residual Block with Stochastic Depth A residual block is composed of convolution layers, batch normalization (BatchNorm) [10] and rectified linear units (ReLU) [16] which is represented as function f in Equation 4. x_l represents skip-connection path and $f(x_l, W_l)$ is a residual block. A configuration of the base-learner with maximum depth 4 is shown in Figure 2 (a). The meta-learner has recommended a depth size 3 so the last residual block has been disabled for the current episode. Hence, the gradient update of the last block is stopped for the current episode.

The depth of the network is controlled by stochastic depth approach presented by [9]. It leverages the skip-connection path of the residual block x_l to control network depth even during training of the network. The idea of original stochastic depth work, [9], is to randomly skip the residual blocks by letting through only the identity of the raw feature in order to skip a path. In this work rather than randomly skipping the blocks, meta-learner suggests which blocks to skip. Therefore, when a block is skipped, the identity path has been chosen which stops updating the block's gradients.

3 Formulation

The approach to optimize parameters and hyper-parameters simultaneously is outlined in Algorithm 2. It has two components: a) meta-learner and b) base-learner. A meta-learner is an RNN which suggests a tuple of hyper-parameters in the form of actions. These actions are applied to the environment which is a base-learner. The base-learner is a CNN which trains the task at hand on the actions of current time-step for a few steps. Furthermore, the network computes accuracy on a hold-out validation-set which is used as an immediate reward at the time-step t. This reward and the state of the network is observed and used to update the weights of the meta-learner that generates new actions for time-step $t+1$ which are dependent on how well the base-learner performs.

Algorithm 3 shows how stochastic depth approach is modified for this work. The base-learner only updates the gradients of the residual blocks which are less than the suggested depth (D). For the rest of the layers, a skip-connection path has opted. The base-learner is initialized with a maximum value of the depth once and modifies, often, on every episode.

Algorithm 2 Meta-Reinforcement learning algorithm to optimize parameters and hyper-parameters simultaneously

▷ META-LEARNER
Network depth = D
Dropout rate = DR
Base-learner's Learning rate = α_b
Momentum = p
Actions (a) = $< D, DR, \alpha_b, p >$
Time-step = t
Meta-learner's Learning rate = α_m
Reward at time $t = r_t$
Differential policy at time t which maps actions to probabilities = $\pi_\theta(s_t, a_t)$
Initialize the policy parameter: θ = Xavier-initialization
Initialize base-learner CNN: $model \leftarrow ResNet(a)$

for $episode \leftarrow 1$ to $\pi_\theta : s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T$ **do**
    ▷ policy network
    for $t \leftarrow 1$ to $T - 1$ **do**
        $\theta \leftarrow \theta + \alpha_m \nabla_\theta log \pi_\theta(s_t, a_t) r_t$ ▷ gradient update

        ▷ BASE-LEARNER
        ▷ Tune the hyper-parameters of network with θ
        for $s \leftarrow 1$ to $Steps \leftarrow 50$ **do**
            $features \leftarrow next\_batch(train, labels)$
            if $training = True$ **then**
                $fit\_model \leftarrow model.fit(a, features)$
            end if
        end for
        if $testing = True$ **then**
            $test\_accuracy \leftarrow fitted\_model(testset)$
        end if

        $r_t = test\_accuracy_t - moving\_accuracy_{t-1}$
        $s_t^1 = train\_loss$ ▷ states of t
        $s_t^2 = final\_layer\_statistics$
    end for
end for

Algorithm 3 Stochastic Depth routine

Depth suggested by meta-learner = D
Maximum depth of a network = $maxD$

for $block\_no \leftarrow 1$ to $maxD$ **do**
    if $block\_no >= D$ **then** $x \leftarrow ReLU(x + \mathbf{f}(x, W))$ ▷ residual block
    end if
    if $block\_no < D$ **then** $x \leftarrow Identity(x)$ ▷ shortcut
    end if
end for

