1 Optimizer Descriptions

1.1 Stochastic Gradient Descent (SGD)

- This is the most common gradient optimization technique on which most other gradient optimization algorithms attempt to improve.
- SGD utilizes a stochastic approximation of the loss across the entire dataset to compute the gradient across a single data example or mini batch. This tends to accelerate learning in the earlier parts of training during which significantly fewer examples can give a reasonable approximation of the proper gradient direction.
- Although vanilla gradient descent works well, it has a tendency to converge slowly and get stuck on saddle points or local minima in the optimization landscape.

Update Rule

$$\theta_{t+1} = \theta_t + \Delta\theta$$
$$\Delta\theta = -\alpha \bigtriangledown_\theta L(\theta)$$

- *Variables:* θ (model parameters), α (step size), L (loss)

Pros and Cons

- Simple, computation is fast.
- Tends to get stuck on local minima/saddle points and converge slowly.
- Requires the setting of a learning rate. Performance is dependent on a proper learning rate.
- The variance of parameter updates can be quite high if the batch size is not chosen properly (i.e., batch size is too small).

1.2 SGD + Momentum

- One of the simplest and most effective improvements upon vanilla SGD.
- Creates separate learning rates for each parameter that is being optimized (i.e., per-feature learning rates) by keeping track of previous parameter updates inside of a "momentum" term.
- Tries to avoid issues that SGD has with getting stuck or converging slowly to an optimum. It is especially effective in improving training speed in a landscape where gradients are significantly higher in some dimensions than others.

- Tries to increase the step size in dimensions where the parameter update is continually in the same direction (i.e., smooth and flat areas of the optimization landscape), while decreasing the step size in dimensions where the gradient is continually changing (i.e., areas of rapid change or high gradients in the optimization landscape).

Update Rule

$$\theta_{t+1} = \theta_t - m_t$$
$$m_t = \gamma m_{t-1} + n \bigtriangledown_\theta L(\theta)$$

- *Variables:* m_t (momentum term), γ (momentum/decay term)

Pros and Cons

- Tends to improve convergence speed and performance compared to vanilla SGD.
- Introduces an additionally hyperparameter γ that must be tuned.
- Performance tends to be good with $\gamma = 0.9$ (i.e., performance is less sensitive to the setting of γ than it is to the setting of α), but recent research is hinting that such a default setting can be improved depending on the application.
- Performance is still dependent on the learning rate.

1.3 AdaGrad

- Attempts to assign larger step sizes to model parameters that have smaller updates or are updated less frequently and vice versa.
- Scales the update for each parameter by the inverse of the root sum of past squared gradients at that parameter (i.e., per-feature step sizes using past gradient information).
- The update rule effectively creates a learning rate annealing effect because the sum of previous squared gradients monotonically increases for each parameter throughout training (this is not strictly monotonic because the gradient may be zero for an iteration).

Update Rule

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} \bigtriangledown_\theta L(\theta)$$
$$G_{t,ii} = \sum_t g_{t,i}^2 = \sum_t (\bigtriangledown_{\theta_{t,i}} L(\theta))^2$$

- *Variables:* G_t (diagonal matrix, stores sum of squared previous gradients for each parameter), ϵ (small additive term for numerical stability in denominator), $g_{t,i}$ (gradient w.r.t. a single parameter i at iteration t)

Pros and Cons

- Has a learning rate annealing effect that is created by the monotonically increasing term in the denominator of the update term.
- Performance is often very dependent on learning rate. If learning rate is too low, update rule will tend to decrease too quickly and cause training to become very slow or even halt.
- Tends to be sensitive to initial optimization conditions. If a parameter begins training with very large gradients, the updates to this parameter will tend to be very small for the rest of training because the denominator of the update rule is a sum of all previous squared gradients.
- The progress of updates along each dimension tends to be balanced over time because of the nature of the update rule. This is useful because earlier layers of models (e.g., deep neural networks) tend to have gradients of significantly smaller magnitude than later layers. This update rule can combat such an issue by using past gradient information to balance the step sizes across different layers of the network.

1.4 RMSProp

- One method for resolving the issue of learning rates decreasing too quickly in AdaGrad.
- The general approach is similar to AdaGrad but is slightly modified.
- Instead of accumulating all previous squared gradients in the denominator of the update term, a window of previous squared gradients is maintained using an exponentially decaying average. The exponentially decaying average is an approximation of the window of previous squared gradients, which is used because it is simpler than maintaining a list of k previous gradients.
- The exponentially decaying average of previous squared gradients is not monotonically increasing (i.e., because previous squared gradients will fall out of the window over time as others are added). Therefore, it avoids the issue of diminishing learning rates seen in AdaGrad.

