Neural Optimizer Search with Reinforcement Learning

Irwan Bello [*1] **Barret Zoph** [*1] **Vijay Vasudevan** [1] **Quoc V. Le** [1]

Abstract

We present an approach to automate the process of discovering optimization methods, with a focus on deep learning architectures. We train a Recurrent Neural Network controller to generate a string in a specific domain language that describes a mathematical update equation based on a list of primitive functions, such as the gradient, running average of the gradient, etc. The controller is trained with Reinforcement Learning to maximize the performance of a model after a few epochs. On CIFAR-10, our method discovers several update rules that are better than many commonly used optimizers, such as Adam, RMSProp, or SGD with and without Momentum on a ConvNet model. These optimizers can also be transferred to perform well on different neural network architectures, including Google's neural machine translation system.

1. Introduction

The choice of the right optimization method plays a major role in the success of training deep learning models. Although Stochastic Gradient Descent (SGD) often works well out of the box, more advanced optimization methods such as Adam (Kingma & Ba, 2015) or Adagrad (Duchi et al., 2011) can be faster, especially for training very deep networks. Designing optimization methods for deep learning, however, is very challenging due to the non-convex nature of the optimization problems.

In this paper, we consider an approach to automate the process of designing update rules for optimization methods, especially for deep learning architectures. The key insight is to use a controller in the form of a recurrent network to generate an update equation for the optimizer. The recurrent network controller is trained with reinforcement learning to maximize the accuracy of a particular model architecture, being trained for a fixed number of epochs by the update rule, on a held-out validation set. This process is illustrated in Figure 1.

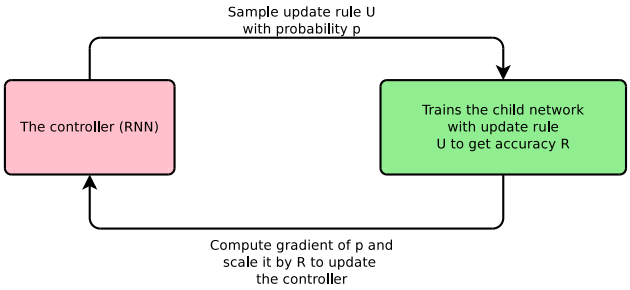

Figure 1. An overview of Neural Optimizer Search.

On CIFAR-10, our approach discovers several update rules that are better than many commonly used optimizers such as Adam, RMSProp, or SGD with and without Momentum on a small ConvNet model. Many of the generated update equations can be easily transferred to new architectures and datasets. For instance, update rules found on a small ConvNet architecture, when applied to the Wide ResNet architecture (Zagoruyko & Komodakis, 2016), improved accuracy over Adam, RMSProp, Momentum, and SGD by a margin up to 2% on the test set. The same update rules also work well for Google's Neural Machine Translation system (Wu et al., 2016) giving an improvement of up to 0.7 BLEU on the WMT 2014 English to German task.

2. Related Work

Neural networks are difficult and slow to train, and many methods have been designed to tackle this difficulty (e.g., Riedmiller & Braun (1992); LeCun et al. (1998); Schraudolph (2002); Martens (2010); Le et al. (2011); Duchi et al. (2011); Zeiler (2012); Martens & Sutskever (2012); Schaul et al. (2013); Pascanu & Bengio (2013); Pascanu et al. (2013); Kingma & Ba (2014); Ba et al. (2017)). More recent optimization methods combine insights from both stochastic and batch methods in that they use a small minibatch, similar to SGD, yet they implement many heuristics to estimate diagonal second-order informa-

[*]Equal contribution [1]Google Brain. Correspondence to: Irwan Bello <ibello@google.com>, Barret Zoph <barretzoph@google.com>, Vijay Vasudevan <vrv@google.com>, Quoc V. Le <qvl@google.com>.

Proceedings of the 34^{th} International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017.

tion, similar to Hessian-free or L-BFGS (Liu & Nocedal, 1989). This combination often yields faster convergence for practical problems (Duchi et al., 2011; Dean et al., 2012; Kingma & Ba, 2014). For example, Adam (Kingma & Ba, 2014), a commonly-used optimizer in deep learning, implements simple heuristics to estimate the mean and variance of the gradient, which are used to generate more stable updates during training.

Many of the above update rules are designed by borrowing ideas from convex analysis, even though optimization problems in neural networks are non-convex. Recent empirical results with non-monotonic learning rate heuristics (Loshchilov & Hutter, 2017) suggest that there are still many unknowns in training neural networks and that many ideas in non-convex optimization can be used to improve it.

The goal of our work is to search for better update rules for neural networks in the space of well known primitives. In other words, instead of hand-designing new update rules from scratch, we use a machine learning algorithm to search the update rules. This goal is shared with recently-proposed methods by Andrychowicz et al. (2016); Ravi & Larochelle (2017); Wichrowska et al. (2017), which employ an LSTM to generate numerical updates for training neural networks. The key difference is that our approach generates a mathematical equation for the update instead of numerical updates. The main advantage of generating an equation is that it can easily be transferred to larger tasks and does not require training any additional neural networks for a new optimization problem. Finally, although our method does not aim to optimize the memory usage of update rules, our method discovers update rules that are on par with Adam or RMSProp while requiring less memory.