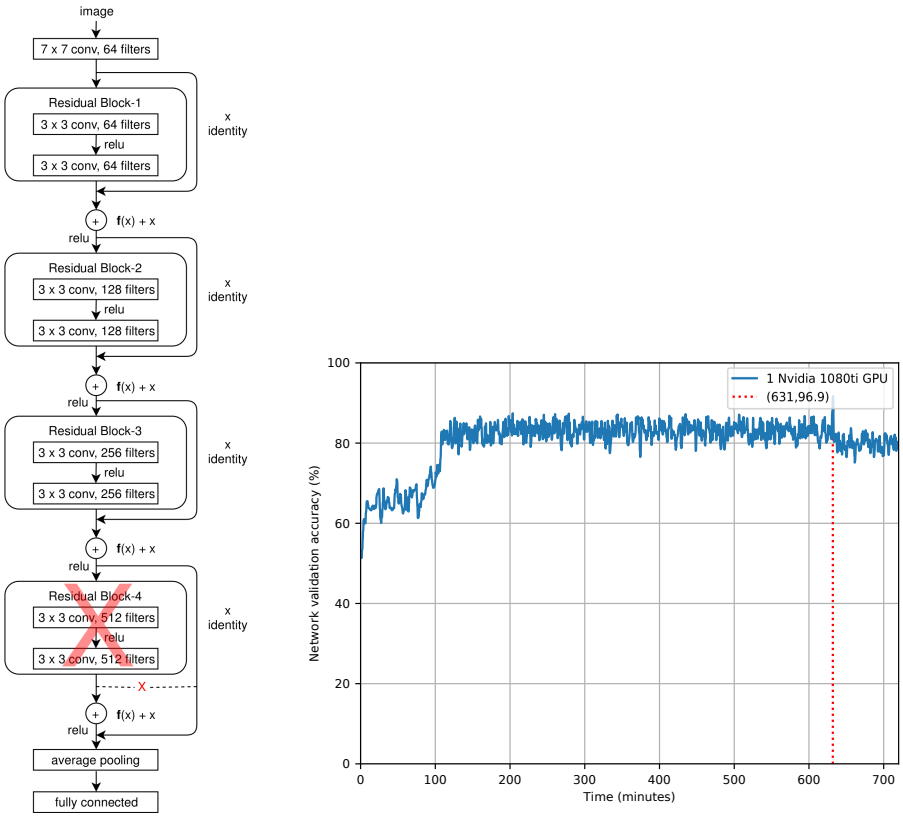

Fig. 2: (a) A schematic view of base-learner with maximum depth 4 and current depth 3. (b) Cifar-10 time taken versus network validation accuracy plot

4 Experimentation Environment

In order to evaluate the proposed approach, a number of experiments have been performed. These experiments use different image classification tasks listed in Table 2. A tuple of hyper-parameters is tested for only a few steps rather than till convergence. Hence, the number of steps the base-learner trains on a tuple of hyper-parameters is a critical parameter. Thus, different values of step-size and batch size have been tested to obtain the optimal values which can evaluate a recommended tuple in the shortest time. The experiments suggest a step-size 50 with a batch size 32 which is sufficient to test a tuple of hyper-parameters efficiently. Likewise, capturing the appropriate parameters which can better represent the state of the network after a training episode is key. The accuracy or loss can be sufficient if for each of the generated tuples the network is trained until convergence. Hence, the behaviour of the environment, at every episode, has been captured to evaluate the effectiveness of the recommended tuple. The effectiveness of a tuple is measured using the validation accuracy.

4.1 Datasets

In this work, five publicly available datasets have been used with different characteristics and complexity levels. The proposed approach is equally effective for both small and large datasets unlike most of the neural architecture search approaches which are only tested on small datasets. The datasets size, number of classes and image resolution is listed in Table 2. The datasets are divided into training- and validation-set with 80-20 split.

Table 2: Image datasets used in this work

Dataset	Training-set	Testing-set	Classes	Dimensions
Mnist [14]	50,000	10,000	10	28x28x1
Fashion-mnist [22]	60,000	10,000	10	28x28x1
Cifar-10 [12]	50,000	10,000	10	32x32x3
Cifar-100 [12]	50,00	10,000	100	32x32x3
Tiny-imagenet [13]	100,000	20,000	200	64x64x3

5 Results and Analysis

A comprehensive set of experiments is conducted to evaluate the effectiveness of the proposed approach. The experiments were performed on 5 Nvidia 1080Ti GPUs, one dataset per GPU. A comparison of the proposed approach with other architecture search approaches is shown in Table 3. This comparison is only available for Cifar-10 dataset as most of the previous studies used it in their experiments. A plot of validation accuracy against the time taken can be seen in Figure 2 (b). The vertical red dotted line is pointing to the top accuracy whose hyper-parameters settings are mentioned in Table 4.

