Implementation and Convergence Analysis of First-Order Optimization Methods for a Fully-Connected Neural Network

Eric M. Fischer
University of California Los Angeles
Los Angeles, CA 90095
emfischer712@ucla.edu
303 759 361

Abstract

This study presents an in-depth comparative analysis into the Python implementation of 5 first-order optimization methods: stochastic gradient descent (SGD), stochastic gradient descent with momentum (SGD-M), stochastic gradient descent with Nesterov momentum (SGD-NM), root mean square prop (RMSprop), and Adam. We then perform a convergence analysis and discuss why the results confirm the distinguishing characteristics of the first-order methods discussed in the Methods section. Adam and RMSprop exhibit the best convergence and prove to be the best first-order optimization methods for a fully-connected neural network applied to a CIFAR-10 classification task.

1. Introduction

Stochastic gradient-based optimization methods are of critical practical importance to various fields of science and engineering. If an objective function is differentiable with respect to its parameters, gradient descent is a relatively effective optimization method, as evaluating the first-order partial derivatives with respect to the parameters is just as computationally expensive as evaluating the function. Often, however, objective functions are stochastic, composed of a sum of subfunctions evaluated at different data subsamples. In these cases, stochastic gradient descent can improve convergence, taking gradient steps with respect to individual subfunctions. In addition to data subsampling, objective functions can be noisy when they employ optimization techniques such as dropout regularization in neural networks. In this paper, as higher-order optimization methods are ill-suited for high-dimensional parameter spaces such as those of neural networks, we focus on several of the most popular first-order optimization methods [10].

Thus the primary aim of this investigation is to examine the convergence properties of the following first-order optimization methods on a simple, yet practical example of a fully-connected neural network: Stochastic Gradient Descent (**SGD**), Stochastic Gradient Descent with Momentum (**SGD-M**), Stochastic Gradient Descent with Nesterov Momentum (**SGD-NM**), Root Mean Square Prop (**RMSprop**), and **Adam**.

The fully-connected neural network, its objective function, and the first-order optimization methods used for the convergence analysis in this study were implemented first-hand in Python without using advanced programming libraries. This allows us to compare directly the implementational differences between the first-order methods and gives us a reference for discussion.

As the objective function is implemented first-hand, it is of critical importance to perform gradient checks, testing our (analytic) gradient implementation against a correct numerical gradient.

After applying each of the first-order methods on the CIFAR-10 classification task, we analyze their convergence and reason about why the results are sensible given their defining characteristics. We use a neural network and the CIFAR-10 dataset [11] from the University of Toronto, which contains 60,000 images representing 10 classes.

2. Problem

We use a neural network with 3 fully-connected layers and 500 neurons per layer for the basis of this convergence analysis. Although a neural network with just 1 fully-connected layer is a universal approximator i.e. it can approximate any continuous function, shallow neural networks with just 1 or 2 fully-connected layers are uncommon for many complex real-world applications and hence our conclusions would be less practical [5] [16]. We also recognize that neural networks generally have relatively small performance increases after more than 3 fully-connected layers [8] [13] [16]. Also, while more than 3 fully-connected layers may improve network performance,

gradient-based training becomes more difficult as deeper networks are increasingly non-linear [17] [1] [3]. A deeper network would simply make our convergence results more noisy.

We use 500 neurons per layer for two key reasons. First, it is known that the capacity of a neural network, or its ability to represent complex functional relationships, increases with its number of layers and neurons. As we only use 3 fully-connected layers to avoid further non-linearities, a large enough number of neurons should ensure the model has sufficient capacity to express complex relationships [8] [13]. 500 neurons is sufficiently large and is a good representative of a neural network in practice. Second, designs in which hidden layers have an equal number of neurons in general perform as well or better than designs in which neurons in layers form a different schema, e.g. a pyramid-like schema [8] [13] [12]. In this sense as well, we have a good example of a neural network in practice.

