# Investigating "Learning to learn by gradient descent by gradient descent"

**Nicholas Bradford** [1] **Michael Giancola** [1] **Tim Petri** [1]

## Abstract

Significant research has been conducted on the design of gradient-based optimization algorithms for neural networks such as Adadelta, RMSProp, and Adam, as well as adapting optimizer parameters to fit a specific problem. However, the paper "Learning to learn by gradient descent by gradient descent" (Andrychowicz et al., 2016) has demonstrated that superior optimizers can instead be learned automatically using Long Short-Term Memory (LSTM) recurrent neural networks, which can then generalize to new problem domains (similar network architectures). The work presented in this paper documents our attempt to reproduce their results to train an LSTM optimizer used for training a simple neural network, and then test its capability for generalization. Although we were able to implement their procedure for optimizer training using TensorFlow, we were unable to reproduce their exceptional results.

## 1. Introduction

We can define the task of training a neural network (or any other function) as minimizing an optimizee function $f(\theta)$ over parameter domain $\theta$. The standard gradient descent approach is to perform a sequence of updates according to a learning rate $\alpha$ and the gradients of $f$:

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

In most deep learning work done today, researchers expend substantial effort to select an optimizer and associated parameters for their particular problem, as (Wolpert & Macready, 1997) demonstrated that no single optimization algorithm can be considered superior for use in all classes of problems. In contrast to this typical approach, our goal is to learn an optimizer function $g$ with its parameters $\phi$ to perform updates of the form:

$$\theta_{t+1} = \theta_t - g_t(\nabla f(\theta), \Phi)$$

Learning this optimizer becomes a problem of generalization, as we seek to develop a method of automatically building $g$ such that it performs well on a class of optimization problems (i.e. network architectures).

The primary goal of this research was to reimplement the meta-learner in the paper (Andrychowicz et al., 2016). Specifically, we wanted to tackle the task of optimizing a simple feed-forward neural network used for classifying MNIST images. From there, we seeked to understand the circumstances under which a learned optimizer will generalize well to another problem (eg. how well will the optimizer learned for MNIST perform on a different dataset?). (Andrychowicz et al., 2016) didn't discuss this in much detail, so our goal was to explore this problem further. We also experimented with taking a learned optimizer for a network, changing the network's activation functions, and evaluating the performance.

### 1.1. Research contributions

We produced an implementation of (Andrychowicz et al., 2016) that is succinct while also easily extendable. Our hope is that such open-source code will foster improved understanding among interested readers and researchers, as well as provide a starting point for future work.

## 2. Related Work

Various approaches to the task of meta-learning have been proposed. Schmidhuber introduced the concept of a network which can modify its own weights as a form of meta-learning. (Schmidhuber, 1987), (Schmidhuber, 1992), and (Schmidhuber, 1993).

Later, (Younger et al., 2001) and (Hochreiter et al., 2001) showed that output from a network's backpropagation can be passed to another network in order to expedite learning of the first network.

Andrychowicz et al. successfully trained LSTMs to learn how to optimize several different problems, including simple convex problems, image classification, and neural art. Further, the paper demonstrated that these neural optimizers are able to generalize to small variations in the problem,

[*]Equal contribution [1]Worcester Polytechnic Institute. Correspondence to: Jacob Whitehill <jrwhitehill@wpi.edu>.

*CS525 Special Topics Course Final Project*, Worcester Polytechnic Institute. 

