$Optimizers^3$ Outline

Alexander Brown (abrown53), Dichuan Gao (dgao7),
Chace Hayhurst (chayhurs), Alexander Young (ayoung38)

November 2020

1 Introduction

A *model* is a parameterized function

$$f_\theta : X \to Y$$

where X is the space of all possible inputs (say, all possible images), and Y is the space of all possible outputs (say, a binary set where the items indicate whether or not an input image contains a cat); and θ is the parameter on f. We denote Θ to be the parameter space, i.e. the set of all possible values θ can take on.

A *loss-function* is a function $l : \Theta \to \mathbb{R}$ measuring the performance of the model: the lower the value of l, the better the model. Most machine learning tasks boil down to minimizing the loss-function; that is, we seek

$$\theta^* = \arg\min_{\theta \in \Theta} l(\theta)$$

The idea of supervised learning is to successively approximate the optimal value θ^* by repeatedly updating θ so as to decrease the value of $l(\theta)$ gradually. This successive update requires an *update rule* $\theta_t \mapsto \theta_{t+1}$, or in other words, an *optimizer*, specifying how θ is to be updated. For example, gradient descent is the update rule

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta \, l(\theta)$$

In this course, all the optimizers we worked with are hand-crafted. The update rule is explicitly written in closed mathematical form.

But it is clear that an optimizer is itself a model. It takes as input the current parameter θ_t, together with some information about the loss-function (say, the gradient $\nabla_\theta \, l(\theta)$), and it returns as output a new parameter θ_{t+1}; or formally, an optimizer is a model

$$g : \Theta \times \Lambda \to \Theta$$

where Λ is a space of some information about the loss-function. Why, then, should we not make this optimizer itself trainable?

We will implement the 2016 paper "Learning to Learn by Gradient Descent" [1]. The idea is to parameterize the optimizer itself, so that it is itself a trainable model

$$g_\phi : \Theta \times \Lambda \to \Theta$$

where Λ is simply the space of possible gradients $\nabla_\theta \, l(\theta)$, and Φ is the parameter space for the optimizer, i.e. the set of all possible values for ϕ. We will create an RNN architecture for this optimizer, define a reasonable loss-function $\mathcal{L} : \Phi \to \mathbb{R}$ for the optimizer, and train it to become a good optimizer for several different supervised models.

In this sense, a model (say a model for categorizing MNIST data) will be an *optimizee*. Our RNN will be an *optimizer*. Our program, which trains the *optimizer*, will be an *optimizer optimizer*. Thus, we, the team of programmers writing this program, are *optimizer optimizer optimizers*. We are optimizers cubed.

1.1 Related Work

The process of "learning to learn" has a long history in the field of computer science. Recently, with the explosion in popularity of Neural Networks and the accessibly of DL libraries, there has been a large amount of time and resources invested into creating efficient and correct optimizing functions, which allow for significantly reduced training time and overall model correctness.

The standard approach has been to use hand designed optimizing functions, see Tseng [4] for examples of one such design. In this paper, Tseng proposes an optimization function that takes the "momentum" of the gradients into account. This addition can help prevent the optimization from getting stuck as the momentum ideally will carry optimization through some local minima. Furthermore, this paper proposes an adaptive step size, that only updates gradients when "sufficient progress has been made" which helps increase the stability of training using this optimizing function.

Deep learning based meta-learning involving recurrent neural networks has also been explored before, see A.S. Younger [6]. As Younger shows in this paper, gradient descent on recurrent neural networks can be used to form an efficient meta-learning system. He concludes that "any recurrent network topology and its corresponding learning algorithm(s) is a potential meta-learning system". Finally, he even goes on to show an LSTM based meta-learning model that can learn to classify any 2-d quadratic function with only 30 training examples!

Truly this is an important problem to solve, as the majority of problems one face in creating a neural network stem from how long it takes to train and how well the model converges. Meta learning has the ability to solve both of these problems by creating better update rules that have more stable convergence and are possibly more efficient when run on hardware.

Here is an existing implimentation of the paper! (that actually was trained using the same graphics card in my computer!)

2 Methodology

2.1 Data

We will start with randomly generated data sampled from known distributions, to test the simpler task of minimizing square loss on an optimizee model

$$l(\theta) = ||\mathbf{W}\theta - \mathbf{y}||^2$$

where $\mathbf{W}$ and $\mathbf{y}$ are randomly generated.

We will then move on to using a two-fully-connected-layers optimizee model f_θ on the MNIST dataset. For this, we will use the same dataset as provided for Assignment 1.