The concept of using a Recurrent Neural Network for meta-learning has been attempted in the past, either via genetic programming or gradient descent (Schmidhuber, 1992; Hochreiter et al., 2001). Similar to the above recent methods, these approaches only generate the updates, but not the update equations, as proposed in this paper.

A related approach is using genetic programming to evolve update equations for neural networks (e.g., Bengio et al. (1994); Runarsson & Jonsson (2000); Orchard & Wang (2016)). Genetic programming however is often slow and requires many heuristics to work well. For that reason, many prior studies in this area have only experimented with very small-scale neural networks. For example, the neural networks used for experiments in Orchard & Wang (2016) have around 100 weights, which is quite small compared to today's standards.

Our approach is reminiscent of recent work in automated model discovery with Reinforcement Learning (Baker et al., 2016), especially Neural Architecture Search (Zoph & Le, 2017), in which a recurrent network is used to generate the configuration string of neural architectures instead. In addition to applying the key ideas to different applications, this work presents a novel scheme to combine primitive inputs in a much more flexible manner, which makes the search for novel optimizers possible.

Finally, our work is also inspired by the recent studies by Keskar et al. (2016); Zhang et al. (2017), in which it was found that SGD can act as a regularizer that helps generalization. In our work, we use the accuracy on the validation set as the reward signal, thereby implicitly searching for optimizers that can help generalization as well.

3. Method

3.1. A simple domain specific language for update rules

In our framework, the controller generates strings corresponding to update rules, which are then applied to a neural network to estimate the update rule's performance; this performance is then used to update the controller so that the controller can generate improved update rules over time.

To map strings sampled by the controller to an update rule, we design a domain specific language that relies on a parenthesis-free notation (in contrast to the classic infix notation). Our choice of domain specific language (DSL) is motivated by the observation that the computational graph of most common optimizers can be represented as a simple binary expression tree, assuming input primitives such as the gradient or the running average of the gradient and basic unary and binary functions.

We therefore express each update rule with a string describing 1) the first operand to select, 2) the second operand to select, 3) the unary function to apply on the first operand, 4) the unary function to apply on the second operand and 5) the binary function to apply to combine the outputs of the unary functions. The output of the binary function is then either temporarily stored in our operand bank (so that it can be selected as an operand in subsequent parts of the string) or used as the final weight update as follows:

$$\Delta w = \lambda * b(u_1(op_1), u_2(op_2))$$

where op_1, op_2, $u_1(.)$, $u_2(.)$ and $b(.,.)$ are the operands, the unary functions and the binary function corresponding to the string, w is the parameter that we wish to optimize and λ is the learning rate.

With a limited number of iterations, our DSL can only represent a subset of all mathematical equations. However we note that it recovers common optimizers within one iteration assuming access to simple primitives. Figure 2 shows

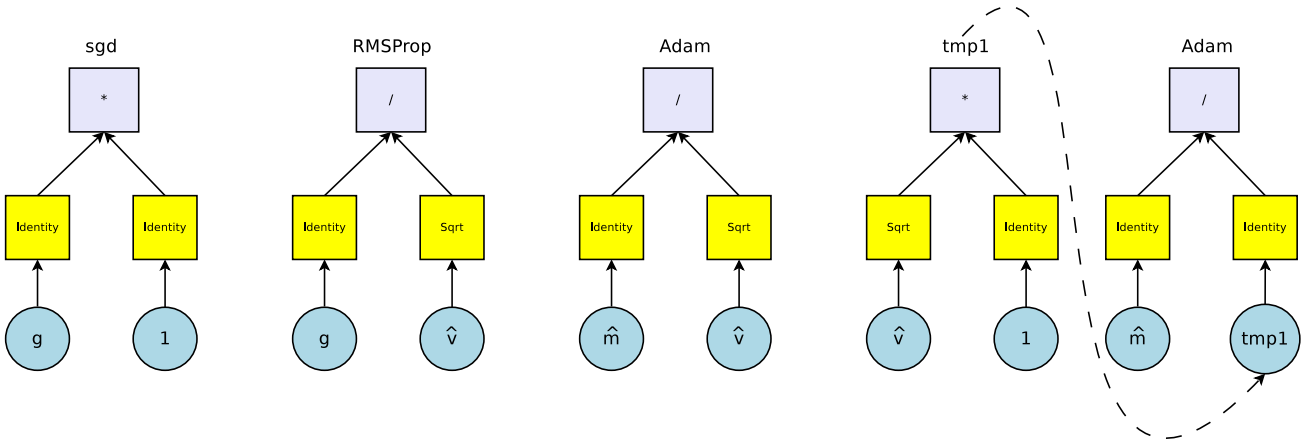

Figure 2. Computation graph of some commonly used optimizers: SGD, RMSProp, Adam. Here, we show the computation of Adam in 1 step and 2 steps. Blue boxes correspond to input primitives or temporary outputs, yellow boxes are unary functions and gray boxes represent binary functions. g is the gradient, $\hat{m}$ is the bias-corrected running estimate of the gradient, and $\hat{v}$ is the bias-corrected running estimate of the squared gradient.

