Neural Architecture Search with Bayesian Optimization

Tong GAO, Xuhua HUANG, Jiarui XU

May 3, 2018

Abstract

Exploring a better Neural Network Architecture for Image Classification has been an actively researched topic in Deep Learning. In our project, we shed our light on using Bayesian Optimization to automatically optimize Neural Network Architecture for classifying CIFAR-10 Image Dataset.

1 Introduction

Based on the recommendation from TA, we decided to use Bayesian Optimization on architecture rather than just hyper-parameters. In the project, we generate many new architectures by mutations from origin pool of networks, evaluate the acquisition function of them, and evaluate the networks with highest acquisition value. In order to get a reasonable kernel function, We follow the idea from [KNS+18], using OTMANN distance to measure the distance between two architectures and defining a new kernel on it for Gaussian Process. Some improvements to get a more reasonable result, like marginalizing of hyper-parameters in acquisition function are also done.
Our codes are available in: https://github.com/gaotongxiao/Bayesian_Optimization_In_Neural_Architecture

2 Project Highlights

2.1 OTMANN Distance

Layer mass: Define layer mass lm as the product of number of incoming units and the units in this layer. Layer mass can be regarded as the computation done in this layer, i.e. the "significance" of the layer. For input and output layers, we cannot directly use this definition, instead we let $lm(u_{ip}) = lm(u_{op}) = \zeta \sum_{u \in PL} lm(u)$ where lm denotes layer mass and PL denotes the set of processing layers (convolution layers, pool layers and fully connect layers, etc.). Moreover, since the significance of decision layers (softmax) should be dependent on all the layers in the graph, it follows a similar definition: $\forall u \in DL, lm(u) = \frac{\zeta}{|DL|} \sum_{u \in PL} lm(u)$. In our project, we use $\zeta = 0.1$.

Path length: The position of layer u is encoded in the definition of path length $\delta(u)$. To get the value, we take average of six kinds of path length: the longest, shortest and random-walk path length of u from input layer and output layer, where random-walk path length is the path length of randomly chosen path from input/output layer to layer u.

Now we're ready to define distance between two architecture graph G_1 and G_2. Suppose G_1 has n_1 layers and G_2 has n_2 layers, we let $Z \in \mathbb{R}_+^{n_1 \times n_2}$ be such that $Z(i, j)$ denotes the amount of mass matched between layer $i \in G_1$ and $j \in G_2$. Distance is computed by solving:

$$d = \min_Z \phi_{lmm}(Z) + \phi_{nas}(Z) + v_{str}\phi_Z$$

subject to $\sum_{j \in G_2} Z_{ij} \leq lm(i), \sum_{i \in G_1} Z_{ij} \leq lm(j), \forall i, j$.

The label mismatch penalty $\phi_{lmm}(Z)$: We begin with define a matrix $M \in \mathbb{R}^{L \times L}$ where L is the number of all label types and $M(x, y)$ denotes the penalty for match layer mass from type of x to y. The definition can be found in 1. Then define a matrix $C_{lmm}(i, j) = M(ll(i), ll(j))$ where $ll(u)$ means the layer type of u, which means the penalty from layer i in G_1 to layer j in G_2. Then

	c3	c5	c7	mp	ap	fc	sm
c3	0	0.2	0.3				
c5	0.2	0	0.2				
c7	0.3	0.2	0				
mp				0	0.25		
ap				0.25	0		
fc						0	
sm							0

Table 4. The label mismatch cost matrix M we used in our CNN experiments. $M(\mathtt{x}, \mathtt{y})$ denotes the penalty for transporting a unit mass from a layer with label `x` to a layer with label `y`. The labels abbreviated are `conv3`, `conv5`, `conv7`, `max-pool`, `avg-pool`, `fc`, and `softmax` in order. A blank indicates ∞ cost. We have not shown the `ip` and `op` layers, but they are similar to the `fc` column, 0 in the diagonal and ∞ elsewhere.

Figure 1: The mismatch cost matrix we used

we make $\phi_{lmm}(Z) = Z \cdot C_{lmm}$.