Table 3: Comparison with different architecture search approaches on Cifar-10 dataset

Method	GPUs	Exploration time (days)	Parameters (millions)	Error rate (%)
DenseNet [2]	-	-	26.20	3.46
NASNet-A [25]	450	3-4	3.30	3.41
PNAS [15]	100	1.5	3.20	3.63
ENAS [17]	1	0.60	4.60	2.89
This work (Cifar-10)	1	0.40	4.58	3.11

There are 5 datasets used for experiments with different complexity-levels. The exploration of hyper-parameters for the datasets posses different behaviors in terms of the number of episodes and time. The Mnist, Fasion-mnist and Cifar-10 datasets were comparatively easier to learn. On the other hand, the exploration of Cifar-100 and tiny-imagenet was hard. The complex datasets took many more episodes to explore the optimal parameters from the search space. Moreover, the maximum depth of the architectures was bigger for complex datasets. So a large increase of depth size from one episode to other, particularly in the

initial phase, makes the training quite unstable. Figure 2 (b) shows a consistent accuracy after 100 minutes of training till 630 followed by a spike on a tuple. This tuple produced the maximum accuracy which is reported in Table 4. At the beginning of the training, a much bigger improvement in accuracy has been observed with a tuple which is different than the highest performing hyper-parameter tuple. A network is trained separately from scratch using the highest performing tuple until convergence which produces an error rate of 3.19 which is close to the one mentioned in Table 3. This approach is repeated for the rest of the datasets which produces the accuracy close to the one reported in Table 4 with a marginal difference range of ± 0.15. It depicts the effectiveness of the reported highest performing hyper-parameters tuple in the shortest time.

Figure 3 shows the policy loss, reward and network validation accuracy of the 5 datasets. The plots show a vertical line along y-axis representing maximum accuracy. The best hyper-parameters found against each dataset are reported in Table 4 along with the exploration and network accuracy information. The mnist and fashion-mnist tasks took very few episodes to find the top performing hyper-parameters. On the contrary, the complex tasks, cifar-100 and tiny-imagenet, took many more episodes to try different permutations of the hyper-parameters.

Table 4: Accuracy of various datasets including optimal parameters and episodes required to achieve the optimal value

Dataset	Network Hyper-parameters $[D, DR, \alpha, p]$	Episodes	Duration (hours)	Network Accuracy (%)
Mnist	[4, 0.06, 0.02, 0.95]	720	0.72	98.29
Fashion-mnist	[4, 0.06, 0.02, 0.95]	466	0.36	95.37
Cifar-10	[4, 0.3, 0.006, 0.95]	7,203	10.53	96.89
Cifar-100	[11, 0.2, 0.0007, 0.93]	9,810	19.39	76.94
Tiny-imagenet	[16, 0.25, 0.0004, 0.89]	13,770	36.83	64.39

The network hyper-parameters are initialized once and tuned after every 50 steps. The policy gradient took more episodes to learn the hyper-parameters for more complex tasks. To evaluate a recommended tuple, 50 steps are very limited, hence the behavior of the network was captured and provided to the meta-learner to more fully observe the impact of the tuple.

6 Conclusions

This study has presented an efficient approach to hyper-parameters search of deep models. A Policy-based Reinforcement Learning method is used to generate a tuple of hyper-parameters. The tuple is used by the target network, base-learner, which is initialized once with random hyper-parameters and, often, tunes on every episode. In each episode, a validation accuracy has been computed after training for 50 steps with a batch size of 32. The delta of the accuracy, which is referred to as reward, is fed back to the policy network along with the behavior of the environment. The attributes that represent behavior are training loss and statistics of the target network's final layer outcome. A more refined tuple of hyper-parameters, in turn, is generated for the next episode. This cycle tunes

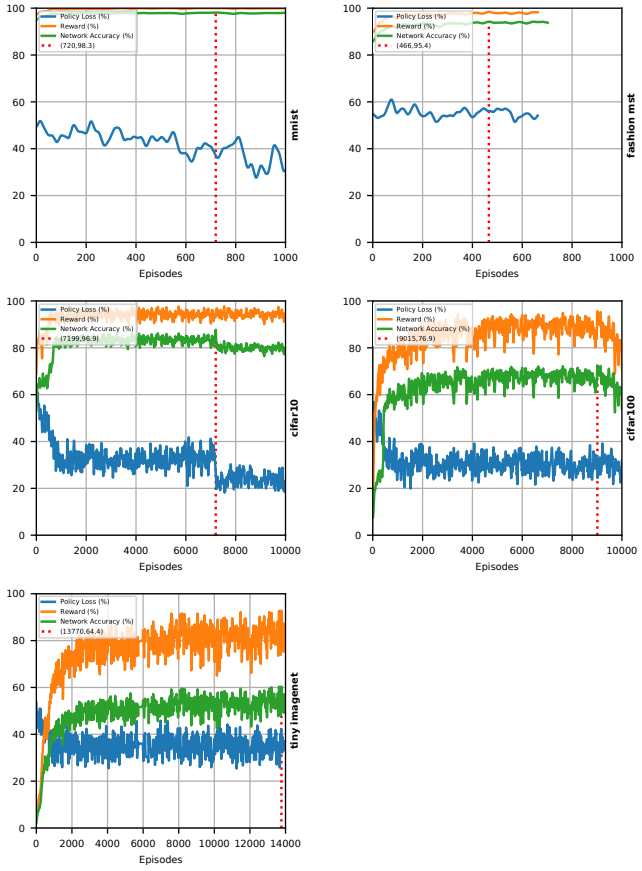

Fig. 3: Statistics of different datasets including policy loss, reward and network accuracy

the parameters of the policy network and hyper-parameters of the network at the same time which makes the overall process more computationally efficient than the existing approaches.

