Neural Architecture Search Using Automated Planning

Felipe Roque Tasoniero
School of Technology
Pontifical Catholic University of Rio Grande do Sul
Porto Alegre - Brazil
felipe.tasoniero@edu.pucrs.br

Abstract

Due to recent interest in designing neural architecture without human assistance, there has been a great effort to find methods that generate high-performing CNN architecture with less computational time processing. Traditional techniques ~~as~~ Reinforcement Learning and Evolutionary Algorithms are widely used in Neural Architecture Search (NAS) research, but they are costly to compute. In this paper, we propose to explore a based RL algorithm, known as MetaQNN, to automatically generate CNN architectures and implement a penalize configuration to reduce computational cost.

Introduction

It's known that state-of-the-art neural network architectures require a lot of effort of human experts to be discovery. Recently, Neural Architecture Search research field has been widely investigated due to its capacity to automatically find neural network architectures that perform as well as the ones designed manually (Elsken, Metzen, and Hutter 2018).

Two main techniques that are usually used to solve NAS problems: Reinforcement Learning (RL) (Zoph and Le 2016), and Evolutionary Algorithms (EA) (Miikkulainen et al. 2019). In Evolutionary Algorithms, each neural network structure is encoded as a string, and random mutations and recombinations of the strings are performed during the search process; each string is then trained and evaluated on a validation set, and the top-performing models generate "children" (Liu et al. 2018). When using reinforcement learning, the agent performs a sequence of actions, which specifies the structure of the model; this model is then trained, and its validation performance is returned as the reward, which is used to update the controller (Liu et al. 2018).

Although Evolutionary Algorithms have shown significant progress in designing high-performing neural architectures, RL algorithms are still considered a go-to for NAS. An approach to solve NAS problems is using a Q-learning algorithm, a type of reinforcement learning.

Even though the Q-learning algorithm is less costly than EA algorithms, they are still a problem for stand-alone researchers to train it. A solution to this problem was the implementation of MetaQNN (Baker et al. 2016). It was shown that MetaQNN took 8-10 days to complete the training for each dataset using 10 GPUs.

In this paper, we intend to explore the use of MetaQNN to search a high-performing CNN architecture for the MNIST dataset, aiming at the reduction of GPUs/hour cost by penalizing the reward function.

Related Word

Some different strategies, as a random search, Bayesian optimization, evolutionary algorithms, and reinforcement learning, can be used to explore the space of NAS.

In 2015, a Bayesian optimization algorithm reached the state-of-the-art for CIFAR-10 classification task (Domhan, Springenberg, and Hutter 2015). The Neural Search Architecture problem become popular in the machine learning community in 2016 when a reinforcement learning algorithm obtained a competitive performance on the CIFAR-10 (Zoph and Le 2016).

Recently, works related to the NAS research field that makes use of RL algorithms has proven its capability to generate high-performing neural architectures to deal with specific~~s~~ machine learning and deep learning problems (Zoph et al. 2018; Suganuma, Shirakawa, and Nagao 2017; Cortes et al. 2017; Negrinho and Gordon 2017; Brock et al. 2017). Due to the known problem of processing time consume related to NAS, many search models were developed to deal with it.

In the search space models, two types of search have showed a high accuracy in neural architectures generated. One of them is the Cell Search Space, where an agent selects cells based on a reward, where these cells are composed of many feature layers, and then these cells are stacked to generate a new neural architecture (Tan et al. 2019). The other one is the Global Search Space, where an agent selects layers based on a reward to create a new neural architecture (Zoph and Le 2016).

Although the capacity of these models in generating high-performing CNN architectures, they are very similar considering the computational cost. To solve this problem, an RL algorithm called MetaQNN was proposed to reduce the computational time consuming significantly (Baker et al. 2016). Some improvements to this model were proposed too by *Baker et all* (Baker et al. 2017).