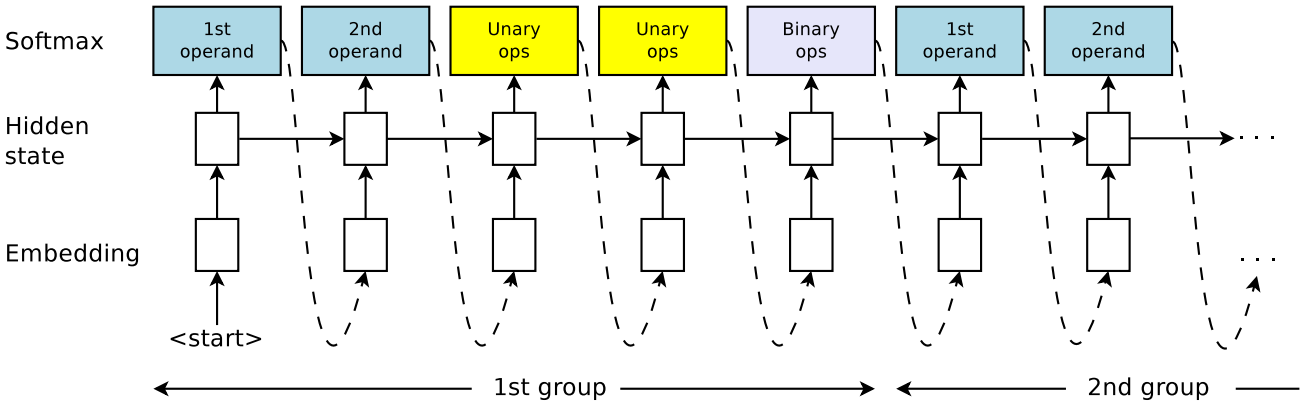

Figure 3. Overview of the controller RNN. The controller iteratively selects subsequences of length 5. It first selects the 1st and 2nd operands op_1 and op_2, then 2 unary functions u_1 and u_2 to apply to the operands and finally a binary function b that combines the outputs of the unary functions. The resulting $b(u_1(op_1), u_2(op_2))$ then becomes an operand that can be selected in the subsequent group of predictions or becomes the update rule. Every prediction is carried out by a softmax classifier and then fed into the next time step as input.

how some commonly used optimizers can be represented in the DSL. We also note that multiple strings in our prediction scheme can map to the same underlying update rule, including strings of different lengths (c.f. the two representations of Adam in Figure 2). This is both a feature of our action space corresponding to mathematical expressions (addition and multiplication are commutative for example) and our choice of domain specific language. We argue that this makes for interesting exploration dynamics because a competitive optimizer may be obtained by expressing a standard optimizer in an expanded fashion and modifying it slightly.

3.2. Controller optimization with policy gradients

Our controller is implemented as a Recurrent Neural Network which samples strings of length $5n$ where n is a number of iterations fixed during training (see Figure 3). Since the operand bank grows as more iterations are computed, we use different softmax weights at every step of prediction.

The controller is trained to maximize the performance of its sampled update rules on a specified model. The training objective is formulated as follows:

$$J(\theta) = \mathbb{E}_{\Delta \sim p_\theta(.)}[R(\Delta)] \tag{1}$$

where $R(\Delta)$ corresponds to the accuracy on a held-out val-

idation set obtained after training a target network with update rule Δ.

Zoph & Le (2017) train their controller using a vanilla policy gradient obtained via REINFORCE (Williams, 1992), which is known to exhibit poor sample efficiency. We find that using the more sample efficient Trust Region Policy Optimization (Schulman et al., 2015) algorithm speeds up convergence of the controller. For the baseline function in TRPO, we use a simple exponential moving average of previous rewards.

3.3. Accelerated Training

To further speed up the training of the controller RNN, we employ a distributed training scheme. In our distributed training scheme the samples generated from the controller RNN are added to a queue, and run on a set of distributed workers that are connected across a network. This scheme is different from (Zoph & Le, 2017) in that now a parameter server and controller replicas are not needed for the controller RNN, which simplifies training. At each iteration, the controller RNN samples a batch of update rules and adds them to the global worker queue. Once the training of the child network is complete, the accuracy on a held-out validation set is computed and returned to the controller RNN. whose parameters get updated with TRPO. New samples are then generated and this same process continues.

Ideally, the reward fed to the controller would be the performance obtained when running a model with the sampled optimizer until convergence. However, such a setup requires significant computation and time. To help deal with these issues, we propose the following trade-offs to greatly reduce computational complexity. First, we find that searching for optimizers with a small two layer convolutional network provides enough of a signal for whether an optimizer would do well on much larger models such as the Wide ResNet model. Second, we train each model for a modest 5 epochs only, which also provides enough signal for whether a proposed optimizer is good enough for our needs. These techniques allow us to run experiments more quickly and efficiently compared to Zoph & Le (2017), with our controller experiments typically converging in less than a day using 100 CPUs, compared to 800 GPUs over several weeks.

4. Experiments

We aim to answer the following questions:

- Can the controller discover update rules that outperform other stochastic optimization methods?
- Do the discovered update rules transfer well to other architectures and tasks?

In this section, we will focus on answering the first question by performing experiments with the optimizer search framework to find optimizers on a small ConvNet model and compete with the existing optimizers. In the next section, we will transfer the found optimizers to other architectures and tasks thereby answering the second question.