Non-assignment penalty ϕ_{nas}: This is set to be the layer mass that is unassigned in both graphs, where $\phi_{nas} = \sum_{i \in G_1}(lm(i) - \sum_{j \in G_2} Z_{ij}) + \sum_{j \in G_2}(lm(j) - \sum_{i \in G_1} Z_{ij})$.

The structural penalty term ϕ_{str}: Define a matrix $C_{str} \in \mathbb{R}^{n_1 \times n_2}$ and $C_{str}(i, j) = |\delta(i) - \delta(j)|$, which represents the structural difference of $i \in G_1$ and $j \in G_2$. Then $\phi_{str} = Z \cdot C_{str}$.

Minimizing the above three terms means that we want to make the layer masses between layers of two graphs to be matched as many as possible, while ensuring the large proportion of layer masses will only be matched to the most similar (in terms of node type and structure) layers in other graph. The sum of three costs therefore stands for the difference between two graphs.

We define $\bar{d}(G_1, G_2) = d(G_1, G_2)/(tm(G_1) + tm(G_2))$ where $tm(G_i) = \sum_{\mu \in G_i} lm(\mu)$ is the total layer mass of the graph G_i. It provides a useful measure of similarity normalized by the amount of computation. [KNS+18] suggests that d emphasizes more on the computation at the layers of the graph and vice versa for $\bar{d}$.

2.2 The Kernel

The distance defined in the previous section by minimization. Now we could construct our kernel. Typically, we choose a negative exponentiated distance as our kernel [KNS+18]. Specifically, we use

$$\kappa(\cdot, \cdot) = \alpha e^{-\sum_i \beta_i d_i^p(\cdot,\cdot)} + \bar{\alpha} e^{-\sum_i \bar{\beta}_i \bar{d}_i^{\bar{p}}(\cdot,\cdot)}$$

The d_i and $\bar{d}_i$ are the OTMANN distance and its normalization in the previous section, computed with different $\nu_{\text{str}} \in \{\nu_{\text{str},i}\}_i$. The relative contribution is controlled by β_i and $\bar{\beta}_i$, while $\alpha, \bar{\alpha}$ are the weights of unnormalized and normalized distance. The above ensemble style designed kernel will be more robust and less sensitive to one of the hyperparameters. In our experimentation, we set $p = 1, \bar{p} = 2$.

2.3 Acquisition Optimization

2.3.1 Acquisition Function Definition

In Bayesian Optimization, we need to optimize a function f over a domain χ. At time step t, we have already evaluated the function at points $\{x_i\}_{i=1}^{t-1}$, and obtained observations $\{y_i\}_{i=1}^{t-1}$. To determine the next point for evaluation x_t, we need to define an *acquisition function* a_t which is a function of posterior and is a measure of the value of evaluating f at any $x \in \chi$. Here we decided to use Expected Improvement (EI), below is the definition from lecture note [Gar18]

Suppose that f' is the minimal value of f observed so far. Expected improvement evaluates

f at the point that, in expectation, improves upon f' the most. This corresponds to the following utility function:

$$u(x) = \max\big(0, f' - f(x)\big).$$

The expected improvement acquisition function is then the expected utility as a function of x:

$$\begin{aligned} a_{\text{EI}}(x) = \mathbb{E}\big[u(x) \mid x, \mathcal{D}\big] &= \int_{-\infty}^{f'} (f' - f)\,\mathcal{N}\big(f; \mu(x), K(x,x)\big)\,\mathrm{d}f \\ &= \big(f' - \mu(x)\big)\Phi\big(f'; \mu(x), K(x,x)\big) + K(x,x)\mathcal{N}\big(f'; \mu(x), K(x,x)\big). \end{aligned}$$

According to our project setting, x represents a specific Neural Network architecture, $f(x)$ represents the negative log of testing accuracy for architecture x in CIFAR-10 Dataset, f' represents the minimal value of f observed so far.