3. Data

We use the CIFAR-10 dataset, which consists of 60,000 32x32 pixel color images, with 6,000 images representing each of 10 classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck [11]

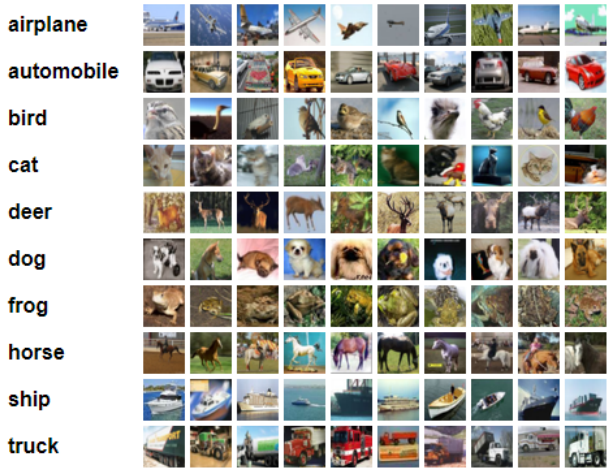

Figure 1. Cifar-10 Class Images [11]

4. Methods

The first-hand Python implementations and key characteristics of the first-order optimization methods Stochastic Gradient Descent (SGD), Stochastic Gradient Descent with Momentum (SGD-M), Stochastic Gradient Descent with Nesterov Momentum (SGD-NM), Root Mean Square Prop (RMSprop), and **Adam** are expounded here.

There exist other ways of performing neural network optimization. For example, Limited-memory BFGS (L-BFGS) is an optimization method in the family of quasi-Newton methods that approximates the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm using limited computer memory [14]. But gradient descent is the most established way to optimize neural network loss functions.

4.1. SGD

Stochastic gradient descent (SGD) is named as such because the method uses randomly selected samples to evaluate the gradient and perform a parameter update. It can thus be regarded as a stochastic approximation of gradient descent optimization [18].

The true definition of SGD is to perform a parameter update after approximating the gradient at just 1 example, but it is more common to approximate the gradient with a mini-batch, or batch, of training examples. Occasionally this is described as Minibatch Gradient Descent or Batch Gradient Descent, but more often the name SGD is still used.

Not only do batches provide more data for a more precise gradient calculation, which lends smoother convergence as the gradient is averaged over more training examples, but SGD exhibits better performance when it can make use of vectorization libraries as opposed to computing each step separately [18]. When we refer to SGD in this study, we refer to SGD with mini-batches of training examples. Namely, we used batches of 100 training examples for the SGD optimization.

SGD updates the parameters, or weights **w** of the neural network, by subtracting the gradient **dw** multiplied by the **learning_rate**. This is to move in the negative direction of the gradient, as the gradient gives the direction of increase and we wish to minimize the loss function, or move in the direction of decrease. The learning rate here is a hyperparameter that is fixed. With a small enough learning rate, SGD is guaranteed to make non-negative progress on the loss function [13].

```
w -= learning_rate * dw
```

Figure 2. Stochastic Gradient Descent

4.2. SGD-M

Stochastic gradient descent with momentum (SGD-M) almost always exhibits faster convergence on deep neural networks than SGD. The key difference compared to SGD is the role of the gradient **dw**. From a physics point of view, in SGD the gradient directly integrates the position. However, in SGD-M the gradient only influences the velocity **v**, which in turn has an effect on the position. In the SGD implementation, this corresponds to **dw** directly affecting **w**, while in this SGD-M implementation **dw** affects **v** which in turn affects **w**.

The parameter vector **w** will thus build up velocity in any direction that has consistent gradient.

```
v = mu * v - learning_rate * dw
w += v
cache['velocity'] = v
```

Figure 3. Stochastic Gradient Descent with Momentum

The velocity **v** is usually initialized as zeros and is stored as a cached value that is updated each iterative parameter update. The term **mu** is commonly referred to as momentum, but a more appropriate name might be friction or damping. This is because the **mu** term, over iterations of the algorithm, damps the velocity of gradient descent to take smaller steps in later stages of learning, until stopping completely. Typical values for momentum include 0.5, 0.9, 0.95, and 0.99, with 0.9 the most common [19] [13] [4].