4.1. Search space

The operands, unary functions and binary functions that are accessible to our controller are as follows (details below):

- Operands: g, g^2, g^3, $\hat{m}$, $\hat{v}$, $\hat{\gamma}$, $sign(g)$, $sign(\hat{m})$, 1, small constant noise, $10^{-4}w$, $10^{-3}w$, $10^{-2}w$, $10^{-1}w$, ADAM and RMSProp.
- Unary functions which map input x to: x, $-x$, e^x, $\log|x|$, $clip(x, 10^{-5})$, $clip(x, 10^{-4})$, $clip(x, 10^{-3})$, $drop(x, 0.1)$, $drop(x, 0.3)$, $drop(x, 0.5)$ and $sign(x)$.
- Binary functions which map (x, y) to $x+y$ (addition), $x-y$ (subtraction), $x*y$ (multiplication), $\frac{x}{y+\epsilon}$ (division) or x (keep left).

Here, $\hat{m}$, $\hat{v}$, $\hat{\gamma}$ are running exponential moving averages of g, g^2 and g^3, obtained with decay rates β_1, β_2 and β_3 respectively, $drop(.|p)$ sets its inputs to 0 with probability p and $clip(.|l)$ clips its input to $[-l, l]$. All operations are applied element-wise.

In our experiments, we use binary trees with depths of 1, 2 and 3 which correspond to strings of length 5, 10 and 15 respectively. The above list of operands, unary functions and binary function is quite large, so to address this issue, we find it helpful to only work with subsets of the operands and functions presented above. This leads to typical search space sizes ranging from 10^6 to 10^{10} possible update rules.

We also experiment with several constraints when sampling an update rule, such as forcing the left and right operands to be different at each iteration, and not using addition as the final binary function. An additional constraint added is to force the controller to reuse one of the previously computed operands in the subsequent iterations. The constraints are implemented by manually setting the logits corresponding to the forbidden operands or functions to $-\infty$.

4.2. Experimental details

Across all experiments, our controller RNN is trained with the ADAM optimizer with a learning rate of 10^{-5} and a minibatch size of 5. The controller is a single-layer LSTM with hidden state size 150 and weights are initialized uniformly at random between -0.08 and 0.08. We also use an

entropy penalty to aid in exploration. This entropy penalty coefficient is set to 0.0015.

The child network architecture that all sampled optimizers are run on is a small two layer 3x3 ConvNet. This ConvNet has 32 filters with ReLU activations and batch normalization applied after each convolutional layer. These child networks are trained on the CIFAR-10 dataset, one of the most benchmarked datasets in deep learning.

The controller is trained on a CPU and the child models are trained using 100 distributed workers which also run on CPUs (see Section 3.3). Once a worker receives an optimizer to run from the controller, it performs a basic hyperparameter sweep on the learning rate: 10^i with i ranging from -5 to 1, with each learning rate being tried for 1 epoch of 10,000 CIFAR-10 training examples. The best learning rate after 1 epoch is then used to train our child network for 5 epochs and the final validation accuracy is reported as a reward to the controller. The child networks have a batch size of 100 and evaluate the update rule on a fixed held-out validation set of 5,000 examples. In this setup, training a child model with a sampled optimizer generally takes less than 10 minutes. Experiments typically converge within a day. All experiments are carried out using TensorFlow (Abadi et al., 2016).

The hyperparameter values for the update rules are inspired by standard values used in the literature. We set ϵ to 10^{-8}, β_1 to 0.9 and $\beta_2 = \beta_3$ to 0.999. Preliminary experiments indicate that the update rules are robust to slight changes in the hyperparameters they were searched over.

4.3. Experimental results

Our results show that the controller discovers many different updates that perform well during training and the maximum accuracy also increases over time. In Figure 4, we show the learning curve of the controller as more optimizers are sampled.

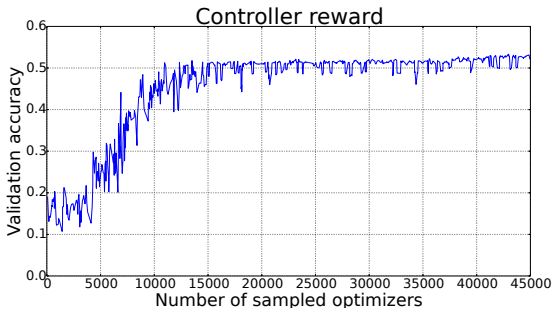

Figure 4. Controller reward increasing over time as more optimizers are sampled.

The plots in Figure 5 show the results of two of our best optimizers being run for 300 epochs on the full CIFAR-10 dataset. From the plots we observe that our optimizers slightly outperform Momentum and SGD, while greatly outperforming RMSProp and Adam.

