Gradient Descent is a Technique for Learning to Learn

ISSN: 2311-8636 (Print)
ISSN: 2312-2021 (Online)

Source of Support: Nil

No Conflict of Interest: Declared

***Email for correspondence:**
bynagari.gs@gmail.com

Taposh Kumar Neogy[1], Naresh Babu Bynagari[2*]

[1]Assistant Professor of Accounting, Department of Business Administration, Institute of Business Administration (IBA), Rajshahi (under National University), **BANGLADESH**
[2]Andriod Developer, Keypixel Software Solutions, 777 Washington rd Parlin NJ 08859, Middlesex, **USA**

ABSTRACT

In machine learning, the transition from hand-designed features to learnt features has been a huge success. Regardless, optimization methods are still created by hand. In this study, we illustrate how an optimization method's design can be recast as a learning problem, allowing the algorithm to automatically learn to exploit structure in the problems of interest. On the tasks for which they are taught, our learnt algorithms, implemented by LSTMs, beat generic, hand-designed competitors, and they also adapt well to other challenges with comparable structure. We show this on a variety of tasks, including simple convex problems, neural network training, and visual styling with neural art.

Keywords: Gradient decent, Long Short Term Memory (LSTM), Gauss-Newton matrix, Machine learning, Recurrent neural network (RNN)

INTRODUCTION

The problem of maximizing an objective function, $f(\theta)$ defined across some domain $\theta \in \ominus$ is frequently presented in machine learning problems. In this situation, the purpose is to identify the minimizer $\theta^* = argmin_{\theta \in \ominus} f(\theta)$. While any method capable of reducing this objective function can be used, for differentiable functions, the conventional approach is some type of gradient descent, which results in a series of updates.

$$\theta_{t+1} = \theta_{t-\alpha t} \nabla f(\theta_t)$$

However, the performance of vanilla gradient descent is limited since it only uses gradients and ignores second-order information. Traditional optimization algorithms rectify this behavior by rescaling the gradient step using curvature information, usually via the Hessian matrix of second-order partial derivatives, though other options such as the extended Gauss-Newton matrix or Fisher information matrix are also viable.

Much of today's optimization research is focused on developing update rules that are customized to certain classes of issues, with the types of problems that are of interest varying among research communities. For example, optimization methods specialized for high-dimensional, non-convex optimization problems have proliferated in the deep learning community. Momentum (Nesterov, 1983, Tseng, 1998), Rprop (Riedmiller and

Braun, 1993), Adagrad (Duchi et al., 2011), RMSprop (Tieleman and Hinton, 2012), and ADAM (Kingma and Ba, 2015) are only a few of them. When the structure of the optimization problem is known, more targeted strategies can be used (Martens and Grosse, 2015). Communities that prioritize sparsity, on the other hand, embrace completely different techniques (Donoho, 2006, Bach et al., 2012; Ganapathy, 2016). This is especially true in combinatorial optimization, where relaxations are frequently employed (Nemhauser and Wolsey, 1988).

This optimizer design industry enables various communities to develop optimization methods that take advantage of structure in their issues of interest at the risk of potentially poor performance on problems outside of that scope.

Furthermore, the No Free Lunch Theorems for Optimization (Wolpert and Macready, 1997) indicate that no algorithm can outperform a random strategy in expectation in combinatorial optimization. This shows that narrowing one's focus to a certain problem category is the only way to increase overall performance.

Objective of the Study

In this article, we propose to replace hand-designed update rules with a learnt update rule, which we refer to as the optimizer g and which has its own set of parameters ϕ. As a result, the optimize f of the form is updated. In what follows, we will use a recurrent neural network (RNN) to explicitly describe the update rule g, which keeps its own state and so dynamically updates as a function of its iterates:

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

LITERATURE REVIEW

Transfer learning and generalization

Ganapathy & Neogy (2017) create a method for building a learning algorithm that excels at a specific class of optimization problems. The ability to identify the class of problems we are interested in by casting algorithm design as a learning problem allows us to specify the class of problems we are interested in through example problem cases. This is in contrast to the traditional strategy of analyzing the features of intriguing problems and then hand-designing learning algorithms based on these discoveries (Bynagari, 2015).