One can implement a momentum schedule to increase momentum in later stages of learning, similar to an annealing schedule for the learning rate, and this can increase the rate of convergence. For a more direct comparison to other methods, we did not implement a momentum schedule in this study.

4.3. SGD-NM

Stochastic gradient descent with Nesterov momentum (SGD-NM) has strong theoretical convergence guarantees for convex functions and also consistently exhibits better convergence than SGD-M.

First, notice from the SGD-M implementation above that when the parameter vector is at some position **w**, it will be increased by a factor **mu * v**, ignoring the second term in the velocity **v** calculation. The core idea of Nesterov momentum is that we can exploit this fact and compute the gradient at the new "lookahead" position **w + mu * v**, where the momentum step will soon take us, instead of at the current soon-to-be-outdated position **w**, the position before applying a momentum step [20] [13].

This concept is expressed in Python below for conceptual understanding. (Our actual Python implementation of SGD-NM follows.) Here **w_ahead** represents the "lookahead" position, where the momentum step will soon take us. Compared to SGD-M, the calculation for velocity **v** differs in that we use the gradient **dw_ahead** calculated at the "lookahead" position **w_ahead**, instead of the gradient **dw** calculated at the position **w**.

```
w_ahead = w + mu * v
# evaluate dw_ahead (gradient at w_ahead instead of at w)
v = mu * v - learning_rate * dw_ahead
w += v
```

Figure 4. Understanding SGD-NM

Now observe our Python implementation of SGD-NM below, which provides a direct visual comparison with SGD-M. Upon observation, the implementations are identical, yet now we update the parameter values **w** not just by adding **v** but by adding **v + mu * (v - v_prev)**. Accordingly, we first store the previous velocity **v_prev** before calculating the new velocity **v**.

```
v_prev = v
v = mu * v - learning_rate * dw
w += v + mu * (v - v_prev)
cache['velocity'] = v
```

Figure 5. Stochastic Gradient Descent with Nesterov Momentum

In this Python implementation for SGD-NM, the **w_ahead** position and **dw_ahead** gradient are written shorthand as **w** and **dw**. Crucially then, the **w** and **dw** in this SGD-NM implementation are not equivalent to the **w** and **dw** in the SGD-M implementation. The **w** and **dw** here are the parameters and gradients for the "lookahead" position **w + mu * v** [20] [13].

Thus, also in contrast to SGD-M, the velocity **v** that we store in the cache for SGD-NM is the "lookahead" velocity based on the "lookahead" gradient **dw_ahead** (here just named **dw**), and is not equivalent to the current velocity **v** based on the current gradient used in SGD-M. Again, this is despite using the same variable name **v** that was used in the SGD-M implementation.

4.4. RMSprop

Interestingly, Root Mean Square Prop (RMSprop) is an unpublished adaptive learning rate method from Geoffrey Hinton at the University of Toronto [9]. It is based on Adagrad, which for understanding is shown in Python below.

The key to per-parameter adaptive learning rate methods such as Adagrad and RMSprop is annealing the learning rate. In neural networks, this allows gradient descent to settle in deeper but more narrow areas of the loss function that the parameter vector **w** may bounce over with a constant learning rate.

The rate at which to decay the learning rate is the subject of much research. In this study we implement RMSprop, an advancement of Adagrad, to consider a popular adaptive learning rate method. As RMSprop is an extension of Adagrad, we first consider Adagrad.

```
beta += dw**2
w -= learning_rate * dw / (np.sqrt(beta) + eps)
cache['beta'] = beta
```

Figure 6. Adagrad

Adagrad uses **beta** to keep track of the per-parameter sum of squared gradients, which is then used to normalize element-wise the parameter **w** update step. The smoothing term **eps**, usually set from roughly 1e-4 to 1e-8, is only to avoid division by 0 [6].