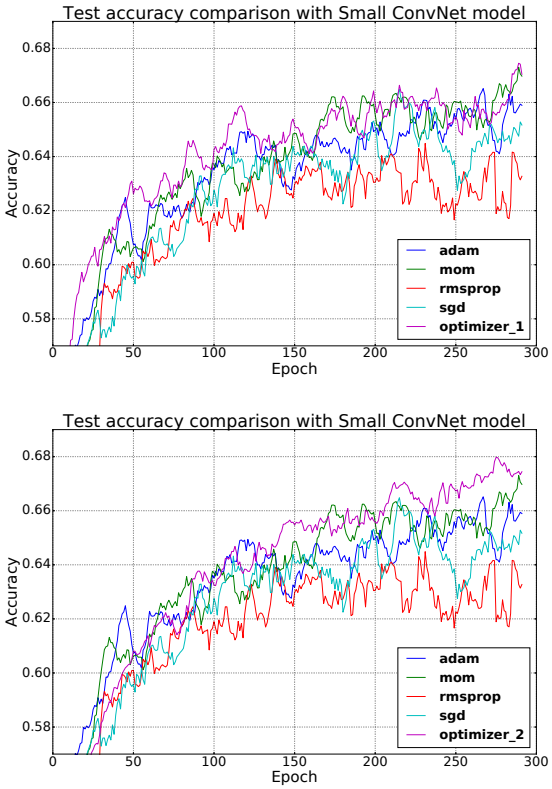

Figure 5. Comparison of two of the best optimizers found with Neural Optimizer Search using a 2-layer ConvNet as the architecture. Optimizer_1 refers to $[e^{\text{sign}(g)*\text{sign}(m)} + \text{clip}(g, 10^{-4})] * g$ and Optimizer_2 refers to $\text{drop}(\hat{m}, 0.3) * e^{10^{-3}*w}$.

The controller discovered update rules that work well, but also produced update equations that are fairly intuitive. For instance, among the top candidates is the following update function:

- $e^{\text{sign}(g)*\text{sign}(m)} * g$

Because $e^{\text{sign}(g)*\text{sign}(m)}$ is positive, in each dimension the update follows the direction of g with some adjustments to the scale. The term $e^{\text{sign}(g)*\text{sign}(m)}$ means that if the signs of the gradient and its running average agree, we should make an update to the coordinate with the scale of e, otherwise make an update with the scale of $1/e$. This expression appears in many optimizers that the model found, showing up in 5 of the top 10 candidate update rules:

- $[e^{\text{sign}(g)*\text{sign}(m)} + \text{clip}(g, 10^{-4})] * g$

- $\text{drop}(g, 0.3) * e^{\text{sign}(g)*\text{sign}(m)}$
- $\text{ADAM} * e^{\text{sign}(g)*\text{sign}(m)}$
- $\text{drop}(g, 0.1) * e^{\text{sign}(g)*\text{sign}(m)}$

5. Transferability experiments

A key advantage of our method of discovering update equations compared to the previous approach (Andrychowicz et al., 2016) is that update equations found by our method can be easily transferred to new tasks. In the following experiments, we will exercise some of the update equations found in the previous experiment on different network architectures and tasks. The controller is not trained again, and the update rules are simply reused.

5.1. Control Experiment with Rosenbrock function

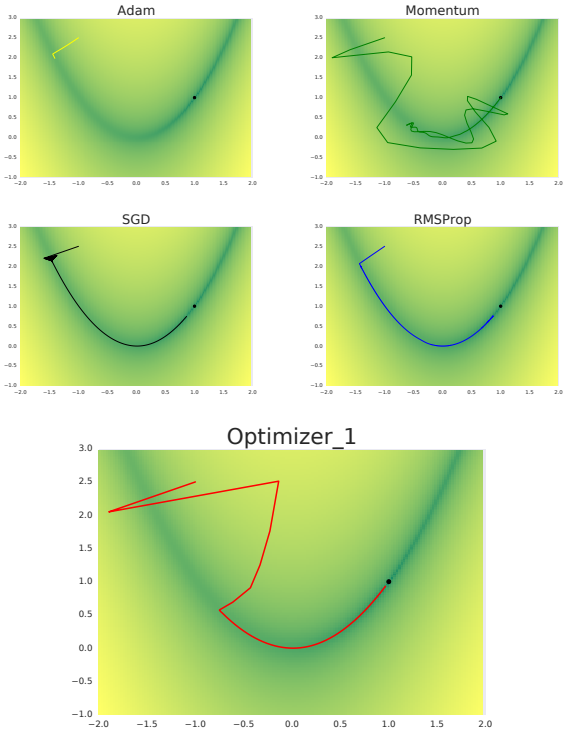

Figure 6. Comparison of an optimizer found with Neural Optimizer Search to the other well-known optimizers on the Rosenbrock function. Optimizer_1 refers to $e^{\text{sign}(g)*\text{sign}(m)} * g$. The black dot is the optimum.

We first test one of the optimizers we found in the previous experiment on the famous Rosenbrock function against the commonly used deep learning optimizers in TensorFlow (Abadi et al., 2016): Adam, SGD, RMSProp and Momentum, and tune the value of ϵ in Adam in a log scale between 10^{-3} and 10^{-8}. In this experiment, each optimizer is run for 4000 iterations with 4 different learning rates on a logarithmic scale and and the best performance is plotted. The update rule we test is the intuitive $g * e^{\text{sign}(g)*\text{sign}(m)}$. The results in Figure 6 show that our optimizer outperforms Adam, RMSProp, SGD, and is close to matching the performance of Momentum on this task.

