Neural Optimizers with Hypergradients for Tuning Parameter-Wise Learning Rates

Jie Fu* JIE.FU@U.NUS.EDU
Ritchie Ng* RITCHIENG@U.NUS.EDU
Danlu Chen DLCHEN13@FUDAN.EDU.CN
Ilija Ilievski ILIJA.ILIEVSKI@U.NUS.EDU
Christopher Pal CHRISTOPHER.PAL@POLYMTL.CA
Tat-Seng Chua CHUATS@COMP.NUS.EDU.SG
National University of Singapore, Fudan University, Polytechnique Montréal

Abstract

Recent studies show that LSTM-based neural optimizers are competitive with state-of-the-art hand-designed optimization methods for short horizons. Existing neural optimizers learn how to update the optimizee parameters, namely, predicting the product of learning rates and gradients directly and we suspect it is the reason why the training task becomes unnecessarily difficult. Instead, we train a neural optimizer to only control the learning rates of another optimizer using gradients of the training loss with respect to the learning rates. Furthermore, with the assumption that learning rates tend to remain unchanged over a certain number of iterations, the neural optimizer is only allowed to propose learning rates every S iterations where the learning rates are fixed during these S iterations and this enables it to generalize to longer horizons. The optimizee is trained by Adam on MNIST, and our neural optimizer learns to tune the learning rates for the Adam. After 5 meta-iterations, another optimizee trained by Adam whose learning rates are tuned by the learned but frozen neural optimizer, outperforms those trained by existing hand-designed and learned neural optimizers in terms of convergence rate and final accuracy for long horizons across several datasets.

Keywords: Hyperparameter optimization, Learning rates, Recurrent neural networks

1. Introduction

Deep neural networks (DNNs) are extremely sensitive to their hyperparameter settings, especially the learning rates (Schaul et al., 2012). Therefore, choosing good learning rates is a crucial step in achieving a desirable performance. Bayesian optimization (Brochu et al., 2010) has been shown to reach or outperform expert-set hyperparameters on a variety of benchmark datasets. But such optimization methods can only find one fixed learning rate for all iterations, while changing the learning rates during training usually gives better performance (Maclaurin et al., 2015). On the other hand, several sophisticated hand-designed adaptive learning rate optimizers have been proposed (Kingma and Ba, 2014). However, these hand-designed optimizers are usually developed to exploit structures in

* Equal contribution

a particular domain and thus can hardly provide good generalization performance across problems (Andrychowicz et al., 2016).

Recently, there has been a rising interest in *learning to learn*, i.e. learning to update an optimizee's parameters directly. (Andrychowicz et al., 2016) demonstrated how to train an LSTM to optimize another DNN more effectively than hand-designed optimizers. But it is not effective for long horizons. (Lv et al., 2017) introduced the random scaling trick to enable the neural optimizer to generalize to longer horizons, but the performance is still worse than Adam. (Wichrowska et al., 2017) proposed to use meta training samples and hierarchical LSTMs to improve the generalization ability. However, it is unclear if this approach can outperform carefully tuned hand-designed optimizers for long horizons in terms of test loss and is not efficient when testing due to its LSTM overhead at every iteration.

We suspect that incorporating domain knowledge rather than learning everything from data would ease the learning task and improve the performance in terms of efficacy and efficiency. As a result, in this work, we extend this line of research by combining hand-designed and learned optimizers in which an LSTM-based optimizer learns to propose parameter-wise learning rates for the hand-designed optimizer and is trained with hypergradients (the gradients with respect to learning rates). In fact, our method can be used to tune any continuous dynamic hyperparameters (including momentum (Maclaurin et al., 2015)), but we focus on learning rates as a case study.

2. Learning learning rates

2.1. Setup

Following the notation in (Loshchilov and Hutter, 2015), training a deep model with n parameters can be formulated as the problem of minimizing a function $l : \mathbb{R}^n \to \mathbb{R}$. We define a loss function $\psi : \mathbb{R}^n \to \mathbb{R}$ for each training sample; the distribution of training samples then induces a distribution $\mathcal{D}$ over the loss function ψ, and then the function l is the expectation of this distribution $l(x; w) := \mathbb{E}_{\psi \sim \mathcal{D}}[\psi(x; w)]$. Usually, l is optimized by iteratively adjusting w_t (the weight vector at time step t) using gradient information obtained on a mini-batch $\{\psi_{i=1}^{b}\} \sim \mathcal{D}^b$. Based on this mini-batch, the gradient $\nabla l(x; w_t)$ is computed with $\nabla l(x; w_t) = \frac{1}{b} \sum_{i=1}^{b} \psi_i(x; w_t)$. Then the weight vector is updated using stochastic gradient descent (SGD) as $w_{t+1} = w_t - \alpha_t \nabla l(x; w_t)$, where α_t is the SGD learning rate at time t.