2.2 Method

We will use the same method as described in [1]. At a high level, the architecture consists of three ingredients: an RNN cell m; an update rule for optimizee parameters g, and a loss function for the optimizer $\mathcal{L}$.

We will start with an optimizee in mind. As usual, we denote f, θ, l for the optimizee model, parameter, and loss respectively.

Create an RNN cell m, with hidden state h and parameters ϕ. This cell will take as input the gradient of the optimizee's loss function with respect to the optimizee's parameters, evaluated at time t (for brevity we denote $\nabla_t = \nabla_\theta \, l(\theta)|_{\theta=\theta_t}$); and return as output a pair $\Delta\theta_t$ and h_{t+1}, which are respectively a vector of the same shape as the optimizee parameter θ, and the next hidden state. Formally: for each t,

$$\begin{pmatrix} \Delta\theta_t \\ h_{t+1} \end{pmatrix} = m_\phi(\nabla_t, h_t) \tag{1}$$

Then, our update rule g for optimizee parameters will simply be

$$\theta_{t+1} = \theta_t + \Delta\theta_t \tag{2}$$

And we will measure loss on the optimizer as a weighted time sum as follows:

$$\mathcal{L}(\phi) = \sum_{t=1}^{T} w_t \, l(\theta_t) \tag{3}$$

Where T is the window size of our RNN. Or, in cases where l is unknown at the time of computation for $\mathcal{L}$, or when l is stochastic, we take

$$\mathcal{L}(\phi) = \mathbb{E}_l \left[\sum_{t=1}^{T} w_t \, l(\theta_t) \right] \tag{4}$$

Where the expectation $\mathbb{E}$ is either taken in the Bayesian sense or in the frequentist sense, depending on whether l is unknown or stochastic.

Now, in practice, the optimizee parameter will be a collection of many weights, say, $\theta = (\theta^{(1)}, \ldots, \theta^{(m)})$. We will use the same RNN cell for the optimizer of each weight, but keep separate hidden states $h_t^{(1)}, \ldots, h_t^{(m)}$ for each weight. In other words, the hidden state of our RNN will be coordinate-wise.

For ease of computation for the loss of optimizer, we will simply weigh the optimizee loss at each time step equally, so that $w_t = 1$ for all $1 \leq t \leq T$.

To train our optimizer, we will need to compute the gradient $\nabla_\phi \mathcal{L}(\phi)$. We refer to page 4 of [1] for the computation graph through which this gradient is computed. Note that, when this gradient is computed, we assume $\frac{\partial \nabla_\theta l(\theta)}{\partial \phi} = 0$ (as shown by the dotted lines in the computation graph). Without this assumption, the gradient $\nabla_\phi \mathcal{L}$ would depend on the Hessian of l with respect to θ, which we must avoid.

Finally, we will train our optimizer using different hyperparameters, and look for the best possible performance.

2.3 Metrics

To determine the success of our model, we will run it to minimize the square loss on an optimizee model (discussed in the data section), and also to minimize loss on the MNIST data set. We will then do the same process using standard optimization functions (Adam, SGD, etc...) and chart the performance of loss at each epoch of training.

Accuracy doesn't really apply here, as there is no objectively correct or incorrect update. Furthermore, we expect that loss during training will not always be strictly going down, especially as the model nears convergence. A much better metric for correctness is seeing that the model converges in fewer or the same number of epochs as the typical optimization functions we are testing against.

The authors of the original paper were hoping to see that their network performed better than or comparable to standard optimization methods. To this end, they charted their model's performance versus the standard methods as we will do. They found that for certain architectures, their model was able to converge faster than any optimization function they tested with, with quite stable training as well.