5.2. CIFAR-10 with Wide ResNet

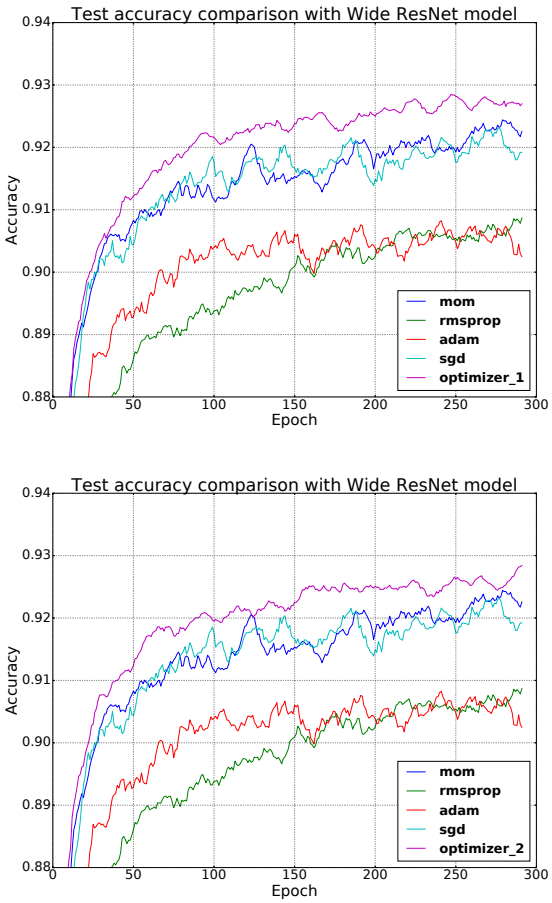

Figure 7. Comparison of two of the best optimizers found with Neural Optimizer Search using Wide ResNet as the architecture. Optimizer_1 refers to $[e^{\text{sign}(g)*\text{sign}(m)} + \text{clip}(g, 10^{-4})] * g$ and Optimizer_2 refers to $\text{drop}(\hat{m}, 0.3) * e^{10^{-3}w}$.

We further investigate the generalizability of the found update rules on a different and much larger model: the Wide ResNet architecture (Zagoruyko & Komodakis, 2016). Our controller finds many optimizers that perform well when run for 5 epochs on the small ConvNet. To filter optimizers that do well when run for many more epochs, we run dozens of our top optimizers for 300 epochs and aggressively stop optimizers that show less promise. The top optimizers identified by this process are also the top optimizers for the small ConvNet and the GNMT experiment.

Optimizer	Final Val	Final Test	Best Val	Best Test
SGD	92.0	91.8	92.9	91.9
Momentum	92.7	92.1	93.1	92.3
ADAM	90.4	90.1	91.8	90.7
RMSProp	90.7	90.3	91.4	90.3
$[e^{\text{sign}(g)*\text{sign}(m)} + \text{clip}(g, 10^{-4})] * g$	92.5	92.4	93.8	93.1
$\text{clip}(\hat{m}, 10^{-4}) * e^{\hat{v}}$	93.5	92.5	93.8	92.7
$\hat{m} * e^{\hat{v}}$	93.1	92.4	93.8	92.6
$g * e^{\text{sign}(g)*\text{sign}(m)}$	93.1	92.8	93.8	92.8
$\text{drop}(g, 0.3) * e^{\text{sign}(g)*\text{sign}(m)}$	92.7	92.2	93.6	92.7
$\hat{m} * e^{g^2}$	93.1	92.5	93.6	92.4
$\text{drop}(\hat{m}, 0.1)/(e^{g^2} + \epsilon)$	92.6	92.4	93.5	93.0
$\text{drop}(g, 0.1) * e^{\text{sign}(g)*\text{sign}(m)}$	92.8	92.4	93.5	92.2
$\text{clip}(\text{RMSProp}, 10^{-5}) + \text{drop}(\hat{m}, 0.3)$	90.8	90.8	91.4	90.9
$\text{ADAM} * e^{\text{sign}(g)*\text{sign}(m)}$	92.6	92.0	93.4	92.0
$\text{ADAM} * e^{\hat{m}}$	92.9	92.8	93.3	92.7
$g + \text{drop}(\hat{m}, 0.3)$	93.4	92.9	93.7	92.9
$\text{drop}(\hat{m}, 0.1) * e^{g^3}$	92.8	92.7	93.7	92.8
$g - \text{clip}(g^2, 10^{-4})$	93.4	92.8	93.7	92.8
$e^g - e^{\hat{m}}$	93.2	92.5	93.5	93.1
$\text{drop}(\hat{m}, 0.3) * e^{10^{-3}w}$	93.2	93.0	93.5	93.2

Table 1. Performance of Neural Search Search and standard optimizers on the Wide-ResNet architecture (Zagoruyko & Komodakis, 2016) on CIFAR-10. Final Val and Final Test refer to the final validation and test accuracy after for training for 300 epochs. Best Val corresponds to the best validation accuracy over the 300 epochs and Best Test is the test accuracy at the epoch where the validation accuracy was the highest. For each optimizer we report the best results out of seven learning rates on a logarithmic scale according to the validation accuracy.