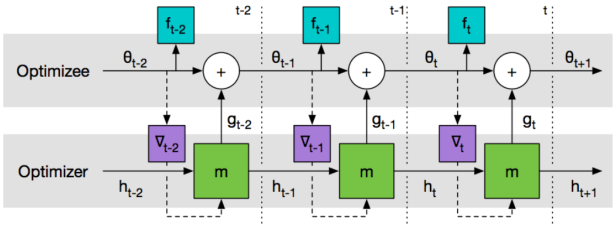


*Figure 1.* Computational graph for computing the optimizer's gradient (Andrychowicz et al., 2016)

.

such as a different number of hidden layers or hidden layer nodes. However, the LSTM optimizer was unable to generalize well across different activation functions (specifically, ReLU instead of sigmoid).(Andrychowicz et al., 2016)

## 3. Proposed Method

In order to train the LSTM optimizer, at every step we ran an entire optimization sequence (100 mini-batches) on the optimizee network. The inputs to the LSTMs are the gradients produced by the optimizee cost, and the outputs are a set of updates to the optimizee's parameters (see Figure 1). We then define the LSTM optimizer's cost to be equal to the sum of losses across every training step produced by the optimizee network. This approach unfortunately requires unrolling the entire computational graph of the optimizee training in order for TensorFlow to compute gradients, which is enormously computationally expensive (see Appendix A for a TensorBoard visualization).

In order to use significantly fewer parameters in the LSTM optimizer, a coordinatewise LSTM was used, which has shared parameters between cells but separate hidden states. This allows for use of a much smaller optimizer: a two-layer LSTM with 20 hidden units in each layer. Updates to the LSTM optimizer itself were done using Adam.

## 4. Experiment

Our testing code was modularized such that switching between problems for the LSTM to optimize was as simple as passing a different $f$ function, which takes as input a parameter vector, and outputs a cost (computation of the gradients is made generic by TensorFlow).

### 4.1. Multdimensional Quadratic Function

Our first experiment was implementing the simplest optimization problem in (Andrychowicz et al., 2016): that is, learning an optimizer to find the minimum of a 10-dimensional quadratic function (which is always the origin). We did proof-of-concept primarily for better understanding of how meta-learning worked, and also to verify that we set up the LSTM correctly.

The loss function for this problem took one parameter, $x$, which represents the x-coordinate of a point on the given quadratic function. The loss is computed by scaling $x$ by a random constant (to add noise) and squaring it.

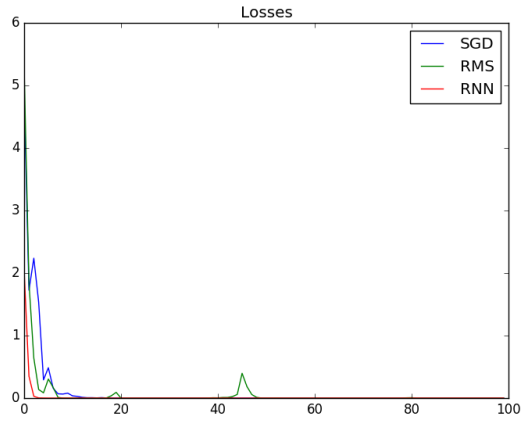


*Figure 2.* Performance of each optimizer on the quadratic function. RNN is our learned optimizer.

As expected, the learned optimizer converges more quickly than Stochastic Gradient Descent and RMSProp.

### 4.2. MNIST Network

Our base optimizee network had a single layer. The output layer always used softmax as its activation function. For MNIST, it was set to 784 inputs, 20 hidden units, and 10 outputs.

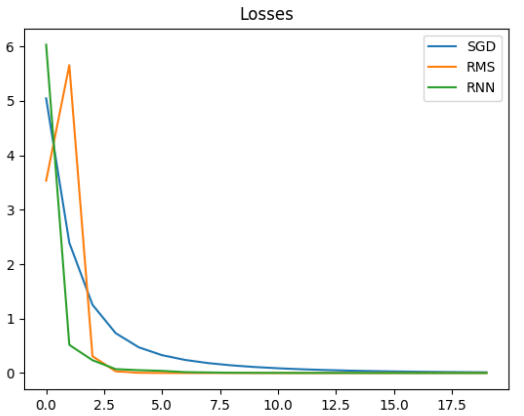


*Figure 3.* Performance of each optimizer on an MNIST network. RNN is our learned optimizer.