Update Rule

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{E[g^2]_t + \epsilon}} \bigtriangledown_\theta L(\theta)$$

$$E[g^2]_{t+1} = \gamma E[g^2]_t + (1-\gamma)(\bigtriangledown_\theta L(\theta)_{t+1})^2$$

- *Variables:* $E[g^2]$ (exponentially decaying average of previous squared gradients), γ (decay constant)

Pros and Cons

- Have to set value for the extra decay term.
- Performance is still dependent upon the value of the learning rate.
- Eliminates the issue of diminishing learning rates by using the exponentially decaying average instead of a sum of previous squared gradients.
- Identical to AdaDelta update, but without the extra term in the numerator (see below).

1.5 AdaDelta

- Another method that resolves the issue of diminishing learning rates in AdaGrad. Similar to RMSProp but adds another term into the numerator of the update rule.
- Again utilizes exponentially decaying averages to avoid the problem of quickly decreasing step sizes during training. The exponentially decaying average again approximates the observation of a fixed window of past gradients.
- Eliminates the learning rate from the update term.

Update Rule

$$\theta_{t+1} = \theta_t - \frac{\sqrt{E[\Delta\theta^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} \bigtriangledown_\theta L(\theta)$$

$$E[\Delta\theta^2]_{t+1} = \gamma E[\Delta\theta^2]_t + (1-\gamma)\Delta\theta^2_{t+1}$$

- *Variables:* $\Delta\theta_t$ (parameter update at iteration t), $E[\Delta\theta^2]$ (exponentially decaying average of past squared parameter updates)
- The numerator term depends on the squared parameter updates up to the previous iteration because the update for the current iteration is not known before the update is computed/applied. During the first iteration, $E[\Delta\theta^2]_0$ is zero because no previous iterations exist.

Pros and Cons

- Again, must determine the proper value for the decay term, γ. However, performance tends to be less sensitive to this hyperparameter compared to the learning rate, α.
- Does not require a learning rate to be set. This method is, therefore, less sensitive to the setting of hyperparameters compared to other methods (i.e., α tends to have the largest effect on performance for many optimization algorithms).
- May perform poorly compared to other methods despite lack of sensitivity to hyperparameters (i.e., performance may be consistent across hyperparameters but worse than other methods with a good learning rate).

1.6 Adam

- Utilizes estimates of the first and second moments (i.e., the mean and uncentered variance) of gradients to properly scale the step size for each parameter.
- One of the most common choices for optimizers in current deep learning applications. Tends to achieve good results and converge quickly compared to other methods.

Update Rule

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \bigtriangledown_\theta L(\theta)_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\bigtriangledown_\theta L(\theta)_t)^2$$

- *Variables:* m_t (first moment term), v_t (second moment term), β_1 (decay term for first moment), β_2 (decay term for the second constant)
- The m_t and v_t exponentially decaying average terms tend to be biased towards zero (especially during the first few iterations of training) because m_{t-1} and v_{t-1} are initialized to zero. Therefore, an extra "de-biasing" step must be performed in the update to solve this issue.

$$m'_t = \frac{m_t}{1 - \beta_1^t}$$

$$v'_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{v'_t} + \epsilon} m'_t$$

Pros and Cons

- Combines many benefits that are achieved with RMSProp (i.e., replicated in the v_t term) and Momentum (i.e., replicated in the m_t term).
- Achieves fast convergence and good performance with all first-order information, which makes the method computationally efficient/tractable compared to other, second-order approaches (e.g., Newton's method).
- Recent research has shown that Adam may lead to poor generalization performance compared to SGD + momentum with carefully tuned hyperparameters.
- Utilizes many hyperparameters in the update rule compared to other methods. However, the performance of Adam tends to be less sensitive to these hyperparameters (i.e., α is still the most important hyperparameter, while β_1 and β_2 are typically set to 0.9 and 0.999).

2 Convex Optimization

As an example of the optimization algorithms applied to a convex optimization problem, I utilized a logistic regression model to perform classification on the MNIST dataset (found at http://yann.lecun.com/exdb/mnist/). This logistic regression model flattened each input image into a single vector and outputted a probability distribution over the ten possible output classes (i.e., each possible digit). Cross Entropy loss was used during training. The hyperparameters for each of the possible optimization methods were tuned using grid search to yield the following settings:

Hyperparameter Settings

- **SGD**: $\alpha = 0.01$
- **SGD + Momentum**: $\alpha = 0.01$, $\gamma = 0.9$
- **RMSProp**: $\alpha = 0.001$, $\gamma = 0.9$, $\epsilon =$ 1e-8
- **AdaGrad**: $\alpha = 0.1$, $\epsilon =$ 1e-8
- **AdaDelta**: $\gamma = 0.9$, $\epsilon =$ 1e-8
- **Adam**: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon =$ 1e-8

Observations and Discussion

- Vanilla SGD performs consistently worse than other methods w.r.t. training loss and validation accuracy, while AdaGrad (labeled as "ADAG") tends to have the best performance.
- The addition of momentum to SGD clearly creates faster convergence and higher validation performance.