In Adagrad, parameters **w** with large gradients **dw** will have their effective learning rates decreased, while parameters with small gradients, i.e. sparser parameters with sparser gradients, will have their effective learning rates increased. Thus Adagrad often exhibits improved convergence compared to various forms of **SGD** in settings where data is sparse and sparse parameters are more informative, such as in image recognition and natural language processing [2] [13].

A disadvantage of Adagrad with respect to neural networks is that the monotonic learning rate usually proves too aggressive, and hence learning is stopped early due to a lack of sufficient decrease in the objective function. Solving this problem was the motivation for the RMSprop method, which caches under the variable name **beta** an exponentially-weighted moving average of squared gradients, instead of just the sum of squared gradients used in Adagrad.

```
beta = decay_rate * cache + (1 - decay_rate) * dw**2
w -= learning_rate * dw / (np.sqrt(beta) + eps)
cache['beta'] = beta
```

Figure 7. RMSprop

The variable **decay_rate** is the hyperparameter used for annealing the learning rate, and typical values are 0.9, 0.99, and 0.999.

A moving average of the second moments of gradients, **beta** in RMSprop can be described as leaky, slowly "forgetting" gradient values **dw** from many iterations ago. As RMSprop is more sensitive to recent gradients, its learning rate will anneal more responsively throughout the convergence process. This in turn causes the RMSprop per-parameter updates to gradually get smaller, thereby preventing an early stop in learning.

Hence, RMSprop like Adagrad adjusts per-parameter learning rates based on the magnitude of its gradients, lending an equalizing effect, but unlike Adagrad the parameter updates gradually decrease to prevent stopping too early [13] [15].

4.5. Adam

Adam, whose name is derived from adaptive moment estimation, exploits both the ideas of adaptive gradients and momentum. It combines the advantages of Adagrad, which works well with sparse gradients, and RMSprop, which works well in on-line and non-stationary settings. It is thus well-suited for problems with noisy and/or sparse gradients and is also appropriate for non-stationary objectives. The hyperparameters, although more numerous than other first-order methods, also require little tuning [10].

Additionally, the magnitude of parameter updates is invariant to rescaling of the gradient, step sizes are approximately bounded by the step size hyperparameter, and it naturally performs step size annealing [10].

```
m = beta1*m + (1-beta1)*dw
v = beta2*v + (1-beta2)*(dw**2)
w -= learning_rate * m / (np.sqrt(v) + eps)
cache['m'] = m
cache['v'] = v
```

Figure 8. Understanding Adam

Adam computes individual adaptive per-parameter learning rates based on moving average estimates of the first and second moments of the gradients, **m** and **v**, respectively. Note that the update for **w** is identical to the RMSprop update, except that **m**, a smoothed version of the first moment of the gradient, is used instead of the raw and perhaps noisy gradient vector **dw** [13].

The variable **eps**, similar to RMSProp, is included to avoid division by 0. The variables **beta1** and **beta2** are decay rates for moving averages of the first and second moments of the gradients, respectively. The Adam paper recommends parameter values **eps** = 1e-8, **beta1** = 0.9, and **beta2** = 0.999 [10].

The full Adam update is shown below. It includes a bias correction mechanism for initialization.

```
t += 1
m = beta1*m + (1-beta1)*dw
v = beta2*v + (1-beta2)*(dw**2)
mt = m / (1-beta1**t)
vt = v / (1-beta2**t)
w -= learning_rate * mt / (np.sqrt(vt) + eps)
cache['m'] = m
cache['v'] = v
cache['t'] = t
```

Figure 9. Adam

The bias correction mechanism compensates for the fact that **m** and **v** do not convey as useful information in the first steps after initialization. They could be for example biased toward zero due to zero initialization. This explains the caching of iteration **t** in addition to the moving averages **m** and **v** of the first and second moments. Adam uses **t** in the revised calculations for the moving averages **mt** and **vt** to appropriately scale them over iterations of the algorithm [10].