Figure 7 shows the comparison between SGD, Adam, RMSProp, Momentum and two of the top candidate optimizers. The plots reveal that these two optimizers outperform the other optimizers by a size-able margin on a competitive CIFAR-10 model.

Table 1 shows more details of the comparison between our top 16 optimizers against the commonly used SGD, RMSProp, Momentum, and Adam optimizers. We note that although Momentum was used and tuned by Zagoruyko & Komodakis (2016), many of our updates outperform that setup. The margin of improvement is around 1%. Our method is also better than other optimizers, with a margin up to 2%.

5.3. Neural Machine Translation

We run one particularly promising optimizer, $g * e^{\text{sign}(g)*\text{sign}(m)}$, on the WMT 2014 English $\rightarrow$ German task. Our goal is to test the transferability of this optimizer on a completely different model and task, since before our optimizers were run on convolutional networks and the translation models are RNNs. Our optimizer in this experiment is compared against the Adam optimizer (Kingma & Ba, 2015). The architecture of interest is the Google Neural Machine Translation (GNMT) model (Wu et al., 2016), which was shown to achieve competitive translation quality on the English $\rightarrow$ German task. The GNMT network comprises 8 LSTM layers for both its encoder and decoder (Hochreiter & Schmidhuber, 1997), with the first layer of the encoder having bidirectional connections. This GNMT model also employs attention in the form of a 1 layer neural network.

The model is trained in a distributed fashion using a parameter server. Twelve workers are used, with each worker using 8 GPUs and a minibatch size of 128. Further details for this model can be found in Wu et al. (2016).

In our experiments, the only change we make to training is to replace Adam with the new update rule. We note that the GNMT model's hyperparameters, such as weight initialization, were previously tuned to work well with Adam (Wu et al., 2016), so we expect more tuning can further improve the results of this new update rule.

The results in Table 2 show that our optimizer does indeed generalize well and achieves an improvement of 0.1 perplexity, which is considered to be a decent gain on this task. This gain in training perplexity enables the model to obtain

a 0.5 BLEU improvement over the Adam optimizer on the test set Wu et al. (2016). On the validation set, the averaged improvement of 5 points near the peak values is 0.7 BLEU.

Optimizer	Train perplexity	Test BLEU
Adam	1.49	24.5
$g * e^{\text{sign}(g)*\text{sign}(m)}$	1.39	25.0

Table 2. Performance of our optimizer versus ADAM in a strong baseline GNMT model on WMT 2014 English → German.

Finally, the update rule is also more memory efficient as it only keeps one running average per parameter, compared to two running averages for Adam. This has practical implications for much larger translation models where Adam cannot currently be used due to memory constraints (Shazeer et al., 2017).

6. Conclusion

This paper considers an approach for automating the discovery of optimizers with a focus on deep neural network architectures. We evaluate our discovered optimizers on widely used CIFAR-10 and NMT models for which we obtain competitive performance against common optimizers and validate to some extent that the optimizers generalize across architectures and datasets.

One strength of our approach is that it naturally encompasses the environment in which the optimization process happens. One may for example use our method for discovering optimizers that perform well in scenarios where computations are only carried out using 4 bits, or a distributed setup where workers can only communicate a few bits of information to a shared parameter server. Unlike previous approaches in learning to learn, the update rules in the form of equations can be easily transferred to other optimization tasks.

Finally, one of the update rules found by our method, $g * e^{\text{sign}(g)*\text{sign}(m)}$, is surprisingly intuitive and particularly promising. Our experiments show that it performs well on a range of tasks that we have tried, from image classification with ConvNets to machine translation with LSTMs. In addition to opening up new ways to design update rules, this new update rule can now be used to improve the training of deep networks.

Acknowledgements

We thank Samy Bengio, Jeff Dean, Vishy Tirumalashetty and the Google Brain team for the help with the project.

References

Abadi, Martín, Barham, Paul, Chen, Jianmin, Chen, Zhifeng, Davis, Andy, Dean, Jeffrey, Devin, Matthieu, Ghemawat, Sanjay, Irving, Geoffrey, Isard, Michael, Kudlur, Manjunath, Levenberg, Josh, Monga, Rajat, Moore, Sherry, Murray, Derek G., Steiner, Benoit, Tucker, Paul, Vasudevan, Vijay, Warden, Pete, Wicke, Martin, Yu, Yuan, , and Zheng, Xiaoqiang. Tensorflow: A system for large-scale machine learning. *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

Andrychowicz, Marcin, Denil, Misha, Gomez, Sergio, Hoffman, Matthew W., Pfau, David, Schaul, Tom, and de Freitas, Nando. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, pp. 3981–3989, 2016.

Ba, Jimmy, Grosse, Roger, and Martens, James. Distributed second-order optimization using Kronecker-factored approximations. In *International Conference on Learning Representations*, 2017.

Baker, Bowen, Gupta, Otkrist, Naik, Nikhil, and Raskar, Ramesh. Designing neural network architectures using reinforcement learning. *arXiv preprint arXiv:1611.02167*, 2016.

Bengio, Samy, Bengio, Yoshua, and Cloutier, Jocelyn. Use of genetic programming for the search of a new learning rule for neural networks. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pp. 324–327. IEEE, 1994.