### 4.3. Generalizing Optimizers

This is the work that we did which was beyond (Andrychowicz et al., 2016). Our goal was to understand why an LSTM trained to optimize a particular network architecture would perform well (or poorly) when applied to a new architecture.

We started by creating a 3-layer network to learn MNIST. The network used ReLU as the activation function for its hidden layer. Then, once the LSTM had learned to optimize this network, we applied it to a new network which was also learning MNIST. However, it used sigmoid as the activation function for its hidden layer instead of ReLU.

## 5. Results

### 5.1. MNIST Network

Consider Figures 4 through 6, which evaluate the performance of the LSTM to train a network learning MNIST with ReLU as the activation function. Note that the graphs are all log-scale, and the x-axis is the training step for the MNIST network.

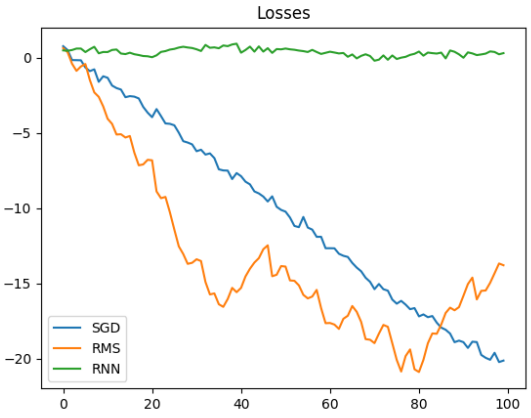


*Figure 4.* Log loss after 10 epochs of LSTM training

Notice that as the LSTM is trained longer, it is able to reduce the loss more. Again, since these graphs are all log-scale, all three optimizers are very close to zero. However, we believe the RNN line levels off because it is unable to appreciate the importance of the very small gradients that arise later in training.

### 5.2. Generalizing Optimizers

Figure 7 shows the result of training the LSTM on an MNIST network using the ReLU activation function, and evaluating it on an MNIST network using the sigmoid activation function.