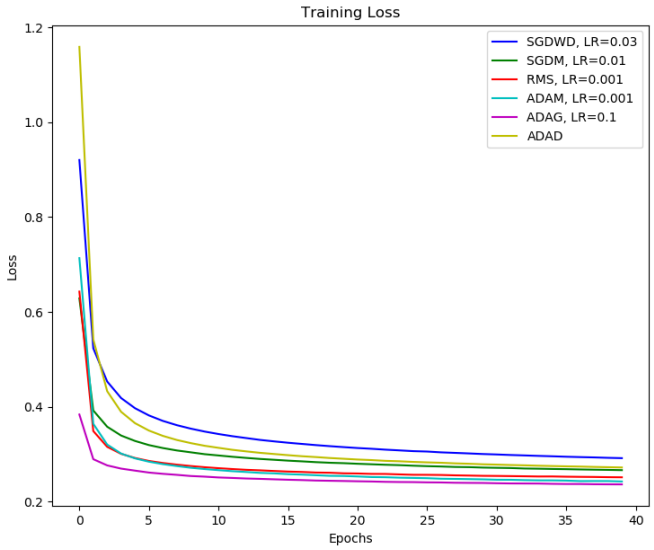

Figure 1: Loss values at each epoch for each of the possible optimization methods throughout training of the logistic regression model on MNIST.

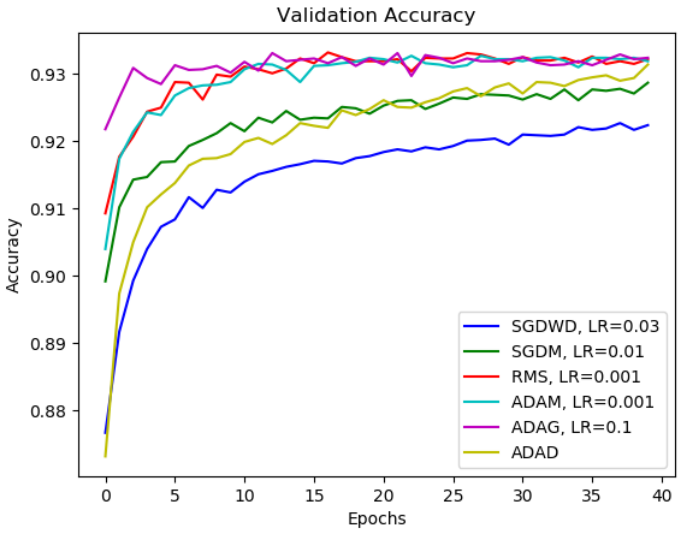

Figure 2: Validation accuracy measures at each epoch for each of the possible optimization methods throughout training of the logistic regression model on MNIST.

- Adam, RMSProp, and AdaGrad seem to converge more quickly than all other methods w.r.t. both loss and validation accuracy.
- AdaDelta (listed as "ADAD" in the figures) is quite competitive despite having very few hyperparameters to tune. However, it does seem to converge more slowly than other, high-performing methods.
- This optimization problem is convex and, therefore, quite different from many current, highly non-convex applications of these algorithms (e.g., deep neural network optimization). Therefore, other, non-convex problems are also useful in gaining a better understanding of the performance of these optimization methods.

3 Non-Convex Optimization

3.1 MNIST Classification

In this experiment, a convolutional neural network (CNN) was applied to the MNIST dataset for image classification. In each of the experiments, I utilized the PyTorch implementation of the ResNet18 architecture (training was performed using Google Colab). I trained each network from scratch (i.e., I did not initialize with pre-trained weights). Aside from this change of model, all settings for this experiment were identical to Section 2 (i.e., cross entropy loss, same data, hyperparameters were selected with grid search). Because the training of neural networks presents a more difficult, non-convex optimization landscape, it alloqa different properties and behaviors of each of the optimization algorithms to be observed (i.e., beyond the simple performance metrics presented in Section 2).