Dean, Jeffrey, Corrado, Greg, Monga, Rajat, Chen, Kai, Devin, Matthieu, Mao, Mark, Senior, Andrew, Tucker, Paul, Yang, Ke, Le, Quoc V, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pp. 1223–1231, 2012.

Duchi, John, Hazan, Elad, and Singer, Yoram. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 2011.

Hochreiter, Sepp and Schmidhuber, Jürgen. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Hochreiter, Sepp, Younger, A Steven, and Conwell, Peter R. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pp. 87–94. Springer, 2001.

Keskar, Nitish Shirish, Mudigere, Dheevatsa, Nocedal, Jorge, Smelyanskiy, Mikhail, and Tang, Ping Tak Peter.

On large-batch training for deep learning: Generalization gap and sharp minima. In *International Conference on Learning Representations*, 2016.

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

Kingma, Diederik P. and Ba, Jimmy. Adam: A method for stochastic optimization. In *International Conference on Learning Representations*, 2015.

Le, Quoc V., Ngiam, Jiquan, Coates, Adam, Lahiri, Ahbik, Prochnow, Bobby, and Ng, Andrew Y. On optimization methods for deep learning. In *Proceedings of the 28th International Conference on Machine Learning*, 2011.

LeCun, Yann A, Bottou, Léon, Orr, Genevieve B., and Müller, Klaus-Robert. Efficient backprop. In *Neural networks: Tricks of the trade*. Springer, 1998.

Liu, Dong C and Nocedal, Jorge. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1):503–528, 1989.

Loshchilov, Ilya and Hutter, Frank. SGDR: stochastic gradient descent with restarts. In *International Conference on Learning Representations*, 2017.

Martens, James. Deep learning via Hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning*, pp. 735–742, 2010.

Martens, James and Sutskever, Ilya. Training deep and recurrent networks with Hessian-free optimization. In *Neural networks: Tricks of the trade*, pp. 479–535. Springer, 2012.

Orchard, Jeff and Wang, Lin. The evolution of a generalized neural learning rule. In *2016 International Joint Conference on Neural Networks (IJCNN)*, pp. 4688–4694, 2016.

Pascanu, Razvan and Bengio, Yoshua. Revisiting natural gradient for deep networks. *arXiv preprint arXiv:1301.3584*, 2013.

Pascanu, Razvan, Mikolov, Tomas, and Bengio, Yoshua. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, 2013.

Ravi, Sachin and Larochelle, Hugo. Optimization as a model for few-shot learning. In *International Conference on Learning Representations*, 2017.

Riedmiller, Martin and Braun, Heinrich. RPROP - a fast adaptive learning algorithm. In *Proc. of ISCIS VII, Universitat*. Citeseer, 1992.

Runarsson, Thomas P. and Jonsson, Magnus T. Evolution and design of distributed learning rules. In *IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks*, 2000.

Schaul, Tom, Zhang, Sixin, and LeCun, Yann. No more pesky learning rates. In *International Conference on Machine Learning*, 2013.

Schmidhuber, Juergen. Steps towards 'self-referential' neural learning: A thought experiment. Technical report, University of Colorado Boulder, 1992.

Schraudolph, Nicol N. Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 14(7):1723–1738, 2002.

Schulman, John, Levine, Sergey, Abbeel, Pieter, Jordan, Michael I, and Moritz, Philipp. Trust region policy optimization. In *International Conference on Machine Learning*, pp. 1889–1897, 2015.

Shazeer, Noam, Mirhoseini, Azalia, Maziarz, Krzysztof, Davis, Andy, Le, Quoc, Hinton, Geoffrey, and Dean, Jeff. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. In *International Conference on Learning Representations*, 2017.

Wichrowska, Olga, Maheswaranathan, Niru, Hoffman, Matthew W., Colmenarejo, Sergio Gomez, Denil, Misha, de Freitas, Nando, and Sohl-Dickstein, Jascha. Learned optimizers that scale and generalize. In *International Conference on Machine Learning*, 2017.

Williams, Ronald J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Machine Learning*, 1992.

Wu, Yonghui, Schuster, Mike, Chen, Zhifeng, Le, Quoc V., Norouzi, Mohammad, Macherey, Wolfgang, Krikun, Maxim, Cao, Yuan, Gao, Qin, Macherey, Klaus, Klingner, Jeff, Shah, Apurva, Johnson, Melvin, Liu, Xiaobing, Kaiser, Lukasz, Gouws, Stephan, Kato, Yoshikiyo, Kudo, Taku, Kazawa, Hideto, Stevens, Keith, Kurian, George, Patil, Nishant, Wang, Wei, Young, Cliff, Smith, Jason, Riesa, Jason, Rudnick, Alex, Vinyals, Oriol, Corrado, Greg, Hughes, Macduff, and Dean, Jeffrey. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Zagoruyko, Sergey and Komodakis, Nikos. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

Zeiler, Matthew D. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

Zhang, Chiyuan, Bengio, Samy, Hardt, Moritz, Recht, Benjamin, and Vinyals, Oriol. Understanding deep learning requires rethinking generalization. In *International Conference on Learning Representations*, 2017.

Zoph, Barret and Le, Quoc V. Neural Architecture Search with reinforcement learning. In *International Conference on Learning Representations*, 2017.