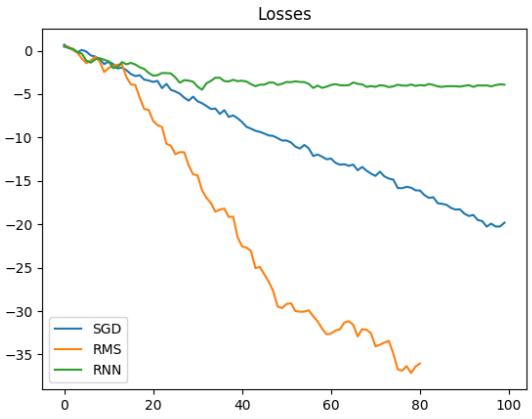


*Figure 5.* Log loss after 200 epochs of LSTM training

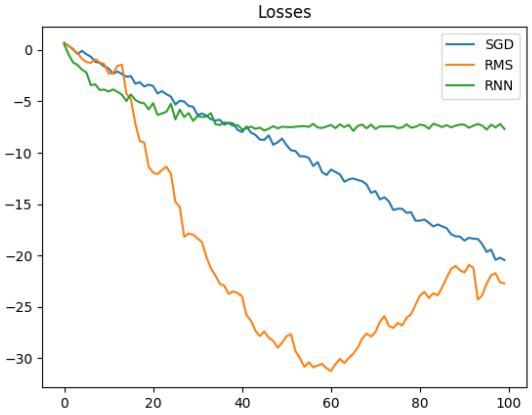


*Figure 6.* Log loss after 3000 epochs of LSTM training

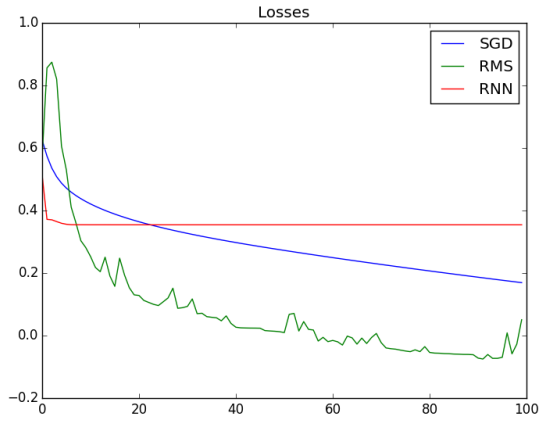


*Figure 7.* Log loss of LSTM when generalized

We found that, in this particular case, the LSTM is not able to generalize to a different type of network.

## 6. Discussion

Figure 8 below contains the original paper's results obtained by training the base network using generic and a neural optimizers.

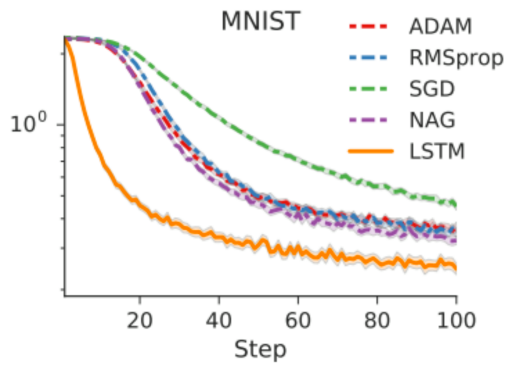


*Figure 8.* Log Loss of LSTM on MNIST in original paper (Andrychowicz et al., 2016)

The original paper showed that the LSTM optimizer not only was able to optimize a neural network for classifying MNIST images, but also that it did so better than generic optimizers. Our results also showed the feasibility of using a neural optimizer but we were not able to outperform Stochastic Gradient Descent or RMSProp with our LSTM. There were a few factors that may have contributed to this.

First of all, constructing and running experiments using the Tensorflow machine learning framework proved to be difficult. Training two neural networks on top of each other is not a basic use case and involved significant work constructing the appropriate computational graphs to allow for training and testing of our LSTM. Additionally, training was extremely expensive as the entire training loop of the base network (optimizee) had to be unrolled for each higher-level LSTM training step.

Secondly, the original paper discusses issues arrising when the LSTM has to handle gradients of very different magnitudes, as often is the case in the context of training neural networks. The gradient problem may have been the reason why our LSTM stopped improving the base network beyond a certain point, while the generic optimizers were able to keep lowering the overall loss. Gradient preprocessing is proposed as a solution by the original authors. Our attempts at preprocessing the gradients did not yield results comparable to those in the original paper.

## 7. Conclusions and Future Work

In summary, we reimplemented the general meta-learner described in (Andrychowicz et al., 2016). We then compared the LSTM's performance against SGD and RMSProp, and were unable to reproduce the results from (Andrychowicz et al., 2016). However, in many cases we were unable to reproduce the results because the detail in the original report was insufficient. (One illustrative example: the original report mentions preprocessing gradients by scaling by a constant; however, the constant is never given).

Although we produced the necessary code, due to time constraints were unable to complete our tests generalizing the LSTM to other optimization problems entirely (alternative number of hidden units, extra hidden layers, other activation functions). For example, one test that we didn't have time to complete was to train the LSTM to optimize for the MNIST problem, then evaluate it on the smile data set.

## References

Andrychowicz, Marcin, Denil, Misha, Gómez, Sergio, Hoffman, Matthew W, Pfau, David, Schaul, Tom, and de Freitas, Nando. Learning to learn by gradient descent by gradient descent. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 29*, pp. 3981–3989. Curran Associates, Inc., 2016.

Hochreiter, Sepp, Younger, A, and Conwell, Peter. Learning to learn using gradient descent. *Artificial Neural NetworksICANN 2001*, pp. 87–94, 2001.