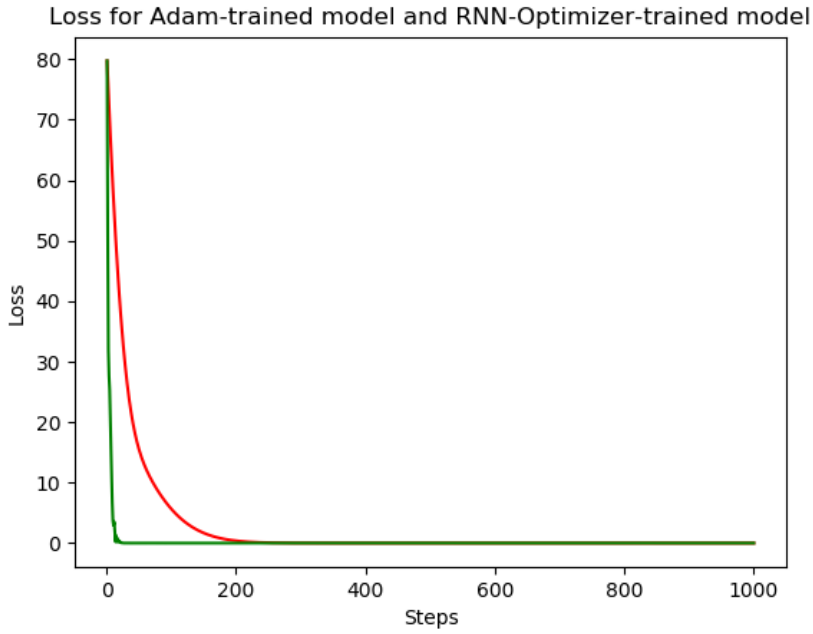
For a base goal, we aim to have a model that performs competitively (in terms of stability/epochs to convergence) with ADAM on both of the datasets we are testing on, as Adam was the hand designed function that performed the best on most of the experiments in the paper. For a target goal, we hope to perform better than ADAM on the square loss (proving that our model is better than most hand designed features for this simple task). For a stretch goal, we hope to perform better than ADAM on our MNIST experiments.

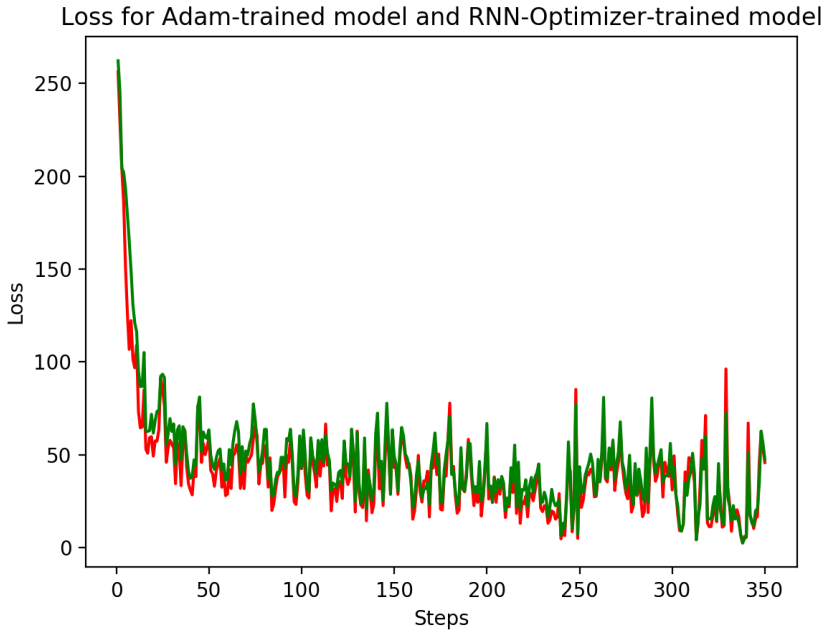
2.4 Ethics

The problem of vanilla gradient descent ignoring second-order properties has traditionally been mitigated by rescaling the gradient step using curvature information, and so by using specialized optimization methods like momentum, Adagrad, RMSprop, and Adam, all of which we have encountered in class and in lab. However, this approach introduces the possibility of human error in optimizer selection, and since these optimization methods tend to perform poorly on problems that lie outside of their scope, we run the risk of extremely shoddy results (and all the consequences that follow from the application of badly-trained models). The application of deep learning to the problem of optimizer design is fitting because we eliminate this human error, and the narrow scope of any individual optimizer, by having an RNN which is capable of learning the update rule for a larger variety of problems.
Like in the paper, we intend to compare our optimizer with the standard aforementioned optimizers - SGD, RMSprop, Adam, et alia - selecting for each one the learning rate which reports the best loss. Given that our optimizer is an RNN, we report the average performance on the test sets of whichever optimizer has the best final validation loss at time of stoppage. By applying our optimizer and the standard optimizers to multiple model architectures - i.e. 20 hidden units, 40 hidden units, 2 hidden layers, and with ReLU activation - and graphing the loss over the steps, we can quantify our performance. If the loss for our RNN optimizer drops as rapidly as (or more rapidly than) that of the standard optimizers, we can confidently say we have succeeded in achieving comparable or superior results. Of course this implies that we are evaluating success based on existing optimization methods, but given the generalizability of the RNN optimizer, even equivalent performance represents a satisfactory outcome.
As for our datasets, given that we are using MNIST, which is publicly available and well-curated, it is reasonable to say that there is relatively little bias or concern with means of collection. Furthermore, we expect that the randomness in the selection for the squared loss optimizee will erase any bias (as the points are completely randomly selected with no pattern between them).