In this context, it is instructive to analyze the notion of generalization. We have a specific function of interest in conventional statistical learning, and its behavior is constrained by a data collection of example function evaluations. We provide a set of inductive biases about how we think the function of interest should act at locations we haven't observed when picking a model, and generalization refers to the ability to anticipate the behavior of the target function at novel positions. Because the examples in our situation are themselves problem instances, generalization refers to the ability to transfer information between difficulties. Transfer learning is the term for this reusing of issue structure, and it is frequently addressed as a separate discipline.

However, if we look at the problem of transfer learning from a meta-learning viewpoint, we may see it as a problem of generalization, which is far well-studied in the machine learning community (Bynagari, 2016; Vadlamudi, 2015).

We may rely on deep networks' ability to generalize to new cases by learning intriguing sub-structures, which is one of the major success stories of deep learning. We hope to use

this generalization power in this paper, but also to extend it beyond simple supervised learning to the more general scenario of optimization.

A brief history and related work

Thrun and Pratt (1998) buttress the idea of using learning to learn or meta-learning to acquire knowledge or inductive biases has a long history. Lake et al. (2016) have lately made a strong case for its usefulness as a building element in artificial intelligence. Santoro et al. (2016) view multi-task learning as generalization, although unlike us, they train a base learner directly rather than via a training algorithm. In general, these concepts entail learning on two time scales: quick learning inside tasks and more slowly, Meta learning across a variety of tasks (Bynagari, 2017).

Schmidhuber (1992, 1993) which builds on Schmidhuber (1987) and analyzes networks that can adjust their own weights, is perhaps the most general method to meta-learning. End-to-end differentiability of such a system allows both the network and the learning algorithm to be taught using gradient descent with few constraints. This universality, however, comes at the cost of making the learning rules extremely difficult to train. Instead of gradient descent, Schmidhuber et al. (1997) employ the Success Story Algorithm to adjust their search strategy; a similar technique was recently taken by Daniel et al. (2016), who used reinforcement learning to train a controller for picking step-sizes.

Bengio et al. (1990, 1995) propose utilizing simple parametric principles to learn updates that avoid back-propagation. Bengio et al (1990, 1995) techniques could be described as learning to learn without gradient descent using gradient descent, which is relevant to the topic of this paper. Runarsson and Jonsson (2000) expand on this work by using a neural network instead of a simple rule. Fixed-weight recurrent neural networks can exhibit dynamic behavior without modifying their network weights, according to Cotter and Conwell (1990) and later Vadlamudi (2016). This has also been demonstrated in the context of filtering (Feldkamp and Puskorius, 1998), which is linked to basic multi-timescale optimizers (Sutton, 1992, Schraudolph, 1999).

Finally, the work of Younger et al. (2001) and Hochreiter et al. (2001) combines these disparate lines of inquiry by allowing backpropagation output from one network to input into an extra learning network, with both networks being trained simultaneously. Our meta-learning technique improves on previous work by altering the optimizer's network design in order to extend this approach to bigger neural-network optimization challenges.

Learning to learn with recurrent neural networks

We look at explicitly parameterizing the optimizer in this paper. As a result, in a minor case of notation misuse, The final optimizee parameters $\theta^*(f, \phi)$ will be written as a function of the optimizer parameters f and the specific function in question Then we can ask, "What does it mean for an optimizer to be?" good? We will write the predicted loss as given a distribution of functions f.

$$\mathcal{L}(\phi) = \mathbb{E}_f \left[f \left(\theta^*(f, \phi) \right) \right]$$

As previously stated, the update steps g_t will be the output of a recurrent neural network m, parameterized by ϕ, and whose state will be clearly denoted with h_t. Next, whereas the objective function just depends on the final parameter value, having an objective that depends on the entire optimization trajectory, for some horizon T, will be useful for training the optimizer.

$$\mathcal{L}(\phi) = \mathbb{E}_f\left[\sum_{t=1}^{T} \Psi t\, f(\theta_t)\right]$$

Where $\theta_{t+1} = \theta_t + g_t$, $\begin{bmatrix} g_t \\ h_{t+1} \end{bmatrix} = m(\nabla t, h_t, \phi)$

The notation $\nabla_t = \nabla_\theta f(\theta_t)$ is used to represent the arbitrary weights associated with each time-step. When $w_t = 1[t = T]$, this formulation is equivalent to $\mathcal{L}(\phi) = \mathbb{E}_f\left[f\left(\theta^*(f,\phi)\right)\right]$, although time-step is explained why having other weights can be advantageous later.