2.3.2 Optimize Acquisition Function

We use Evolutionary algorithm (EA) to optimize this acquisition function:

- Begin with an initial pool of networks and evaluate the acquisition $a(x)$ on those networks
- Then we generate a set of N_{mut} mutations of this pool as follows:
  - Stochastically select N_{mut} candidates from the set of networks already evaluated such that those with higher $a(x)$ values are more likely to be selected than those with lower values.
  - Apply a mutation operator (Figure 2) to each candidate, to produce a modified architecture.
  - Evaluate the acquisition on this N_{mut} mutations, add it to the initial pool, and repeat for the prescribed number of steps

Operation	Description
dec_single	Pick a layer at random and decrease the number of units by 1/8.
dec_en_masse	First topologically order the networks, randomly pick 1/8 of the layers (in order) and decrease the number of units by 1/8. For networks with eight layers or fewer pick a 1/4 of the layers (instead of 1/8) and for those with four layers or fewer pick 1/2.
inc_single	Pick a layer at random and increase the number of units by 1/8.
inc_en_masse	Choose a large sub set of layers, as for dec_en_masse, and increase the number of units by 1/8.
dup_path	This modifier duplicates a random path in the network. Randomly pick a node u_1 and then pick one of its children u_2 randomly. Keep repeating to generate a path $u_1, u_2, \ldots, u_{k-1}, u_k$ until you decide to stop randomly. Create duplicate layers $\tilde{u}_2, \ldots, \tilde{u}_{k-1}$ where $\tilde{u}_i = u_i$ for $i = 2, \ldots, k-1$. Add these layers along with new edges $(u_1, \tilde{u}_2)$, $(\tilde{u}_{k-1}, u_k)$, and $(\tilde{u}_j, \tilde{u}_{j+1})$ for $j = 2, \ldots, k-2$.
remove_layer	Picks a layer at random and removes it. If this layer was the only child (parent) of any of its parents (children) u, then adds an edge from u (one of its parents) to one of its children (u).
skip	Randomly picks layers u, v where u is topologically before v and $(u, v) \notin \mathcal{E}$. Add (u, v) to $\mathcal{E}$.
swap_label	Randomly pick a layer and change its label.
wedge_layer	Randomly pick any edge $(u, v) \in \mathcal{E}$. Create a new layer w with a random label $\ell\ell(w)$. Remove (u, v) from $\mathcal{E}$ and add $(u, w), (w, v)$. If applicable, set the number of units $\ell u(w)$ to be $(\ell u(u) + \ell u(v))/2$.

Table 6. Descriptions of modifiers to transform one network to another. The first four change the number of units in the layers but do not change the architecture, while the last five change the architecture.

Figure 2: Mutator Operator Table [KNS+18]

2.4 Hyper-parameters Marginalization

For the fully-Bayesian treatment of hyper-parameters (denoted by θ here), it is desirable to marginalize over hyper-parameters and compute the *integrated acquisition function* [SLA12]. Under this strategy, the effort of carefully selecting the kernel hyper-parameters is saved. Moreover, better acquisition values is obtained. Even though it takes a little bit longer to compute the acquisition function by MCMC, the time and computation cost is marginal compared to evaluating the whole model on GPU.

$$\hat{a}(\mathbf{x}; \{\mathbf{x}_n, y_n\}) = \int a(\mathbf{x}; \{\mathbf{x}_n, y_n\}, \theta) p(\theta | \{\mathbf{x}_n, y_n\}_{n=1}^{N}) \mathrm{d}\theta$$

2.5 Bayesian Optimization

Since we need to find the next promising CNN architecture, we propose to do it by Bayesian Optimization.

2.5.1 Initial Pool Construction

We first construct basic initial pool using prior knowledge of CNN architecture, e.g. typical convolution and pooling stack structure. It is essential that we should make our initial pool both simple and diverse. It implies that we should include all kinds of layers so that the Evolution Algorithm could have enough models to mutate. We construct our initial pool from simple models with 3-5 layers. This will help to show that our Bayesian Optimization System would be able to automatically learn to construct extremely complex model step by step. The boosting results should be expected in a few number of steps.
To kick start, we evaluate the accuracy of all the models in the initial pool. In our implementation, there are 3 models in the initial pool with different structure, which includes all layer types.