In (Andrychowicz et al., 2016), an LSTM optimizer g with its own set of parameters ϕ, is used to minimize the loss of optimizee l in the form of

$$w_{t+1} = w_t - g(\nabla l(x; w_t); \phi), \tag{1}$$

where $g(\cdot)$ is the optimizer. The final *optimizee* parameters $w_T(\phi, l)$ after T iterations is a function of the *optimizer* parameters ϕ. Given a distribution of function l, the expected loss is $\mathcal{L}(\phi) = \mathbb{E}_l[l(x; w^T(l, \phi))]$. While this objective function only depends on the final parameter value, for training the optimizer we can have an objective that depends on the entire training trajectory:

$$\mathcal{L}(\phi) = \mathbb{E}_l[\sum_{t=1}^{T} \beta_t l(x; w_t)], \tag{2}$$

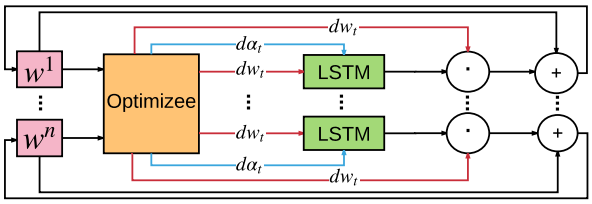

Figure 1: One step of the LSTM optimizer. All LSTMs share the same parameters, but have separate hidden states. The red lines indicate gradients, and the blue lines indicate hypergradients. The diagram is modified from (Andrychowicz et al., 2016).

where β_t is the importance weight for the loss at time t.

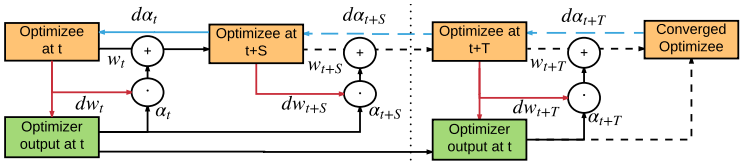

Figure 2: Computational graph for chaining the hypergradients, $d\alpha$, from the optimizee.

2.2. Outputting learning rates

The loss $\mathcal{L}(\phi)$ defined in Eq. 2 is only equivalent to the original problem (i.e. optimizing the final loss $l(w_{t=T})$) when we set $\beta_T = 1$, in which case the back-propagation through time becomes inefficient, and thus (Andrychowicz et al., 2016) proposes to set all $\beta_t = 1$. However, the loss $\mathcal{L}(\phi)$ is not the *original* teaching signal for the neural optimizer anymore, and the optimizer might need many meta-iterations[1] to converge to a good solution. In fact, (Lv et al., 2017) finds that the approach needs thousands of meta-iterations to converge on MNIST. Furthermore, since the neural optimizer learns how to update the optimizee parameters directly, it needs to store all the optimizee parameters over all iterations.

To solve the above problems, our proposed one is trained to predict the learning rates. That is, in contrast to Eq. 1, we adopt the following rule: $w_{t+1} = w_t - g_t(h(\nabla l(w_t)); \phi) \cdot \nabla l(w_t)$, where $h(\cdot)$, the input to the LSTM optimizer, is defined as the state description vector of the optimizee gradients at iteration t, and $g_t(h(\nabla l(w_t)); \phi) = \alpha_t$.

Following (Andrychowicz et al., 2016), we let all the LSTMs share the same parameters but have separate hidden states as shown in Fig. 1, and use the following pre-processing approaches: $h^k(\cdot) = (\frac{\log(\nabla^k l(w_t))}{c}, \text{sgn}(\nabla^k l(w_t))$ if $|\nabla^k l(w_t)| \geq e^{-c}$, and $h^k(\cdot) = (-1, e^c \nabla^k l(w_t))$ otherwise, where $c > 0$ is a constant, $\nabla^k l(w_t)$ is the gradient for the k-th parameter, and $\text{sgn}(\cdot)$ is the sign function.

1. One meta-iteration is an entire training run for optimizing the optimizee till convergence.

2.3. Training the optimizer

SGD or its variants, whose learning rates are controlled by the LSTM optimizer, is used to train the optimizee till convergence. However, the LSTM optimizer is unable to produce a learning rate for the optimizee at every iteration. This is a deeply rooted problem in the LSTM itself: the LSTM and all the other RNN variants are limited to a few hundred time-steps because the long-term memory contents are diluted at every time-step (Chung et al., 2016). In other words, even though we can collect the learning rate gradients at every optimizee iteration, we cannot use them to effectively train an LSTM by unrolling more than a few hundred time-steps.

In this paper we do not attempt to solve this problem. Instead, we conjecture that learning rates tend not to change dramatically across iterations. Thus as a trade-off, we only allow the LSTM optimizer to propose learning rates every S time-steps, which is a *lazy* approach. Essentially, we use a straight line to approximate the optimal learning rate curve within S steps in the hope that the actual curve is not highly bumpy. The computational graph is shown in Fig. 2.