Using gradient descent on ϕ., we can reduce the value of $\mathcal{L}(\phi)$. By choosing a random function f and applying backpropagation to the computational graph in Figure 1, the gradient estimate $\partial\mathcal{L}(\phi)/\partial\phi$ can be obtained. Gradients are allowed to flow along the solid edges of the graph, but they are not allowed to flow along the dashed edges. Ignoring gradients along the dashed edges entails assuming that the optimizee's gradients are independent of the optimizer parameters, i.e. $\frac{\partial \nabla_t}{\partial \phi} = 0$. We can skip computing the second derivatives of f because of this assumption.

When we look at the gradient in $\mathcal{L}(\phi) = \mathbb{E}_f[\sum_{t=1}^{T} \Psi t\, f(\theta_t)]$, we observe that it is non-zero only for terms where $w_t \neq 0$. The gradients of trajectory prefixes are 0 if we choose $w_t = 1[t = T]$ to match the original issue, and only the last optimization step gives information for training the optimizer. Backpropagation through Time (BPTT) is rendered ineffective as a result of this. The solution is to relax the target so that $w_t > 0$ at intermediate locations along the journey. This modifies the goal function yet allows us to train the optimizer on partial paths. For the sake of simplicity, $w_t = 1$ for every t in all of our tests.

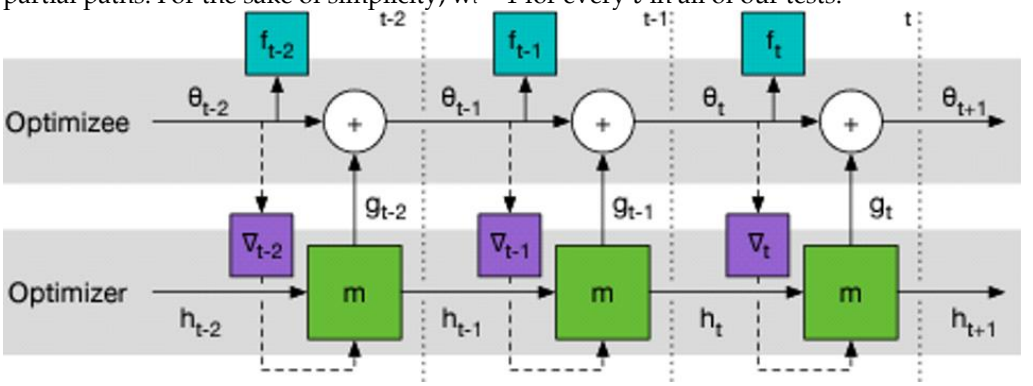

Figure 1: Computational graph used for computing the gradient of the optimizer

Coordinate wise LSTM optimizer

One of the difficulties in using RNNs in our situation is that we need to be able to optimize tens of thousands of parameters. It is not possible to optimize at this scale with a fully linked RNN since it would require a large hidden state and a large number of parameters. To get around this problem, we'll utilize an optimizer called m, which works coordinate wise on the parameters of the objective function, similar to RMSprop and ADAM. This coordinate wise network architecture enables us to define the optimizer with a very small network that only looks at a single coordinate and share optimizer parameters across different optimize parameters (Bynagari, 2014).

Separate activations for each goal function parameter are used to achieve different behavior on each coordinate. This arrangement has the wonderful effect of making the optimizer invariant to the order of parameters in the network, because the same update rule is applied independently on each coordinate, in addition to allowing us to utilize a small network for this optimizer. A two-layer Long Short Term Memory (LSTM) network is used (Hochreiter and Schmidhuber, 1997) with the now-standard forget gate design to implement the update rule for each coordinate. The network receives the optimize gradient for a single coordinate as well as the previous hidden state as inputs and returns the update for the corresponding optimize parameter as output (Ganapathy, 2015). This architecture, seen in Figure 2, will be referred to as an LSTM optimizer.