Schmidhuber, Jurgen. Evolutionary principles in self-referential learning. *On learning how to learn: The meta-meta-... hook.) Diploma thesis, Institut f. Informatik, Tech. Univ. Munich*, 1987.

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

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

Wolpert, David H and Macready, William G. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

Younger, A Steven, Hochreiter, Sepp, and Conwell, Peter R. Meta-learning with backpropagation. In *Neural Networks, 2001. Proceedings. IJCNN'01. International Joint Conference on*, volume 3. IEEE, 2001.

# 8. Appendix A: Tensorboard Visualization

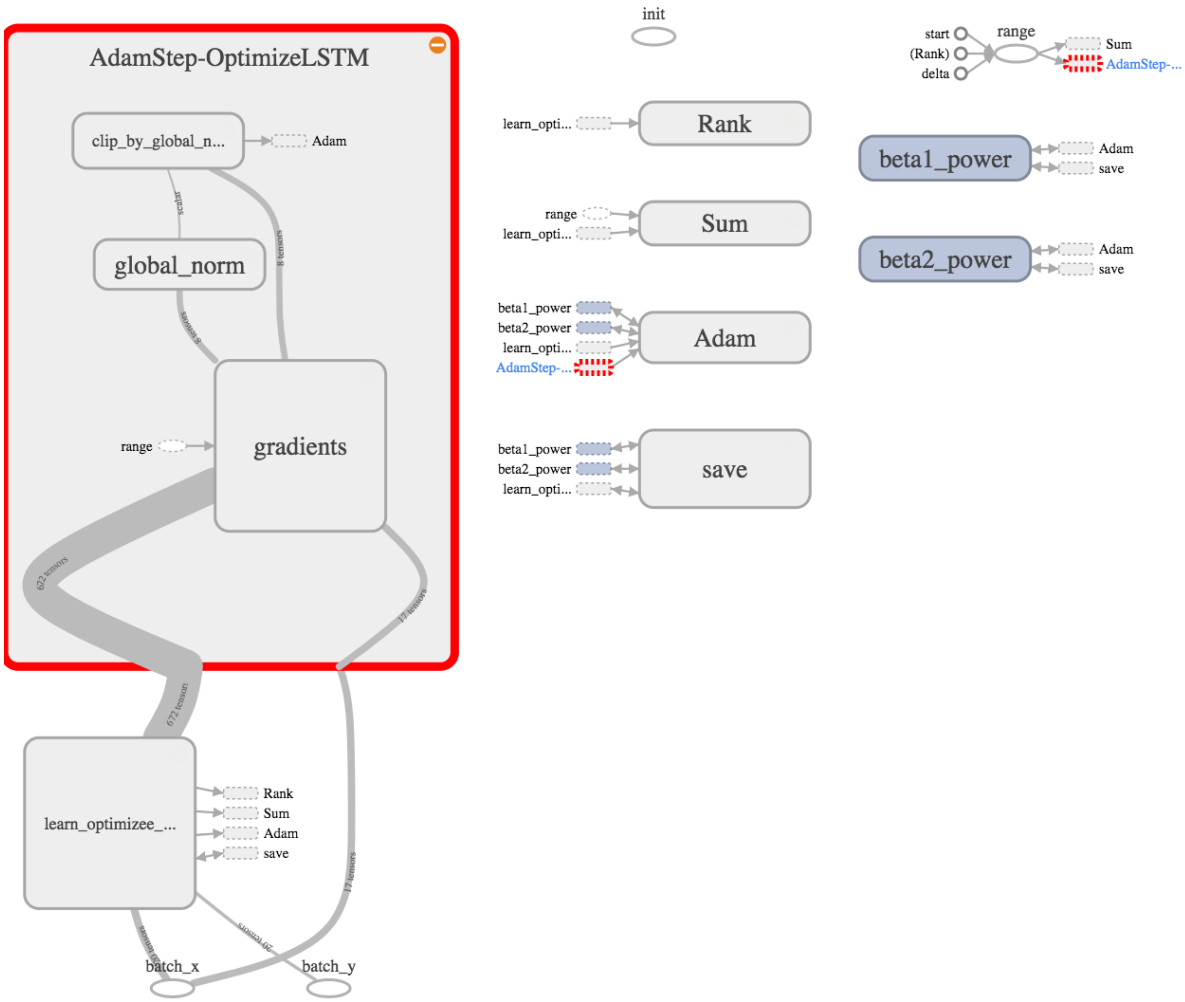


*Figure 9.* TensorBoard visualization for the TensorFlow computational graph.

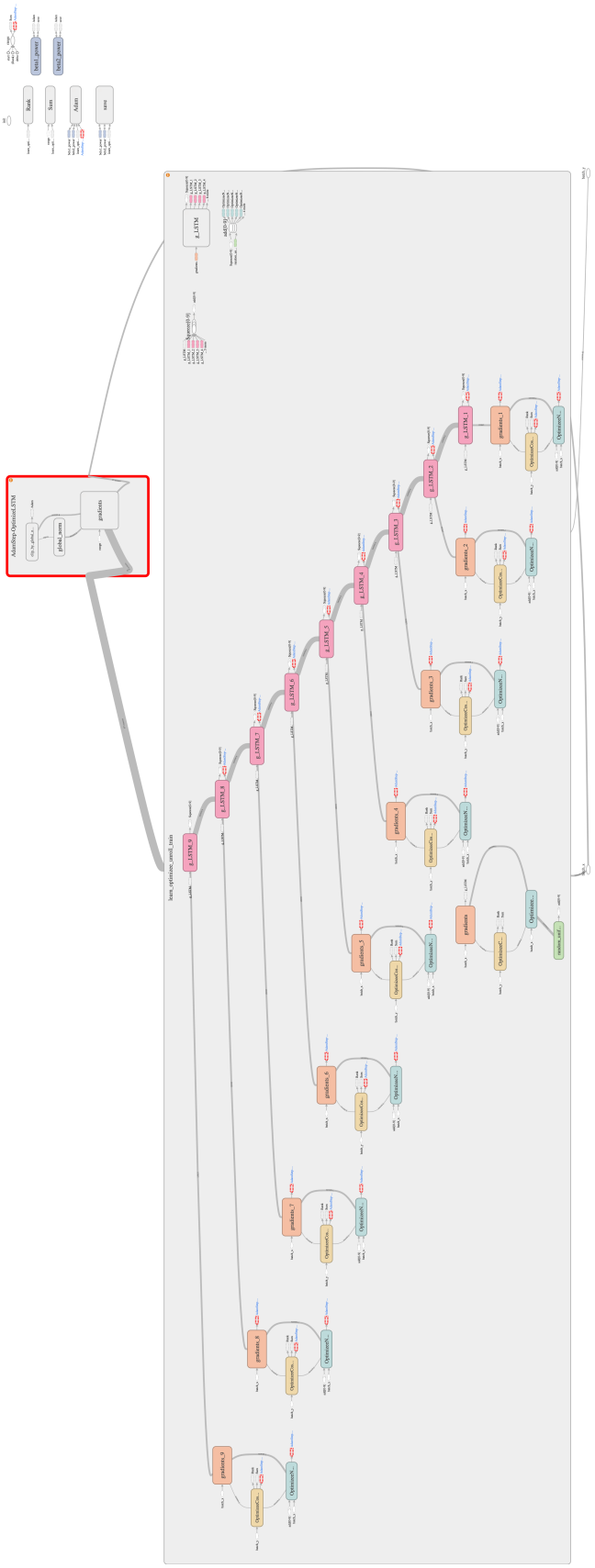

*Figure 10.* TensorBoard visualization for the TensorFlow computational graph. The unrolled training loop for the optimizee feedforward network is expanded (here for 10 training steps).