Learning to learn by gradient descent by gradient descent

Liyan Jiang

July 18, 2019

1 Introduction

The general aim of machine learning is always learning the data by itself, with as less human efforts as possible. Then it comes to the focus that if there exists a way to design the learning method automatically using the same idea of learning algorithm. In general, machine learning problems are usually optimization problems. Basically we try to parameterize an objective function that describes the real life problem and solve it by convex optimization. Most state-of-the-art optimizers like RMSprop, ADAM, NAG require manual adjustment of hyper-parameters and need human inspection when applying to different kinds of problems. This paper introduce a method to learn the update rule of parameters instead of hand-crafted it. So that we can replace the hand-crafted optimizers with a learned optimizer, saving a lot of human efforts.

One challenge of using learned optimizer is how it can transfer what it learned. To this aim, the authors design plenty of experiments to see how this learned optimizer apply to different sorts of problems by comparing with hand-crafted optimizers. In addition, they also test if some modification to the architecture will affect the performance of the optimizer.

2 Methodology

To perceive the problem in a higher level, the task consists of an optimizer and an optimizee. As Figure 1 shows, the gradients of optimizee parameters θ are error signals that feed into the optimizer as an input. The optimizer, parameterized with ϕ, calculates the parameter update as outputs. In the next round, the optimizee update its parameter using the output from the optimizee and the iteration goes on.

To put it in mathematical form, the authors introduce a learned update rule $g(\phi)$ that replaces hand-designed update rules, as the formula 1 shows.

$$\theta_{t+1} = \theta_t + g_t(\nabla f(\theta_t), \phi) \tag{1}$$

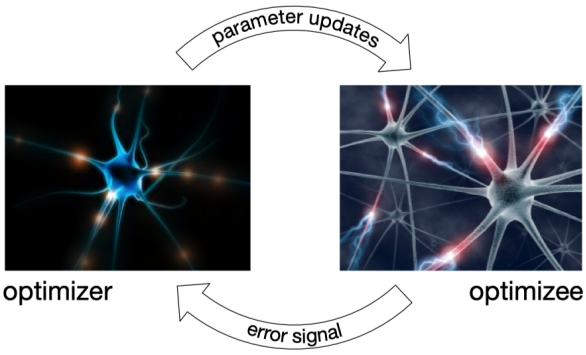


Figure 1: Optimizer and optimizee

The interaction of optimizer and optimizee is analogous to the controller and child network introduced in [4]. In that paper, they use a RNN controller to generate hyper-parameters of child neural networks and train them with reinforcement learning. The accuracy of child network is regarded as a reward that the controller wants to maximize the expectation of. However, this reward is non-differentialble. That's why a policy is needed to update the hyper-parameters.

$$\mathcal{L}(\phi) = E[f(\theta^*(f, \phi))] \tag{2}$$

$$\theta_{t+1} = \theta_t + g_t \begin{pmatrix} g_t \\ h_{t+1} \end{pmatrix} = m(\nabla_t, h_t, \phi) \tag{3}$$

As a comparison, the method introduced in this paper is fully supervised, so that the loss function 2 is differentiable. In this equation, we want to minimize the expectation of function f, which is actually a distribution of functions and is randomly initialized. The target function f uses the optimal parameter θ, which comes out of an update policy that takes function f and optimizee parameter ϕ as inputs. That bring us a lot of convenience, because we can use back-propagation trough time to update the optimizee parameters ϕ directly.

The details of how the optimal θ^* is generated is in the update step 3. Here the g_t is the overall update in the current time-step for parameter θ. m, which is an optimizer, could be think of as an policy in the reinforcement learning. Nevertheless, since we use the gradient of θ as the RNN input, the update rule m is differentiable. That is essentially how it differs from the neural architecture search in [4].

Figure 2 shows the computation graph unrolled by 3 time-steps. In practice, the authors add some modification to this model.

First, they add weights to each time-steps as the equation 4 shows.