2.5.2 Mutation

At the t^{th} step of Bayesian Optimization, we stochastically select t model candidates out of our pool based on the accuracy, where the models with higher accuracy are more likely to be selected. Then we perform mutate operation on these t candidates and compute corresponding acquisition values. The models with $\lfloor\sqrt{t}\rfloor$ highest acquisition value would be evaluated accuracy and add into the pool afterwards.
This mutation step is performed several times, as the number of step increase, there are more models in the pool. The models generated are becoming more complex. The performance of the models generated from Bayesian Optimization will be discussed in the next section.

3 Experiments and Results

3.1 Architecture Preference

The batch normalization and residual network block will help the gradient flow in the very deep CNN. 5+ layers CNN without batchnorm or ResNet Block is extremely (almost impossible for 10+ layers) to train in one single stage.

Since in the initial pool, the CNN with batchnorm and ResNet Block have higher evaluation accuracy, after a few steps of mutation and Bayesian Optimization, we got CNN architectures with more batchnorm node and ResNet node, whose accuracy is reasonably higher.

This indicates that after Bayesian Optimization, our system started to prefer models with better architecture components (i.e. batchnorm and ResNet), because the acquisition function could help it choose models with higher accuracy.

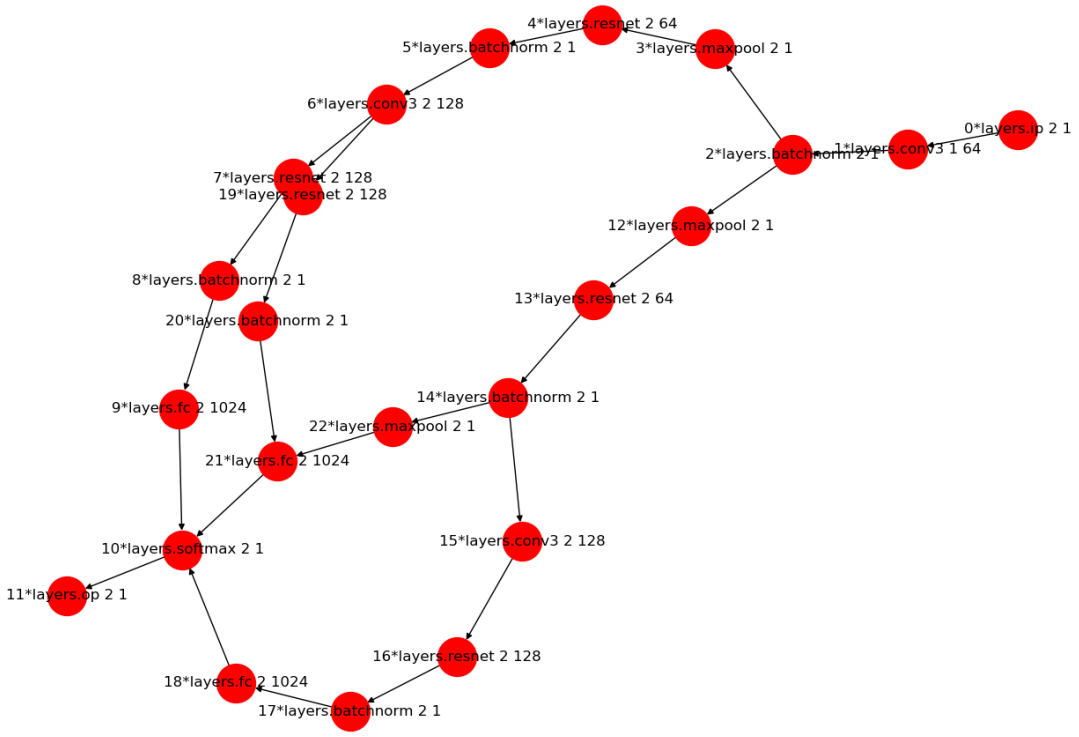

Figure 3: An architecture created and chosen by Bayesian Optimization with accuracy of 0.72

3.2 Accuracy Improvement

As we can see from Figure 4, the accuracy of original models from initial pool is lower than 0.50. After a few optimization steps (i.e. around 8 steps), the accuracy of new models has reached 0.70, which has been a reasonable accuracy for CNN architecture created artificially with only these layer types.