5. Evaluation

In this study, we performed two general kinds of evaluation. We of course evaluated the convergence of the methods, but before this we used a gradient check to evaluate whether the first-hand Python implementation of the objective function was correct.

5.1. Convergence Analysis

To evaluate the convergence of the methods, it is useful to compare SGD, SGD-M, and SGD-NM in one plot of the training loss over iterations. For these 3 methods, we also show in two separate plots the training and validation accuracies over epochs. Then we repeat the same plots and display all 5 methods: SGD, SGD-M, SGD-NM, RMSprop, Adam.

5.2. Gradient Checks

Now we elaborate on some guidelines for gradient checking with neural networks. It was critical to test our analytic gradient implementation against a numerical gradient.

In general, analytic gradients are used in practice because they are exact calculations and inexpensive to compute. This is in contrast to numerical gradients, which are simple to implement but approximate calculations and expensive to compute. Analytic implementations are more error-prone, however, so it is common to perform gradient checks as we did, comparing the implemented analytic gradient to the numerical gradient [13] [8].

In comparing them, we use relative error:

$$\frac{|\nabla f_a - \nabla f_n|}{max(|\nabla f_a|, |\nabla f_n|)}$$

or the ratio of the absolute-value difference between the analytic gradient ∇f_a and numerical gradient ∇f_b over the maximum of the absolute values of both gradients. Using relative error for the comparison, as opposed to just absolute-value difference, is necessary as the gradient could be close to 1 or 1e-5, which would merit different error thresholds in determining whether the absolute-value difference is acceptable. Also, normally the relative error formula only includes ∇f_a or ∇f_b in the denominator but not both. By performing the max operation in the denominator, we prevent dividing by 0 which is often the case with ReLU activation functions commonly used in neural networks and in this study [13].

As a rule of thumb, relative errors greater than 1e-2 usually indicate the implemented analytic gradient is problematic, and errors of 1e-7 or less indicate it is correct. If there are non-differentiable kinks in the neural network objective functions (e.g. the use of tanh nonlinearities or the Softmax function), then a relative error of 1e-4 or less is generally acceptable. Also, as errors build up more in deeper networks e.g. a 10-layer network, a relative error of 1e-2 or less may be acceptable. So the depth of the network must be considered as well. These are empirical guidelines drawn from experiments performed by various authors [8].

We also use double precision floating point calculations and use gradient checks as a way of ensuring we are in the active range of floating point operations [7]. We use double precision floating point because even with correct gradient implementations, we could have errors as high as 1e-2 using single precision floating point, when the true error is closer to 1e-8 [13].

To ensure we are in the active range of floating point operations, during training we print the raw numerical and analytic gradients and ensure these numbers are not extremely small, in the area of 1e-10. This is often overlooked when optimizing neural networks with gradient descent. If the gradient values are too small, we can scale the objective function by a constant to increase their values to a range in which floats are more dense, ideally on the order of 1.0 in which the float exponent is 0 [7].

Lastly, we make sure to perform gradient checks during characteristic modes of operation, after a "burn-in" time for the network in which gradient values could be significantly skewed by weight initializations [8].

6. Results

To reduce noise in the convergence graphs, a subsample of 4,000 images was used from the CIFAR-10 dataset. Hence an epoch in these plots represents one run through this subsampled dataset. As we run 10 epochs of training with batch sizes of 100 samples, the plots below of the training loss are accordingly over 400 iterations of training, i.e. 400 parameter updates for each of the first-order methods.

First we show the training loss for SGD, SGD-M, and SGD-NM over iterations of gradient descent.