3 Results

The following are the results of the loss for our optimizer and other standard optimizers on the quadratic functions and MNIST models respectively:





The green line is the lossed produced using our optimizer, while the red line is those produced by ADAM. As you can clearly see, our optimizer converges to

a low loss value much quicker than ADAM on the quadratics. This means that in practice, you would need to train using our optimizer for less steps to reach peak accuracy. Therefore, our project was successful in the sense that we were able to create a meta-learning system that outperforms ADAM in certain cases. As for the MNIST data set, our optimizer performs on par with ADAM (to the point where they almost have the same loss curves). While not the result we were hoping for (outperforming ADAM), it is still an impressive feat to be on par with it.

4 Challenges

The most challenging aspect of the entire project was the actual coding of the model itself. It was rather difficult to implement some of the portions of the training of the optimizee model, as we cannot use a standard loss function with it, we needed to obtain the gradients and feed them into the optimizer to get updates.

Furthermore, we had issues writing the loss function for the rnn_optimizer class it was rather complicated. We were eventually able to solve these issues after re-reading the original paper to gain a deeper understanding of the systems at work.

Finally, this project was much more hands on with using/applying gradients, to the point where it required us to turn off eager execution so we could have more flexibility with the computation graphs that were being used to calculate gradients. Essentially, we have to stop gradients on the gradients of the optimizee model parameters that are being fed into the optimizer, and we also need to reset all gradients after making an update to the parameters of the optimizer/optimizee. This made our train method very hard to code and, as can be seen from the code itself, very complicated.

Making these changes to the gradients turned out to be incredibly difficult to do in Tensorflow 2, and we suspect this is likely the reason why all the implementations we found online were in pytorch. However, despite this we were able to create a working model

5 Reflection

We feel that this project ultimately turned out great! We were able to construct the model, and it performed better than ADAM on one of the datasets used and on par with the second. We met all of the goals we assigned for ourselves, with the exception of our stretch goal as we were only able to perform as good as ADAM on MNIST. Our original stretch goal was to outperform it, which we clearly did not.

The model worked out largely in the way that we expected it to. There were some bumps along the road involving architecture/computational aspects of it, but by the end of the project we were able to replicate the results of the original

paper.

Our approach did not really change over the course of the project. The largest change we made was we were originally going to code it in pytorch, but decided on tensorflow (a mistake) as we already knew that library. We mostly adhered to the structure laid out in the original paper, with minor code changes to make it so that we updated parameters only after seeing and calculating loss from 20 examples. There were no significant pivots, this project was what we set out to create!.

If we could do this project over, we would absolutely use pytorch because it allows better access to gradients and computation graphs that would have made this project significantly easier. Furthermore, we would also iterate on the framework in the paper and create an optimizer model that would work on any dataset without needing modifications, as this one does.

If we had more time, we could improve the project by doing more of the experiments that were performed in the paper. The original writers show proof of concept on several other data sets (such as MNIST). It would have been interesting to follow in these experiments ourselves, however due to time constraints this was not possible for us. If we had more time, we could also perform our own experiments on data sets not used in the original paper. This could possibly give us more information about where the model starts to break down and where this form of meta-learning operates the best.

Our biggest takeaway from this project is always take the time to examine and learn libraries that work well with the project you are trying to create! As discussed above, a significant amount of energy was spent trying to get gradients to flow/stop properly using tensorflow when the same was accomplished by a single method call in pytorch. If we had used pytorch, we could have avoided all this frustration. On the bright side, another thing we learned was how to manipulate gradients in a manual way in tensorflow so that you can implement non-standard architectures such as this one.

All things considered, we are happy with how the project turned out and think that it works quite well.

References

[1] Andrychowicz, Marcin et. al. "Learning to Learn by Gradient Descent". Published in the 30th Conference of Neural Information Processing Systems, Barcelona, Spain.
https://papers.nips.cc/paper/2016/file/fb87582825f9d28a8d42c5e5e5e8b23d-Paper.pdf

[2] S. Thrun and L. Pratt. Learning to learn. Springer Science Business Media, 1998.

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

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

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

[6] A. S. Younger, S. Hochreiter, and P. R. Conwell. Meta-learning with back-propagation. In International Joint Conference on Neural Networks, 2001.