After we added more types of layers, such as dropout layer, into our mutation choices, we could get architectures with accuracy of more than 0.82 for Figure 3.

Due to the limitation of training resources (e.g. GPU), we have not yet made deeper network architecture available for our Bayesian Optimization system. However, we believe that when the number of models in our pool increases, and the depth of models becomes larger, accuracy of the architecture chosen by our Bayesian Optimization system will increase as well.

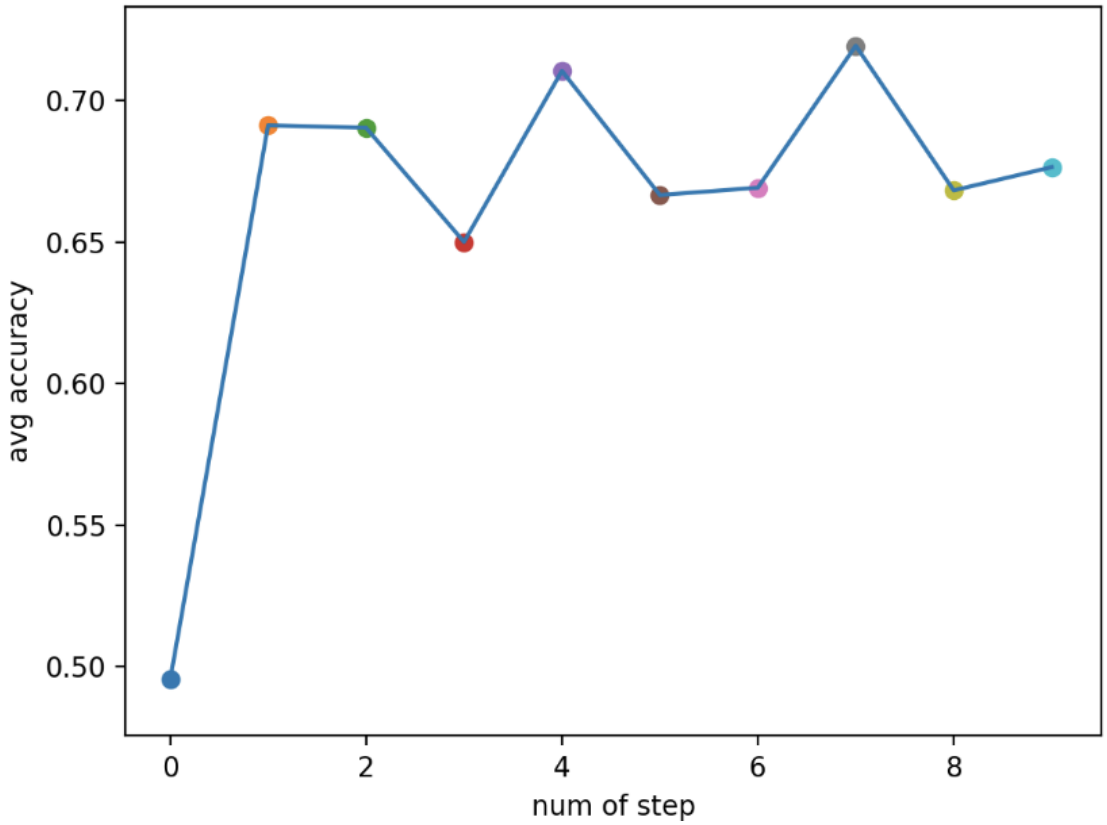

Figure 4: Avg. Accuracy after a few steps

References

[Gar18] Roman Garnett. Materials for bayesian methods in machine learning course. 2018.

[KNS+18] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabás Póczos, and Eric Xing. Neural architecture search with bayesian optimisation and optimal transport. *CoRR*, abs/1802.07191, 2018.

[SLA12] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. *ArXiv e-prints*, June 2012.