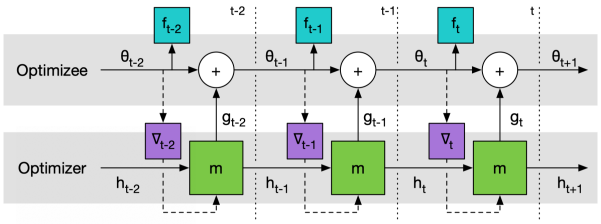


Figure 2: Computational graph

$$\mathcal{L}(\phi) = E[\sum_{t=1}^{T} w_t f(\theta_t)] \tag{4}$$

Analogous to reinforcement learning, the w_t here could be think of as the conditioned probability of action at time t taking place given. And the expectation of the reward at each time step sum up to form the loss function. However, in function 4, there are two difference. On one hand, the w_t here is not probability but a weight, which could be specif ed in configuration. On the other hand, the loss is minimizing the expectation of in total all T time steps accumulated. And the ∇_t at each time step is not conditioned on previous one in a direct way.

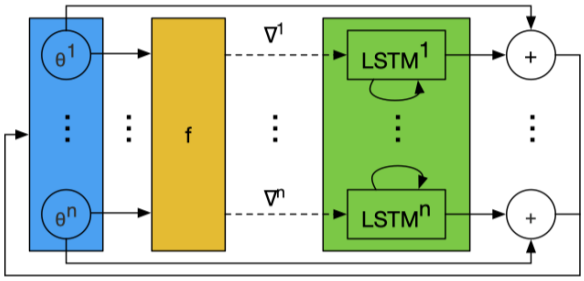
And here comes the second modification 5. The second derivatives are ignored in the computation graph. In Figure 2, arrows with dash lines represent second derivatives that won't be taken into account. Since those second derivatives are intractable, so they forsake them for this purpose.

$$\partial \nabla_t / \partial \phi = 0 \tag{5}$$

3 Coordinate-wise LSTM

In some cases where the optimizee has tens of thousands of parameters, there is a problem that the optimizer parameters ϕ scale with the optimizee parameters θ. Thus the optimizer is huge and hard to train. To keep the network size small, the authors use coordinate-wise neural network as shown in Figure 3. In a single time step, each θ is a training sample that feed into the shame LSTM. So ϕ is shared across all θ and each θ has individual hidden states. This architecture focus on only one coordinate when performing updates. Since the input dimension of LSTM is therefore one dimensional, the amount of optimizer parameters ϕ is substantially reduced.

In addition, they use LSTM instead of RNN to avoid potential vanishing gradient problems. The long-term information in this training process can be integrate in to the model as well.



4 Preprocessing and postprocessing

Another problem that comes into view is that the optimizee parameters θ has different magnitudes. For example, in neural networks, gradients of parameters from different layers and can diversely differ from each other. This makes the training of optimizer difficult, since neural networks only works well when the inputs and outputs are not extremely large or small. Therefore, preprocessing and postprocessing are necessary in some cases.

To this aim, the authors come up with two preproceesing strategies. The first one is simply rescale the input or output by an suitable constant. This method is proved sufficiently successful in the experiments. The second strategies is more complicated, but just slightly improves the results compared to regaling. By using logarithm, the huge difference between numbers of diverse magnitude is substantially reduced. For example, 10 and 10000 will be reduced to $\log 10 = 1$ and $\log 10000 = 4$. But there is another problem that, when the absolute value of gradient $|\nabla_t|$ is approaching 0, the logarithm of it comes to $-\infty$, i.e. diverge. To prevent this, they introduce p to control how small gradients are ignored. Finally, the preprocess formula 6 using absolute values and considering the signs.

$$\nabla^k \to \begin{cases} (\frac{\log(|\nabla|)}{p}, \operatorname{sgn}(\nabla)) & \text{if } |\nabla| \geq e^{-p} \\ (-1, e^p \nabla) & \text{otherwise} \end{cases} \tag{6}$$

5 Experiments

The authors design experiments to compare the LSTM optimizer with the state-of-the-art hand-crafted optimizers and test the robustness to different architecture as well.

5.1 Quadratic functions

This experiment shows how well the LSTM optimizer generalize to quadratic functions of the same distribution. They first sample a function f from this function family 7, and then train a LSTM optimizer on it for 100 steps. The

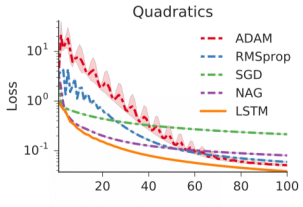


Figure 3: Comparison between learned and hand-crafted optimizers

optimizer parameters ϕ are updated every 20 steps. After training, they sampled n other functions from the same distribution and use the already trained optimizer to optimize them, and compare the loss over time with hand-crafted optimizers. From figure 3 we can tell that the LSTM optimizer outperform all hand-crafted optimizer in this experiment.

$$f(\theta) = ||W\theta - y||_2^2 \tag{7}$$

5.2 Neural Network

In this experiment, the authors want to not only compare the performance of LSTM optimizer with hand-crafted ones, but also test how well it generalize when the neural network architecture changed.