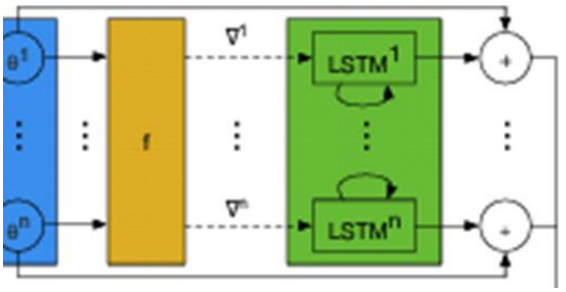

Figure 2: One step of an LSTM optimizer

Preprocessing and post processing

The magnitudes of optimizer inputs and outputs vary greatly depending on the type of function being optimized, although neural networks typically perform well only with inputs and outputs that are neither too little nor too large. In reality, rescaling an LSTM optimizer's inputs and outputs with appropriate constants (shared across all time steps and functions f) is enough to avoid this problem. In Appendix A, we offer a new strategy for preparing optimizer inputs that is more robust and yields somewhat better results.

METHODS

The trained optimizers in all of the studies use two-layer LSTMs with 20 hidden units in each layer. As mentioned in Section 2, each optimizer is trained by minimizing $\mathcal{L}(\phi) = \mathbb{E}_f[\sum_{t=1}^{T} \Psi t\, f(\theta_t)]$ using truncated BPTT. The minimization is done with ADAM and a learning rate determined by a random search. In order to avoid overfitting the optimizer, we employ early halting when training it. We freeze the optimizer parameters and measure its performance after each epoch (a defined number of learning steps). We choose the best optimizer (based on the final validation loss) and present its average performance on a set of randomly selected test problems. Our learned optimizers are compared to conventional Deep Learning optimizers such as SGD, RMSprop, ADAM, and Nesterov's accelerated gradient (NAG). We tweaked the learning rate for each of these optimizers and each problem, and reported results with the rate that gave the best final error for each problem. We utilize the default settings from Torch7's optim package when an optimizer contains more parameters than just a learning rate (decay coefficients for ADAM). All optimize parameters' initial values were drawn from an IID Gaussian distribution.

QUADRATIC FUNCTIONS

We examine training an optimizer on a basic class of synthetic 10-dimensional quadratic functions in this experiment. We focus on minimizing functions of the type for various

10x10 matrices W and 10-dimensional vectors y, the elements of which are taken from an IID Gaussian distribution. Optimizers were taught by optimizing random functions from this family and then tested on new samples from the same distribution (Ganapathy, 2017). Each function was made to be 100 percent efficient. The trained optimizers were unrolled for 20 steps, and the trained optimizers were unrolled for 20 steps. We didn't utilize any kind of preparation, and post processing.

$$f(\theta) = \|W\theta - y\|_2^2$$

The left plot of Figure 3 shows learning curves for many optimizers, averaged over numerous functions. Each curve represents the average performance of one optimization procedure across multiple test functions; the solid curve represents learnt optimizer performance, while the dashed curves represent typical baseline optimizer performance. In this case, it is evident that the learnt optimizers outperform the baselines significantly.

Training a small neural network on MNIST

We investigate whether trainable optimizers can learn to optimize a tiny neural network on MNIST, as well as how the taught optimizers generalize to functions other than those on which they were trained. To do this, we train the optimizer to optimize a base network and then test a variety of network architecture and training technique changes.

We investigate whether trainable optimizers can learn to optimize a tiny neural network on MNIST, as well as how taught optimizers generalize to functions other than those on which they were trained. To do this, we train the optimizer to optimize a base network and then test a variety of network architecture and training technique adjustments.

The cross entropy of a tiny MLP with parameters is the objective function $f(\theta)$ in this case.

The gradients $\partial f(\theta)/\partial\theta$ and the values of f are approximated using random minibatches of 128 instances. The base network is an MLP with a sigmoid activation function and one hidden layer of 20 units. The initial value 0 and unpredictability in mini-batch selection are the only sources of variation between runs. The trained optimizers were unrolled for 20 steps after each optimization was done for 100 stages. We applied the input preprocessing suggested in Appendix A and rescaled the LSTM outputs by 0.1.