As a by-product, proposing learning rates every S iterations also makes it efficient for training and testing. When training, it can save storage dramatically by inputting the averaged gradients during those S iterations into the LSTM; When testing, it can reduce the overhead introduced by LSTMs without increasing the mini-batch size like (Wichrowska et al., 2017). Furthermore, this lazy approach makes it less prone for overfitting.

3. Experiments

We evaluate our method on MNIST (LeCun et al., 1998), SVHN (Netzer et al., 2011) and CIFAR-10 (Krizhevsky and Hinton, 2009). A 2-hidden-layer (20 neurons at each layer) MLP with sigmoid activation functions is trained using Adam. The batch-size is 120. Our LSTM optimizer controls the learning rates of Adam[2]. We use a fixed set of random initial parameters for every meta-iteration for all datasets. The learning rates for baseline optimizers are grid-searched from 10^{-1} to 10^{-10}. The LSTM optimizer itself is not hyperparameter-free. But for such a low-dimensional hyperparameter space, we use a very coarse grid search. The LSTM optimizer has 3 layers, each having 20 hidden units, which is trained by Adam with a fixed learning rate of 10^{-7}. The LSTM is trained for 5 meta-iterations and unrolled for 50 steps.

3.1. MNIST

Fig. 3 (left) shows the learning curves and the learning rate schedules proposed by the frozen LSTM on MNIST. We can see that our LSTM optimizer (NOH) improves the optimizee performance significantly in terms of convergence rate and final accuracy after training for 5 meta-iterations. It is quite striking that Adam with learned parameter-wise learning rates can converge almost as fast as the neural optimizer (DMopt) proposed in (Andrychowicz et al., 2016). In contrast, DMopt is incapable of learning to reduce the optimizee's final

2. Although (Wilson et al., 2017) argues that the solutions found by SGD generalize better than adaptive counterparts, they still suggest tuning learning rates with decay. Our method is generic in that it can also be used to tune global learning rates and decay schemes for SGD with averaging.

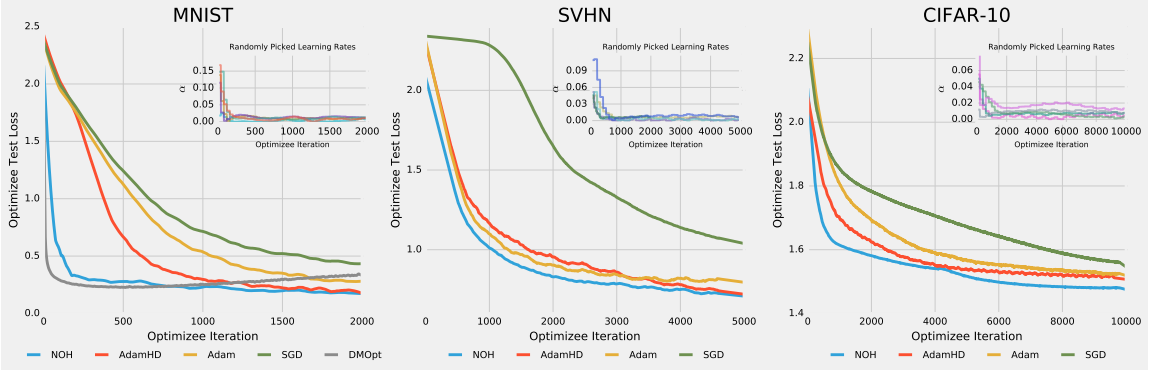

Figure 3: Learning curves of the optimizee and randomly picked parameter-wise learning rate schedules for Adam on different datasets. NOH indicates our proposed method, AdamHD indicates Adam with hypergradient descent (Baydin et al., 2017), and DMopt indicates the method used in (Andrychowicz et al., 2016).

loss within the first 150 meta-iterations and needs about 2000 meta-iterations to converge on MNIST. After training, we have to cherry-pick the best performing models based on validation loss for final testing. Due to the difficulty of training DMopt, we did not run DMopt on SVHN or CIFAR-10. More importantly, it looses the ability to properly train the optimizee after about 400 iterations as shown in (Lv et al., 2017).

The learning rate curves are locally jagged because the LSTM optimizer only changes learning rates every S iterations. Those curves are diverse, but in general share the same pattern: start with large values[3] and then decay. This is different from the observations in (Baydin et al., 2017), where the increase in learning rates for Adam is almost negligible. One possible reason for this might be that the meta learning rates for the hypergradients used in (Baydin et al., 2017) are too small. However, when we increase the meta learning rates, the performance of the optimizee degrades catastrophically, which is mentioned in (Baydin et al., 2017). This might suggest that an adaptive meta learning rate mechanism is needed here, and our LSTM optimizer can be seen as such one which also has reasonable generalization abilities.