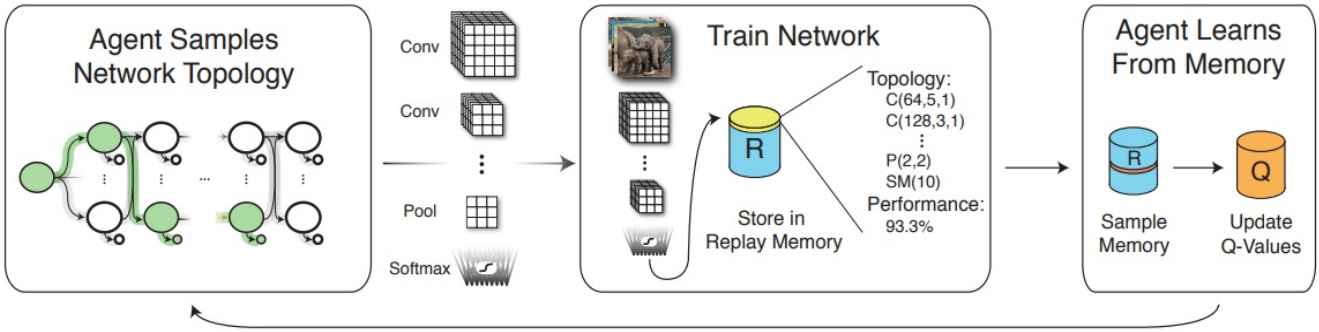

Figure 1: MetaQNN automated process for Neural Architecture Search using Q-learing (Baker et al. 2016).

Method

The automated process of CNN architecture selection through MetaQNN, summarized in Figure 1, consists of incorporating Q-learning agent, ϵ-greedy strategy, and experience replay. The generation of a Neural Network is due to the agent's action, considering the action space and state space, and the agent's reward that is based on the performance estimated by the trained Neural Network.

The Q-learning algorithm is used as an agent to find optimal paths as a Markov Decision Process (MDP) in a finite-horizon environment, as can be seen in Figure 2. The environment here is restricted to a discrete and finite space $\boldsymbol{S}$, as well as action space $\boldsymbol{A}$. So, for any state $S_t \in \boldsymbol{S}$ the agent can choose some action $A_t \in \boldsymbol{A}$ and get a reward R.

The agent's goal is to maximize the reward over all possible paths. As can be seen in 2, this process starts with arbitrary $Q(S, A)$ values. Given an execution represented by the states (S_t), actions (A_t) and rewards (R_t) received at time t, Q-learning updates the value of the maximum total expected reward $Q(S, A)$ of an action A_t in the state S_t after observing the next state S_{t+1} and getting the reward R_{t+1} through the expression formulated as

$$Q(S_t, A_t) := (1 - \alpha_t)Q(S_t, A_t) + \alpha_t D_t(S_t, A_t) \quad (1)$$

$$D_t(S_t, A_t) = R_{t+1} + \gamma max_{A_t \in A(S_{t+1})} Q(S_{t+1}, A_t) \quad (2)$$

where α is the Q-learning rate and and γ is the discount factor. The Q-learning rate can determine the weight given to new information over old information, and the discount factor can determine the weight given to short-term rewards over future rewards.

The Q-learning algorithm is considered *model-free* for solving MDPs as it learns the behavior but not the model, so the learning agent can solve the task without explicitly constructing an estimate of environmental dynamics. Q-learning is considered *off-policy*, too, meaning it can learn about optimal policies while exploring via a non-optimal behavioral distribution, i.e., the distribution by which the agent explores its environment (Baker et al. 2016).

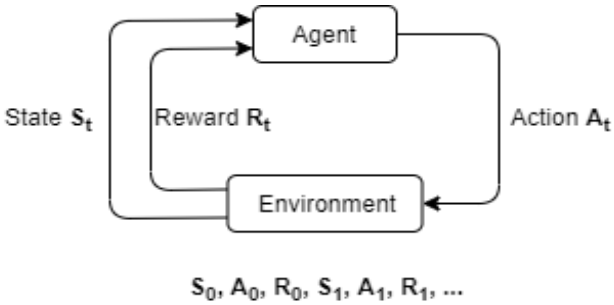

Figure 2: Q-learning update process.

The MetaQNN algorithm uses an ϵ-greedy strategy as the behavior distribution, so that ϵ is the probability ~~of a taken random~~ action and the 1 - ϵ is the probability of a greedy action given by $max_{A_t} Q(S_t, A_t)$. The motivation for ϵ-greedy strategy is that we want to find a way of improving policies while ensuring that we explore the environment. The ϵ varies from 1 to 0, such that the agent starts *exploring* and slowly starts moving towards the *exploitation* phase. So, as the state space for the exploration phase is considered large, MetaQNN algorithm uses the *experience replay* in a manner that at a given interval, the agent samples from memory and update its Q-values via Equation 1.