The middle plot of Figure 4 shows learning curves for the basic network using several optimizers. Although NAG, ADAM, and RMSprop perform similarly in this experiment, the LSTM optimizer surpasses them by a wide margin. The right plot in Figure 3 compares the performance of the LSTM optimizer when it is allowed to run for 200 steps after being trained for 100 steps. We used the LSTM optimizer from the last experiment in this comparison, and we can see that the LSTM optimizer outperforms the baseline optimizers on this task.

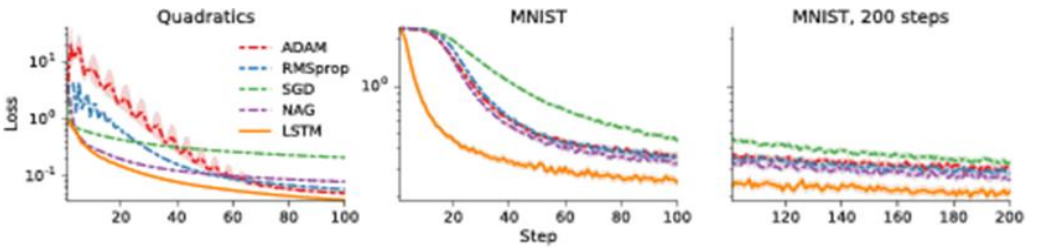

Figure 3: Comparisons between learned and hand-crafted optimizers performance

Generalization to different architectures

Three instances of using the LSTM optimizer to train networks with different topologies than the base network on which it was trained are shown in Figure 4.

- an MLP with 40 hidden units instead of 20,
- a network with two hidden layers instead of one, and
- a network utilizing ReLU activations instead of sigmoid activations are the alterations (from left to right). Despite working outside of its training regime, the

LSTM optimizer generalizes effectively and continues to outperform the hand-designed baselines in the first two examples.

The dynamics of the learning technique are sufficiently different when the activation function is changed to ReLU, however, that the learnt optimizer is no longer able to generate. Finally, we present the results of systematically modifying the tested architecture in Figure 5; for the LSTM results, we used the optimizer trained with one layer of 20 units and sigmoid non-linearities. It's worth noting that in this case, where the test-set problems are sufficiently close to those in the training set, the baseline optimizers perform even better.

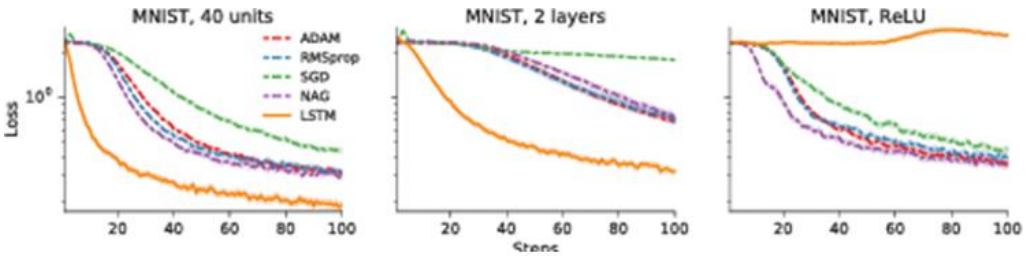

Figure 4: Comparisons between learned and hand-crafted optimizers performance

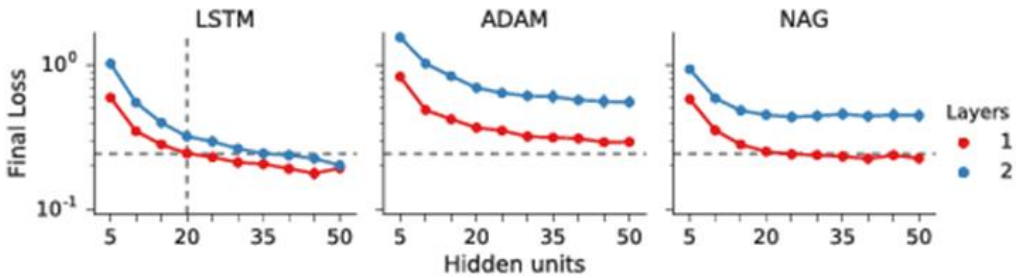

Figure 5: Systematic study of final MNIST performance as the optimizer architecture is varied, using sigmoid non-linearities.