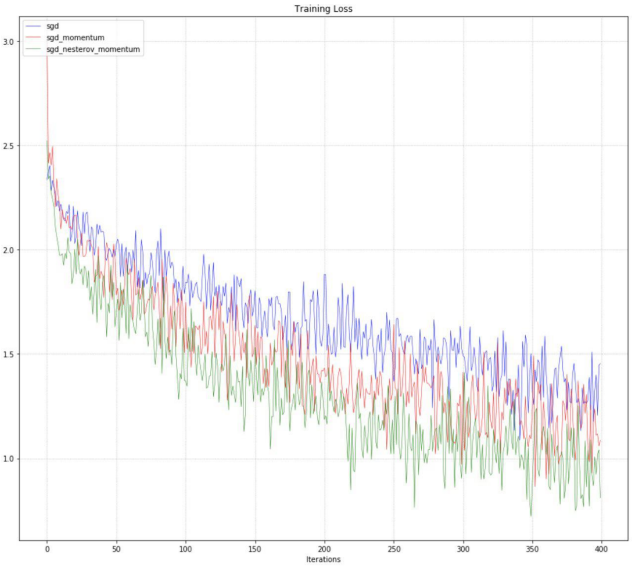

Figure 10. Training Loss for SGD (blue), SGD-M (red), SGD-NM (green) over 400 iterations

And these are the training and validation accuracies for SGD, SGD-M, and SGD-NM over epochs.

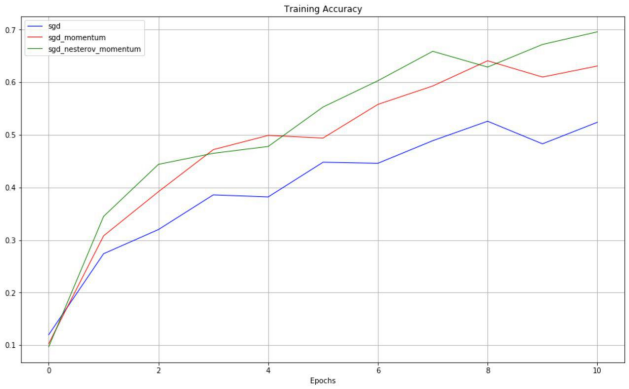

Figure 11. Training Accuracy for SGD (blue), SGD-M (red), SGD-NM (green) over 10 Epochs

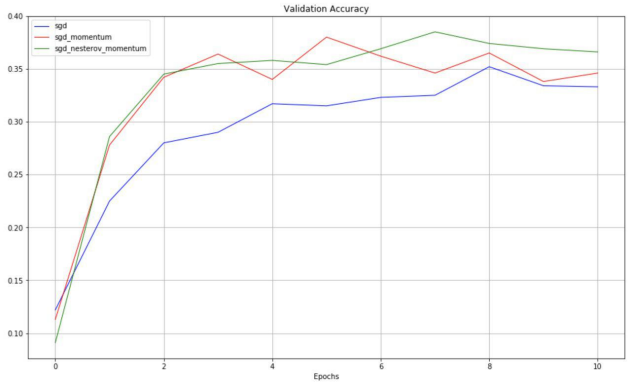

Figure 12. Validation Accuracy for SGD (blue), SGD-M (red), SGD-NM (green) over 10 Epochs

Next we can observe the training loss for SGD, SGD-M, SGD-NM, RMSprop, and Adam over iterations of gradient descent.

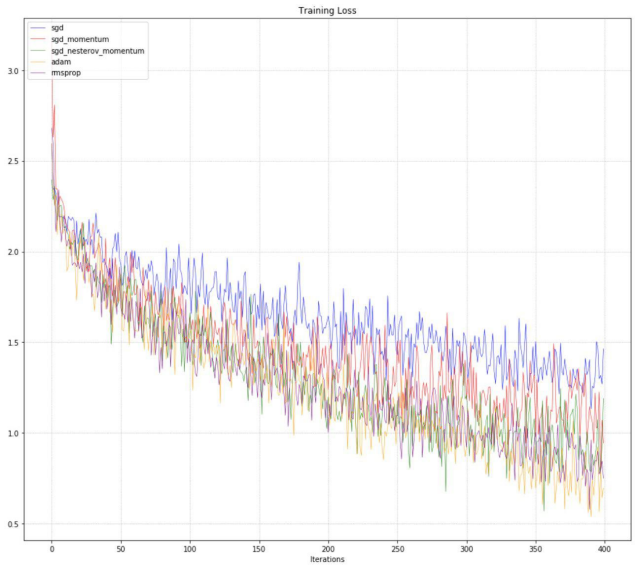

Figure 13. Training Loss for SGD (blue), SGD-M (red), SGD-NM (green), RMSprop (purple), and Adam (orange) over 400 iterations

And these are the training and validation accuracies for SGD, SGD-M, and SGD-NM over epochs.