They first train the LSTM optimizer on a base model with 20 hidden units, 1 hidden layer and sigmoid as activation function. The task of base model is to classify numbers in the MNIST dataset. Figure 4 shows that the LSTM converges faster and also outperform all hand-crafted optimizers as expected. However, after it reaches the plateau, there are noticeable oscillations in the loss function.

In the next step, they use the pre-trained optimizer on the base model and test it on 3 modified models: one with 40 hidden units instead of 20; one with 2 hidden layers; one uses ReLU as activation function. Likewise, they also trained hand-crafted models as comparison. The results are in figure 5

In the first and second plot, the LSTM optimizer works well as expected and outperform all hand-crafted optimizers. However, in the third plot, where we change the activation function to ReLU, the LSTM optimizer fails to converge. We could say that the LSTM optimizer can not generalize to this case. Some possible reason of this could be the different dynamics of sigmoid and ReLU as activation functions. Because the shape of sigmoid is staircase-like while the shape of ReLU is totally different. We could speculate that the LSTM optimizer might generalize to activation functions like tanh, which has similar shape and dynamics with sigmoid.

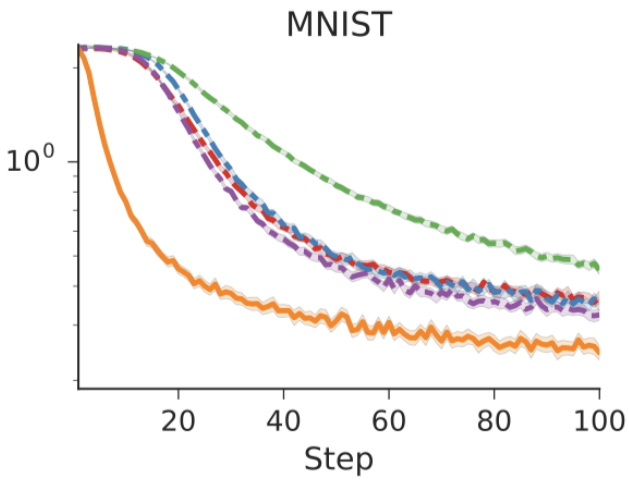


Figure 4: Comparison between learned and hand-crafted optimizers

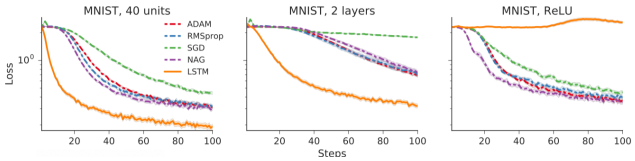


Figure 5: Comparison between learned and hand-crafted optimizers.

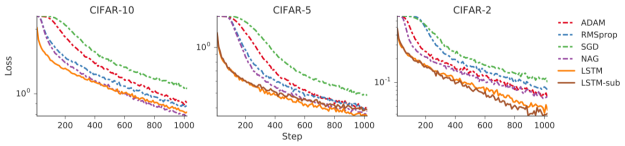


Figure 6: Optimization performance on the CIFAR-10 dataset and subsets.

5.3 Convolutional Neural Network

In this experiment, the authors test the LSTM optimizer on convolutional neural network trained on CIFAR-10. They want to see how the optimizer can transfer to the unseen dataset. So the experiment is designed in this way: At first they train on all 10 classes of pictures from CIFAR-10, and test on a held-out dataset. Then they train on a modified dataset, for example CIFAR-2 and CIFAR-5, in which only 2 or 5 out of 10 classes are included, and test on dataset consisting of samples with unseen labels. The CNN model is with 3 convolutional layers followed by a fully connected layer using ReLU non-linearity.

One thing to notice is that the parameters in convolutional layer and in fully-connected layers have different mechanisms. This makes it difficult if using only one LSTM to capture the update dynamics. Considering the different dynamics of convolutional layers and fully connected layers. The authors use two LSTM in the optimizer for convolutional layers and fully-connected layers each. This modification makes the training less difficult.

As the results in figure 6 shows, both LSTM and LSTM-sub optimizers outperform all hand-crafted optimizers.