Training a convolutional network on CIFAR-10

The trained neural optimizers are then put to the test on the CIFAR-10 dataset (Krizhevsky, 2009) to see how well they can improve classification performance. We employed a model with both convolutional and feed-forward layers in these tests. The model utilized in these trials consists of three convolutional layers with maximum pooling followed by a fully-connected layer with 32 hidden units; all non-linearities were ReLU activations with batch normalization.

The coordinate wise network decomposition and employed in the preceding experiment—uses a single LSTM architecture for each optimize parameter, with shared weights but independent hidden states. Due to the differences between the fully connected and convolutional layers, we discovered that this breakdown was insufficient for the model architecture presented in this section. Instead, we introduce two LSTMs into the optimizer: one suggests parameter updates for fully connected layers, while the other updates the parameters of convolutional layers. We still use a coordinate wise decomposition with shared weights and individual hidden states, same as the prior LSTM optimizer, but LSTM weights are now shared only across parameters of the same type (fully-connected vs. convolutional).

Figure 6 shows the performance of this trained optimizer in comparison to the baseline approaches. The results of applying the optimizer to fit a classifier on a held-out test set are shown in the left-most graphic. The next two plots on the right show the trained optimizer's performance on modified datasets that only contain a subset of the labels, such as the CIFAR-2 dataset, which only contains data for two of the ten labels. We also included an LSTM-sub optimizer that was only trained on the held-out labels.

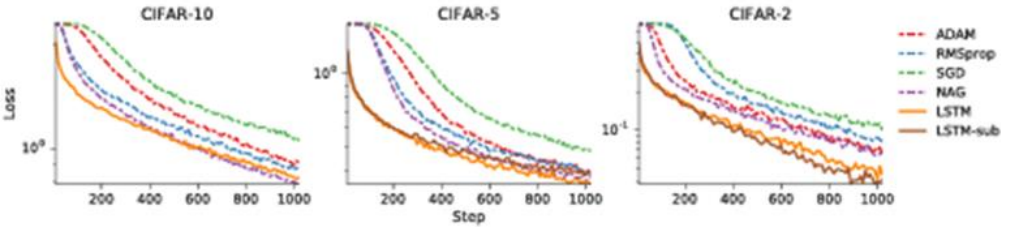

Figure 6: Optimization performance on the CIFAR-10 and Subsets

We can see that the LSTM optimizer learns substantially faster than the baseline optimizers in all of these situations, with large performance improvements for the CIFAR-5 and especially the CIFAR-2 datasets. We can also show that optimizers trained on a disjoint subset of the data are unaffected by this change and transfer well to the new dataset.

Neural Art

Because each content and style image pair generates a novel optimization problem, the recent work on artistic style transfer using neural networks, or Neural Art (Gatys et al., 2015), provides a perfect testbed for our strategy. Each Neural Art problem begins with a content image, c, and a style image, s, and is defined as follows:

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

The stylized image is the minimizer of f. The first two terms attempt to match the styled image's content and style to that of their first argument, while the third term is a regularizer that favors smoothness in the stylized image (Gatys et al., 2015) has further information.

Only one style and 1800 content photos from ImageNet were used to train optimizers (Deng et al., 2009). 100 content images are chosen at random for testing and 20 content images are chosen at random for validation of trained optimizers. We use 64x64 content images from ImageNet and one fixed style image to train the optimizer. We then put it to the test to see how well it adapts to a different visual style and higher quality (128x128).

Each image was optimized for 128 steps, with 32 steps being unrolled using trained optimizers. The effect of utilizing the LSTM optimizer to style two different photos is seen in Figure 6. The LSTM optimizer does not employ any postprocessing and uses the input preprocessing provided.

Figure 7 shows how the LSTM optimizer compares to other optimization techniques. When the resolution and style of the image are the same as the ones on which it was trained, the LSTM optimizer surpasses all regular optimizers. Furthermore, it continues to work admirably even when the resolution and style are altered during the test.