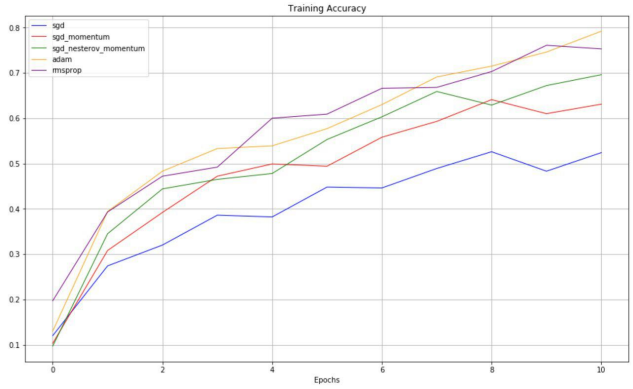

Figure 14. Training Accuracy for SGD (blue), SGD-M (red), SGD-NM (green), RMSprop (purple), and Adam (orange) over 10 Epochs

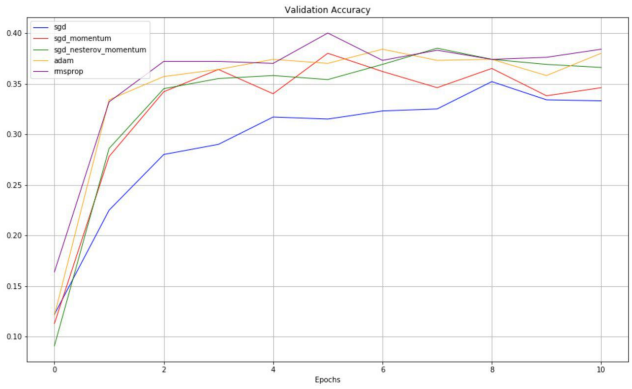

Figure 15. Validation Accuracy for SGD (blue), SGD-M (red), SGD-NM (green), RMSprop (purple), and Adam (orange) over 10 Epochs

7. Discussion

As expected between SGD, SGD-M, and SGD-NM, SGD makes the slowest progress in minimizing the objective function value, or the training loss, over iterations. And SGD-NM exhibits improved convergence over SGD-M, as it only improves SGD-M with the key notion that the gradient **dw** is computed at the "lookahead" position of the parameters **w**, where the momentum step will soon take us.

Hence further confirming the literature that SGD-NM consistently exhibits better convergence than SGD-M, we observe SGD-NM with a lower training loss than SGD-M after 400 training iterations and throughout most of the training history. We can also observe a much quicker initial descent for SGD-NM, due to the relatively large steps it takes in the early iterations of training compared to other methods. This is because, in early compared to later iterations, the gradient **dw** at the "lookahead" position will be even more different than the gradient **w** at the current position, leading to even larger gradient steps for SGD-NM, relative to other methods, than for the rest of the training process.

The training and validation accuracies of SGD, SGD-M, and SGD-NM follow suit, with SGD having the worst training and validation accuracy and SGD-NM having the best training and validation accuracy after 10 epochs of training.

We can also observe that RMSprop and Adam tend to perform slightly better than SGD-NM on this specific neural network problem, with Adam (orange) exhibiting the best convergence, i.e. the lowest training loss, after 400 iterations. We can see the training loss of the orange line falling steadily below RMSprop (purple) and SGD-NM (green), especially toward the later stages of training.