In conclusion, the proposed approach demonstrates a quick and effective hyper-parameter search approach. Unlike previous studies, it is equally effective for both small and large datasets. Although the exploration takes more time if the range of the network depth parameter gets bigger, still using one GPU the exploration takes less than a day for a complex task. This approach is 20% less computation expensive than ENAS with marginally higher error-rate. The depth hyper-parameter is found to be the most effective one where the change of the depth causes a significant jump in the accuracy. There are many possible directions for future work. Currently, only four hyper-parameters are part of the search space which can be enhanced. Accordingly, to evaluate the effectiveness of a tuple of hyper-parameters, state of the intermediary layers of the network can be observed rather than only the statistics of final layer outputs.

References

1. Ali, A., Budka, M., Gabrys, B.: Towards meta-level learning of deep neural networks for fast adaptation. Proceedings of the 16th Pacific RIM International Conference on Artificial Intelligence (PRICAI) (2019)
2. DeVries, T., Taylor, G.W.: Improved regularization of convolutional neural networks with cutout. Computing Research Repository (CoRR) **abs/1708.04552** (2017)
3. Duan, Y., Schulman, J., Chen, X., Bartlett, P.L., Sutskever, I., Abbeel, P.: Rl2: Fast reinforcement learning via slow reinforcement learning. Computing Research Repository (CoRR) **abs/1611.02779** (2016)
4. Finn, C., Abbeel, P., Levine, S.: Model-agnostic meta-learning for fast adaptation of deep networks. In: Proceedings of the 34th International Conference on Machine Learning. vol. 70, pp. 1126–1135. PMLR, International Convention Centre, Sydney, Australia (8 2017)
5. Finn, C., Levine, S.: Meta-learning and universality: Deep representations and gradient descent can approximate any learning algorithm. Computing Research Repository (CoRR) **abs/1710.11622** (2018)
6. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feed-forward neural networks. Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, PMLR (2010)
7. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 770–778 (2016)
8. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Computation pp. 1735–1780 (1997)
9. Huang, G., Sun, Y., Liu, Z., Sedra, D., Weinberger, K.: Deep networks with stochastic depth. Computing Research Repository (CoRR **abs/1603.09382** (2016)
10. Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. International Conference of Machine Learning (ICML) (2015)
11. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. International Conference on Learning Representations (ICLR) (2015)
12. Krizhevsky, A., Nair, V., Hinton, G.: Cifar-10 and cifar-100. Canadian Institute for Advanced Research
13. Le, Y., Yang, X.: Tiny imagenet visual recognition challenge. Stanford CS 231N (2015)
14. LeCun, Y., Cortes, C., Burges, C.J.C.: The mnist dataset of handwritten digits (1999)
15. Liu, C., Zoph, B., Neumann, M., et al.: Progressive neural architecture search. Computing Research Repository (CoRR) **abs/1712.00559** (2018)
16. Nair, V., Hinton, G.E.: Rectified linear units improve restricted boltzmann machines. International Conference of Machine Learning (ICML) (2010)
17. Pham, H., Guan, M.Y., Zoph, B., Le, Q.V., Dean, J.: Efficient neural architecture search via parameter sharing. Computing Research Repository (CoRR) **abs/1802.03268** (2018)
18. Ravi, S., Larochelle, H.: Optimization as a model for few-shot learning. International Conference on Learning Representations (ICLR) (2017)
19. Sutskever, I., Martens, J., Dahl, G., Hinton, G.E.: Practical network blocks design with q-learning. International Conference of Machine Learning (ICML) (2013)

20. Sutton, R.S., McAllester, D., Singh, S., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. NIPS (1999)
21. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning pp. 41–49 (2019)
22. Xiao, H., Rasul, K., Vollgraf, R.: Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. Computing Research Repository (CoRR) **abs/1708.07747** (2017)
23. Xu, T., Liu, Q., Zhao, L., Peng, J.: Learning to explore with meta-policy gradient. Computing Research Repository (CoRR) **abs/1803.05044** (2018)
24. Zoph, B., Le, Q.V.: Neural architecture search with reinforcement learning. International Conference on Learning Representations (ICLR) (2017)
25. Zoph, B., Vasudevan, V., Shlens, J., Le, Q.V.: Learning transferable architectures for salable image recognition. Computer Vision and Pattern Recognition (CVPR) (2018)