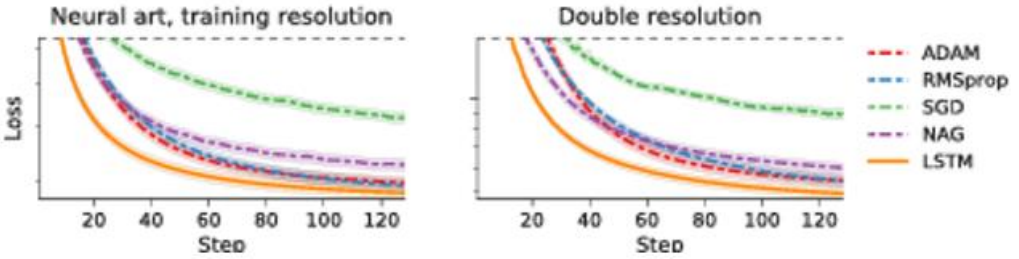

Figure 7: The LSTM optimizer compares to other optimization techniques

CONCLUSION

We've shown how to turn the design of optimization algorithms into a learning issue, allowing us to train optimizers that specialize in specific functions classes. Learned neural optimizers outperform state-of-the-art optimization approaches in deep learning, according to our tests. The LSTM optimizer, for example, was able to generalize to tasks with 49,152 parameters, multiple styles, and distinct content images at the same time, after being trained on 12,288 parameter neural art tasks. When shifting to alternative architectures in the MNIST test, we observed equally excellent results. When moving to datasets selected from the same data distribution, the results of the CIFAR image classification challenge reveal that LSTM optimizers outperform hand-engineered optimizers.

REFERENCES

Bach, F., R. Jenatton, J. Mairal, and G. Obozinski. 2012. Optimization with sparsity-inducing penalties. Foundations and Trends in Machine Learning, 4(1):1–106.

Bengio, S., Y. Bengio, and J. Cloutier. 1995. On the search for new learning rules for ANNs. Neural Processing Letters, 2(4):26–30.

Bengio, Y., S. Bengio, and J. Cloutier. 1990. Learning a synaptic learning rule. Université de Montréal, Département d'informatique et de recherche opérationnelle.

Bynagari, N. B. (2014). Integrated Reasoning Engine for Code Clone Detection. *ABC Journal of Advanced Research*, *3*(2), 143-152. https://doi.org/10.18034/abcjar.v3i2.575

Bynagari, N. B. (2015). Machine Learning and Artificial Intelligence in Online Fake Transaction Alerting. *Engineering International*, *3*(2), 115-126. https://doi.org/10.18034/ei.v3i2.566

Bynagari, N. B. (2016). Industrial Application of Internet of Things. *Asia Pacific Journal of Energy and Environment*, *3*(2), 75-82. https://doi.org/10.18034/apjee.v3i2.576

Bynagari, N. B. (2017). Prediction of Human Population Responses to Toxic Compounds by a Collaborative Competition. *Asian Journal of Humanity, Art and Literature*, *4*(2), 147-156. https://doi.org/10.18034/ajhal.v4i2.577

Cotter N. E. and P. R. Conwell. 1990. Fixed-weight networks can learn. In International Joint Conference on Neural Networks, pages 553–559.

Daniel, C., J. Taylor, and S. Nowozin. 2016. Learning step size controllers for robust neural network training. In Association for the Advancement of Artificial Intelligence.

Deng, J., W. Dong, R. Socher, L.J. Li, K. Li, and L. Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In Computer Vision and Pattern Recognition, pages 248–255. IEEE.

Donoho. D. L. 2006. Compressed sensing. Transactions on Information Theory, 52(4):1289–1306.

Duchi, J., E. Hazan, and Y. Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research, 12:2121–2159.

Feldkamp L. A. and G. V. Puskorius. 1998. A signal processing framework based on dynamic neural networks with application to problems in adaptation, filtering, and classification. Proceedings of the IEEE, 86(11): 2259–2277.

Ganapathy, A. (2015). AI Fitness Checks, Maintenance and Monitoring on Systems Managing Content & Data: A Study on CMS World. *Malaysian Journal of Medical and Biological Research*, *2*(2), 113-118. https://doi.org/10.18034/mjmbr.v2i2.553

Ganapathy, A. (2016). Speech Emotion Recognition Using Deep Learning Techniques. *ABC Journal of Advanced Research*, *5*(2), 113-122. https://doi.org/10.18034/abcjar.v5i2.550

Ganapathy, A. (2017). Friendly URLs in the CMS and Power of Global Ranking with Crawlers with Added Security. *Engineering International*, *5*(2), 87-96. https://doi.org/10.18034/ei.v5i2.541

Ganapathy, A., & Neogy, T. K. (2017). Artificial Intelligence Price Emulator: A Study on Cryptocurrency. *Global Disclosure of Economics and Business*, *6*(2), 115-122. https://doi.org/10.18034/gdeb.v6i2.558

Gatys, L. A., A. S. Ecker, and M. Bethge. 2015. A neural algorithm of artistic style. arXiv Report 1508.06576.