To design and train a neural network, the agent sequentially selects layers in a state space via the ϵ-greedy strategy until it reaches a termination state. The neural network architecture defined by the agent's path is trained to classify the images. After the training is over, the agent is given a reward equal to the validation accuracy. To update the the Q-learning values in Equation 1, the parameters α and γ were set, respectively, to 0.01 and to 1. Setting the discount factor to 1 gives the ability to the agent do not over-prioritize short-term rewards.

During the training process, the ϵ decreases from 1.0 to 0. For the ϵ value equals to 1.0, we set the number of models to be trained to 250. Then, for ϵ values between 0.9 to 0.5, we reduce the amount of models to be trained to 20. For the ϵ values between 0.4 to 0.0, we set the number of training models to 30 and 40. We begin with a high number of train-

ing models due to the exploration phase, given the agent the ability to explore different kinds of architectures. When the ϵ begins to reduce, we reduce the number of training models to exploit more than explore. When the value of ϵ gets closer to zero, we add more number of training models, to let the agent exploit more the states. For each model generated during the entire training process, the replay dictionary stores the network topology and the prediction performance on a validation set. The experience replay dictionary is used to prevent that the model could be re-trained if it has already been trained. So, when the model is re-sampled, the previously found validation accuracy is presented to the agent.

The state space is defined as a tuple of all relevant layers parameters. The MetaQNN algorithm allows five different types of layers: convolution (C), pooling (P), fully connected (FC), global average pooling (GAP), and softmax (SM), though the general method is not limited to this set. Figure 3 shows the relevant parameters for each layer type and also the discretization we chose for each parameter.

Layer Type	Parameters	Values
Convolution (C)	*(i, f, l, d, n)*	<12, {1, 3, 5}, 1, {64,128,256,512}, {(∞, 8], (8, 4], (4, 1]}
Pooling (P)	*(i, (f, l), n)*	< 12, {(5,3), (3,2), (2,2)}, {(∞, 8], (8, 4], (4, 1]}
Fully Connected (FC)	*(i, n, d)*	<12, <3, {512, 256, 128}
Termination State	*(s, t)*	Global Avg. Pooling, Softmax

Figure 3: The state space for classification task. The parameters are: Layer depth *(i)*, Receptive field size *(f)*, Stride *(l)*, Receptive fields *(d)*, Representation size *(n)*, Previous state *(s)* and Type *(t)*

The actions that the agent can perform on MetaQNN algorithm are restricted. In MetaQNN algorithm, the agent is allowed to terminate a path at any point, i.e., it may choose a termination state from any non-termination state. In addition, the transitions are only allowed for a state with layer depth i to a state depth $i+1$. Any state with the maximum layer depth, as shown in Figure 3, may only transition to a termination layer. The number of fully connected layers is set to a maximum two because the number of learnable parameters would be high, so it will take much more time to be trained.

For the state transitions, the agent can only take some specific actions for its actual state space. When an agent is at a state of type Convolution (C), it may take an action to go through to a state with any other layer type. When the agent is at a Pooling (P) state, it may transition to the same type Pooling layer, or it can go through any other layer type. The transition from a Pooling layer to another Pooling layer are allowed because two or more consecutive Pooling layers are equivalent as one. The transitions from a state space to a FC layer state are only allowed for states with representation size in bins (8, 4] and (4, 1]. These majority of these constraints are in place to enable faster convergence due to limited hardware and not a limitation of the method in itself (Baker et al. 2016).

The strategies of using the experience replay while the agent is transiting from a state to another one can reduce the computational cost significantly. Another option to reduce this cost is to apply a penalize configuration to the reward function. One of the possibilities for the penalize configuration is to use the early stop function.

In work from *Baker et all* (Baker et al. 2017), they propose an early stop configuration that models a validation accuracy of a neural network architecture at each epoch using previous observations. They record a time-series of validation accuracy from each CNN trained. So, they predict the performance observation of a neural network training a regression model over a subset from the time-series record. Here, we propose to investigate the use of a much simpler early stop. We set a threshold equals to 0.25 for the validation accuracy value for each epoch from a given neural network architecture. So, if the validation accuracy in a given epoch is equal or below the threshold value, then the training for the actual architecture stops, the Q-learning value is not updated, and the agent moves for the next state space.

Experiment and Results

For our experiments, we evaluate a MetaQNN algorithm to search a neural network for classification tasks over the MNIST dataset. In this work, we didn't use the original implementation of MetaQNN, due to its requirements that were not compatible with our hardware specifications. So, we use a MetaQNN algorithm implemented using the Pytorch framework, and we adapt it to allow the NAS training over the MNIST dataset.