In general, we discussed how RMSprop and Adam prevent stopping learning too early due to their adaptive learning rates. By annealing the learning rate, they can settle in deeper but more narrow areas of the loss function that the parameter vector **w** may skip over with a more aggressive learning rate. This is exactly what we observe. In the later stages of training, the lines of RMSProp (purple) and Adam (orange) both appear to "pull away" even more from SGD-NM, until it is clear they have the least training loss with Adam having the least.

The training and validation accuracies of RMSprop and Adam also follow suit, with RMSProp and Adam very near each other and definitively greater than SGD-NM. Overall Adam reduced the training loss most, but RMSprop is an equally viable contender for this neural network problem and perhaps the best choice, as the validation accuracy is the single most important metric for a machine learning model and RMSprop had the highest validation accuracy.

Training loss over iterations is how we demonstrate convergence, and training accuracy lends a helpful plot, but validation accuracy is a stand-alone metric for determining model quality. RMSprop exhibited comparable training loss and the best validation accuracy after 10 epochs and throughout most of the training history, so it presents the strongest case for the most optimal first-order optimization method for this neural network problem.

8. Conclusion

We have presented an in-depth comparative analysis into the Python implementation of 5 first-order optimization methods: stochastic gradient descent (SGD), stochastic gradient descent with momentum (SGD-M), stochastic gradient descent with Nesterov momentum (SGD-NM), root mean square prop (RMSprop), and Adam.

We then displayed their convergence over 10 epochs of the training data and observed that the results confirm the distinguishing characteristics of the first-order methods discussed in the Methods section.

Importantly, our experiments reflect the mathematical conclusions made for convex problems about the rate of convergence of these first-order methods.

Adam and RMSprop exhibit the best convergence and prove to be the best first-order optimization methods for a fully-connected neural network applied to a CIFAR-10 classification task.

References

[1] Lei Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? *Neural Information Processing Systems*, 2014. https://arxiv.org/pdf/1312.6184.pdf.

[2] Amir Beck. *First-Order Methods in Optimization*. MOS-SIAM Series on Optimization. MOS-SIAM, 2017. https://doi.org/10.1137/1.9781611974997.

[3] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. *arXiv*, September 2012. https://arxiv.org/pdf/1206.5533v2.pdf.

[4] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004. http://stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf.

[5] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 1989. http://www.dartmouth.edu/~gvc/Cybenko_MCSS.pdf.

[6] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, July 2011. http://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf.

[7] David Goldberg. What every computer scientist should know about floating-point arithmetic. *Association for Computing Machinery, Inc.*, March 1991. https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[9] Geoffrey Hinton. Neural networks for machine learning. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[10] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2015. https://arxiv.org/pdf/1412.6980.pdf.

[11] Alex Krizhevsky. The cifar-10 dataset. https://www.cs.toronto.edu/~kriz/cifar.html.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. http://www.cs.toronto.edu/~fritz/absps/imagenet.pdf.

[13] Fei-Fei Li, Justin Johnson, and Serena Yeung. Cs 231n: Convolutional neural networks for visual recognition. http://cs231n.stanford.edu/.

[14] Limited Memory BFGS. Limited memory bfgs — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Limited-memory_BFGS.

[15] Yuri Nesterov. *Lectures on Convex Optimization*, volume 137 of *Springer Optimization and Its Applications*. Springer, 2 edition, 2003. https://link.springer.com/book/10.1007.

[16] Michael Nielsen. A visual proof that neural nets can compute any function, October 2018. http://neuralnetworksanddeeplearning.com/chap4.html.

[17] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *International Conference on Learning Representations*, 2015. https://arxiv.org/pdf/1412.6550.pdf.

[18] Stochastic Gradient Descent. Stochastic gradient descent — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

[19] Lieven Vandenberghe. Ece236b - convex optimization. http://www.seas.ucla.edu/~vandenbe/ee236b/ee236b.html.

[20] Lieven Vandenberghe. Ece236c - optimization methods for large-scale systems. http://www.seas.ucla.edu/~vandenbe/ee236c.html.