Hochreiter S. and J. Schmidhuber. 1997. Long short-term memory. Neural computation, 9(8):1735–1780.

Hochreiter, S., A. S. Younger, and P. R. Conwell. 2001. Learning to learn using gradient descent. In International Conference on Artificial Neural Networks, pages 87–94. Springer.

Kingma D. P. and J. Ba. 2015. Adam: A method for stochastic optimization. In International Conference on Learning Representations.

Krizhevsky. A. 2009. Learning multiple layers of features from tiny images. Technical report.

Lake, B. M., T. D. Ullman, J. B. Tenenbaum, and S. J. Gershman. 2016. Building machines that learn and think like people. arXiv Report 1604.00289.

Martens J. and R. Grosse. 2015. Optimizing neural networks with Kronecker-factored approximate curvature. In International Conference on Machine Learning, pages 2408–2417.

Nemhauser G. L. and L. A. Wolsey. 1988. Integer and combinatorial optimization. John Wiley & Sons.

Nesterov. Y. 1983. A method of solving a convex programming problem with convergence rate o (1/k2). In Soviet Mathematics Doklady, volume 27, pages 372–376.

Riedmiller M. and H. Braun. 1993. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In International Conference on Neural Networks, pages 586–591.

Runarsson and M. T. Jonsson. 2000. Evolution and design of distributed learning rules. In IEEE Symposium on Combinations of Evolutionary Computation and Neural Networks, pages 59–63. IEEE.

Santoro, A., S. Bartunov, M. Botvinick, D. Wierstra, and T. Lillicrap. 2016. Meta-learning with memory-augmented neural networks. In International Conference on Machine Learning.

Schmidhuber, J., J. Zhao, and M. Wiering. Shifting inductive bias with success-story algorithm, adaptive Levin search, and incremental self-improvement. Machine Learning, 28(1):105–130, 1997.

Schmidhuber. J. 1987. Evolutionary principles in self-referential learning; On learning how to learn: The meta-meta-hook. PhD thesis, Institut f. Informatik, Tech. Univ. Munich.

Schmidhuber. J. 1992. Learning to control fast-weight memories: An alternative to dynamic recurrent networks.Neural Computation, 4(1):131–139.

Schmidhuber. J. 1993. A neural network that embeds its own meta-levels. In International Conference on Neural Networks, pages 407–412. IEEE.

Schraudolph. N. N. 1999. Local gain adaptation in stochastic gradient descent. In International Conference on Artificial Neural Networks, volume 2, pages 569–574.

Sutton. R. S. 1992. Adapting bias by gradient descent: An incremental version of delta-bar-delta. In Association for the Advancement of Artificial Intelligence, pages 171–176.

Thrun S. and L. Pratt. 1998. Learning to learn. Springer Science & Business Media.

Tieleman T. and G. Hinton. 2012. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 4:2.

Tseng. P. 1998. An incremental gradient (-projection) method with momentum term and adaptive stepsize rule. Journal on Optimization, 8(2):506–531.

Vadlamudi, S. (2015). Enabling Trustworthiness in Artificial Intelligence - A Detailed Discussion. *Engineering International*, *3*(2), 105-114. https://doi.org/10.18034/ei.v3i2.519

Vadlamudi, S. (2016). What Impact does Internet of Things have on Project Management in Project based Firms?. *Asian Business Review*, *6*(3), 179-186. https://doi.org/10.18034/abr.v6i3.520

Wolpert D. H. and W. G. Macready. 1997. No free lunch theorems for optimization. Transactions on Evolutionary Computation, 1(1):67–82.

Younger, A. S., S. Hochreiter, and P. R. Conwell. 2001. Meta-learning with backpropagation. In International Joint Conference on Neural Networks.

--0--