3.2. Generalization from MNIST to SVHN and CIFAR-10

We evaluate the generalization ability of our proposed method in this section. We argue that the strength of a learned optimizer is that it is highly problem specific and thus an overly general-purpose algorithm may not be able to exploit the extra structure in a specific problem. As a trade-off, we do not aim to design a neural optimizer that can generalize across different activation functions, but only consider transferring across different datasets, which is still quite useful in practice. For example, we can train an optimizer on a small dataset and then freeze and apply it to another larger dataset, thus saving the overall hyperparameter tuning time.

3. Actually, the learning rates might start increasing from an initial relatively small value, but since our neural optimizer only changes learning rates every S iterations, we cannot determine if it is the case.

We first use Adam to train one MLP for 5 meta-iterations (each having 2K iterations) on MNIST, in which our proposed neural optimizer learns to tune the learning rates of Adam. Then we freeze the learned optimizer and let it propose learning rates for Adam which is in turn used to train another MLP on SVHN and CIFAR-10 for longer horizons. Figure 3 (middle and right) shows that after learning on a smaller and different dataset, the neural optimizer is able to generalize to other datasets. The different scales of randomly picked learning rates between different datasets imply that the optimizer can generalize and behaves accordingly on different problems. We can also observe that the learning rates for SVHN, and especially CIFAR-10, decay more slowly than those on MNIST.

4. Conclusion

We presented an LSTM-based neural optimizer for learning parameter-wise learning rates using hypergradients directly from the optimizee, which is efficient, effective, and plug-and-play. We showed that in order to help the optimizee to achieve a good convergence rate and final accuracy, it might not be necessary for the neural optimizer to learn how to update the optimizee parameters directly. Actually, it is more efficient and effective to learn proper learning rates for existing hand-designed optimizers. This simple training method can achieve reasonable generalization abilities in that the learned optimizers are transferable across different datasets without using meta-training datasets (Wichrowska et al., 2017) or random scaling (Lv et al., 2017), though those engineering innovations are orthogonal to our method and could further improve ours. We also observed that on small-scale problems, Adam can reach a significantly better performance when the behavior of the learning rates is that of an increase preceding a decay.

5. Future work

In (Loshchilov and Hutter, 2016), the authors have shown that warm-restarting the SGD training periodically is an effective overall scheme. For example, if we restart the training by using aggressive fixed learning rates for a while every S epoch, neural optimizers can be used to tune the learning rates and decay within that cycle without unrolling the LSTM too many steps.

It has also been shown that learned or adaptive optimizers (Baydin et al., 2017; Andrychowicz et al., 2016) (including ours) on small-scale problems usually prefer the behavior of changes to the parameters of decay following an initial increase. But studies based on SGD (He et al., 2016) suggest warm-up, namely a strategy of using less aggressive learning rates at the start of training, for large-scale ones. Since our method has shown impressive results on small-scale problems, we look forward to future work on extending it to large-scale ones.

Acknowledgments

This work is supported by NUS-Tsinghua Extreme Search (NExT) project through the National Research Foundation, Singapore. We would like to thank Amazon for AWS Cloud Credits for Research program.

References

Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. *Neural Information Processing Systems*, 2016.

Atilim Gunes Baydin, Robert Cornish, David Martinez Rubio, Mark Schmidt, and Frank Wood. Online learning rate adaptation with hypergradient descent. *arXiv preprint arXiv:1703.04782*, 2017.

Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

Junyoung Chung, Sungjin Ahn, and Yoshua Bengio. Hierarchical multiscale recurrent neural networks. *arXiv preprint arXiv:1609.01704*, 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

Ilya Loshchilov and Frank Hutter. Online batch selection for faster training of neural networks. *arXiv preprint arXiv:1511.06343*, 2015.

Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with restarts. *arXiv preprint arXiv:1608.03983*, 2016.

Kaifeng Lv, Shunhua Jiang, and Jian Li. Learning gradient descent: Better generalization and longer horizons. *arXiv preprint arXiv:1703.03633*, 2017.

Dougal Maclaurin, David Duvenaud, and Ryan P Adams. Gradient-based hyperparameter optimization through reversible learning. *International Conference on Machine Learning*, 2015.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, page 5, 2011.

Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rates. *arXiv preprint arXiv:1206.1106*, 2012.

Olga Wichrowska, Niru Maheswaranathan, Matthew W Hoffman, Sergio Gomez Colmenarejo, Misha Denil, Nando de Freitas, and Jascha Sohl-Dickstein. Learned optimizers that scale and generalize. *arXiv preprint arXiv:1703.04813*, 2017.

Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nathan Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. *arXiv preprint arXiv:1705.08292*, 2017.