5.4 Neural Art

The last experiment is conducted on Neural Art [1] project. Neural art project is aiming at transfer artistic style to pictures using convolutional neural network. This forms the test-bed for the LSTM optimizer since the generalization can be tested via changing art styles. The target function 7 consists of the loss from content image c, style image s and a regularizer which adds smoothness to the resulting picture 7.

$$f(\theta) = \alpha\mathcal{L}_{content}(c, \theta) + \beta\mathcal{L}_{style}(s, \theta) + \gamma\mathcal{L}_{reg}(\theta) \tag{8}$$

In the training progress, the authors trained the LSTM optimizer on 1 style image and 1800 content images from the ImageNet dataset for 128 steps. The parameter θ is updated every 20 steps. Next they validate the optimizee using 20 content images and test with 100 content images.

Move on to the test model, they want to test how well the optimizer generalize to different artistic styles and different resolutions. As can be seen in Figure 8, the LSTM optimizer still does a good job.

Figure 7: Examples of images styled using the LSTM optimizer

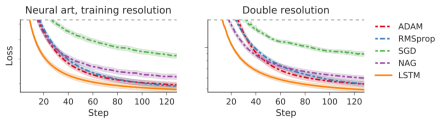


Figure 8: Examples of images styled using the LSTM optimizer

6 Conclusions

As a conclusion, the LSTM optimizer achieves comparable results to hand-crafted optimizer. Compared to unsupervised method using reinforcement learning, this method is much more interpretable and tractable. However, they share something in common in higher level. But if you compare it with hand-crafted optimizers, the strengths are obvious. One strength of this method is that it is fully automatic, which means no human efforts are needed to tune the hyper-parameters. All the optimizee parameters are learned in the LSTM. By using LSTM, the gradient history is integrated in the general update of parameter. This has been proved to have significant effect in convex optimization, similar to momentum. Another favorable thing is that it is applicable to many classes of problems.

Nevertheless, the LSTM optimizer still have some weakness to be improved. As mentioned before, it fails to generalize when using ReLU as activation function in neural networks. This shows the lack of robustness when modifying the neural network architecture. Possible explanations for this are yet to be discover. Second, in backpropagation through time, all second derivatives are ignored so that the computation is not intractable. However, there might be valuable information in the second derivatives if the model architecture is larger and much complex. Since they disregard the second derivatives, some inter-parameter information are not being modeled. Last but not least, the large computation overhead can not been overseen.

7 Outlook

Some other papers extend this model and come up with some improvements.

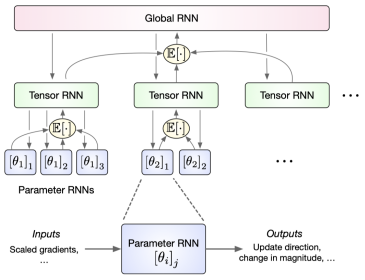


Figure 9: Hierarchical RNN architecture

7.1 Hierarchical RNN

In this paper [3], they introduced a hierarchical RNN to add structural dependencies between parameters. In figure 9, each parameter has its individual RNN. The tensor RNNs govern all the parameter RNN belonging to the same tensor. Likewise, the global RNN controls all tensor RNN. The inputs of upper layers are the expectation of outputs from the lower layer. And the loss of upper layer are regarded as bias of the lower layer. This architecture helps to capture inter-parameter dependencies with low computation overhead. Also, it scale well to problems with larger architecture.

7.2 Unroll Optimization

Recall from the equation 4, one interesting aspect of it is to find optimal unrolled steps. This helps to update parameter θ on partial trajectory. Setting the total unrolled steps T is a trade-off of how much gradient history ought to be integrated. Therefore this paper [2] is focused on finding the optimal unrolled steps to perform truncated backpropagation. The truncated backpropagation is controlled by window size. By searching for the optimal window size, the potential exponential explosion of gradients could be avoided and also introduce bias to the model.

References

[1] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. "A neural algorithm of artistic style". In: *arXiv preprint arXiv:1508.06576* (2015).

[2] Luke Metz et al. "Understanding and correcting pathologies in the training of learned optimizers". In: *International Conference on Machine Learning*. 2019, pp. 4556–4565.

[3] Olga Wichrowska et al. "Learned optimizers that scale and generalize". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 3751–3760.

[4] Barret Zoph and Quoc V Le. “Neural architecture search with reinforcement learning”. In: *arXiv preprint arXiv:1611.01578* (2016).