As one of our goals was to reduce the amount of hardware resources aiming at less computational time cost, we made some changes to originals parameters. First, we reduce the total amount of models generated for each ϵ-greedy value in the schedule. We also change the optimizer parameter from Stochastic gradient descent (SGD) to Adam Optimizer, due to the Adam Optimizer converging faster than SGD, we reduce the number of epochs of training from 40 to 10 for each neural network architecture, and to penalize the model, we apply the early stop configuration using the threshold for validation accuracy in each training epoch. We also record each model generated topology and validation accuracy value on the replay dictionary, and we search for replay experience in each transition state.

As we are still evaluating some experiments using the MetaQNN algorithm, the results obtained until this date show that our model is possibly overfitting. The validation accuracy for some architectures are higher than the training accuracy.

Conclusion

In this this work, we aim to explore the use of MetaQNN for NAS due to its less necessity of GPUs/hour usage comparing

to traditional RL algorithms for this task. Considering that, we propose to use whey to penalize the agent's reward.

To this date we did not manage to get the complete results for the classification task on MNIST dataset using the MetaQNN algorithm. Our prior results shows that the training process are possibly overfitting. We think that this could be fixed by search for weight regularization parameters and possibly for others CNN hyperparameters.

The prior results also show that our changes help to reduce the time significantly consuming during the process. The number of models generated for each epsilon-greedy value shows a considerable influence in time consumption. Having a high number of generated architectures leads the model to explore and find a new possible high-performing model to a given task, but this can be very costly. On the other hand, having a small number of generated architectures prevent the model from finding a high-performance architecture, but can reduce the computational time cost significantly. We think that knowing the state-space search for a given problem, and it could be possible to look for an optimal number of neural network architecture generated for each epsilon-greedy step.

For future work, we will continue making improvements on the MetaQNN algorithm due to its high capacity to search new neural network architectures comparing to other models. For the next steps, we aim to define a more general and flexible search spaces, trying to generalize the search for new architectures to other domains. It could give to NAS the capacity to solve multi-task problems and multi-objective problems. We can see that NAS could be used in an end-to-end manner for machine learning tasks in a way that, given a particular domain, we can search for a high-performance model to solve the task without previous knowledge in this specific domain. Another possible improvement that could be done is to allow NAS to identify novel building blocks with different kinds of convolutions and pooling layers, making NAS more broadly applicable.

References

Baker, B.; Gupta, O.; Naik, N.; and Raskar, R. 2016. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*.

Baker, B.; Gupta, O.; Raskar, R.; and Naik, N. 2017. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*.

Brock, A.; Lim, T.; Ritchie, J. M.; and Weston, N. 2017. Smash: one-shot model architecture search through hypernetworks. *arXiv preprint arXiv:1708.05344*.

Cortes, C.; Gonzalvo, X.; Kuznetsov, V.; Mohri, M.; and Yang, S. 2017. Adanet: Adaptive structural learning of artificial neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 874–883. JMLR. org.

Domhan, T.; Springenberg, J. T.; and Hutter, F. 2015. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *Twenty-Fourth International Joint Conference on Artificial Intelligence*.

Elsken, T.; Metzen, J. H.; and Hutter, F. 2018. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*.

Liu, C.; Zoph, B.; Neumann, M.; Shlens, J.; Hua, W.; Li, L.-J.; Fei-Fei, L.; Yuille, A.; Huang, J.; and Murphy, K. 2018. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 19–34.

Miikkulainen, R.; Liang, J.; Meyerson, E.; Rawal, A.; Fink, D.; Francon, O.; Raju, B.; Shahrzad, H.; Navruzyan, A.; Duffy, N.; et al. 2019. Evolving deep neural networks. In *Artificial Intelligence in the Age of Neural Networks and Brain Computing*. Elsevier. 293–312.

Negrinho, R., and Gordon, G. 2017. Deeparchitect: Automatically designing and training deep architectures. *arXiv preprint arXiv:1704.08792*.

Suganuma, M.; Shirakawa, S.; and Nagao, T. 2017. A genetic programming approach to designing convolutional neural network architectures. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 497–504. ACM.

Tan, M.; Chen, B.; Pang, R.; Vasudevan, V.; Sandler, M.; Howard, A.; and Le, Q. V. 2019. Mnasnet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2820–2828.

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

Zoph, B.; Vasudevan, V.; Shlens, J.; and Le, Q. V. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 8697–8710.