Hyperparameter Settings

- **SGD**: $\alpha = 0.1$
- **SGD + Momentum**: $\alpha = 0.01$, $\gamma = 0.9$
- **RMSProp**: $\alpha = 0.001$, $\gamma = 0.9$, $\epsilon =$ 1e-8
- **AdaGrad**: $\alpha = 0.1$, $\epsilon =$ 1e-8
- **AdaDelta**: $\gamma = 0.9$, $\epsilon =$ 1e-8
- **Adam**: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.99$, $\epsilon =$ 1e-8

Observation and Discussion

- Vanilla SGD again leads to the poorest performance compared to all other methods.

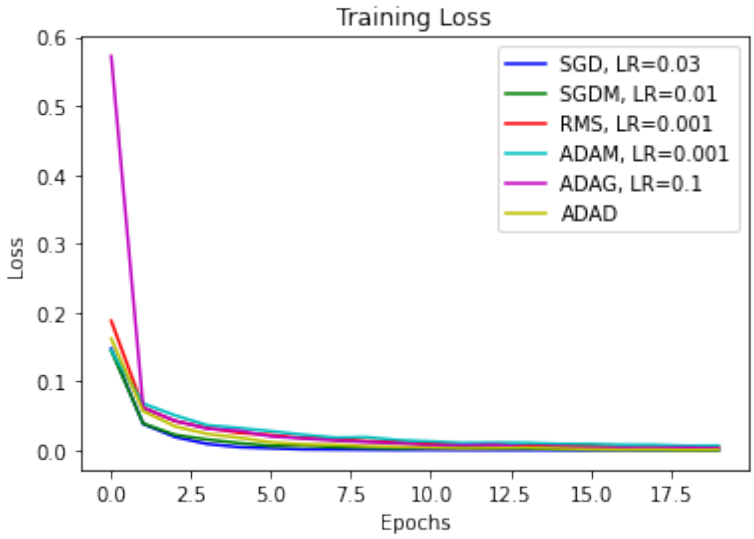

Figure 3: Loss values for CNN optimization on MNIST for all possible optimization algorithms.

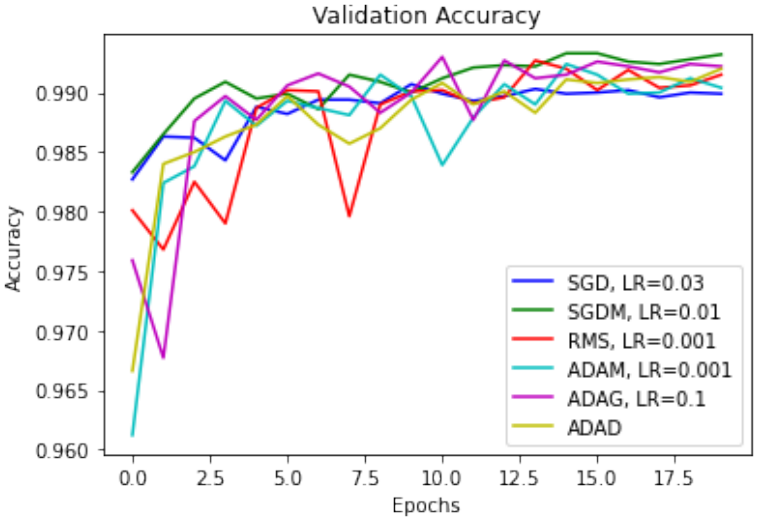

Figure 4: Validation accuracy values for CNN optimization on MNIST for all possible optimization algorithms.

- Many of the optimal hyperparameters that were chosen for these tests were quite similar to the optimal hyperparameters found in Section 2.
- As seen in Fig. 4, the validation accuracy seems to fluctuate more than when the logistic regression model was used but is significantly higher.
- SGD with Momentum results in the highest performing model and has a very stable convergence as seen in Fig. 4. Adam and AdaDelta optimizers also seem to lead to stable, high-performing convergence.
- The performance of many of the different optimization methods seems to be quite similar in both loss and validation accuracy, thus hinting that the MNIST classification problem is not difficult to solve with a CNN. Therefore, CIFAR-10 (i.e., a more difficult image classification dataset) was used to further study the different optimizers.

3.2 CIFAR-10 Classification

In this experiment, I again trained a ResNet18 from scratch (i.e., PyTorch implementation with no pre-training) using each of the possible optimization algorithms. However, in this experiment, I utilized the CIFAR-10 dataset, which contains 10 disjoint classes of small RGB images. I utilized the same hyperparameters presented in Section 3.1. However, I did increase the number of epochs used for training to account for the increased difficulty of the classification problem. Classification on the CIFAR-10 dataset made the differences in methods of optimization become more noticeable, allowing a more accurate comparison between each of the algorithms to be made.

It should be noted that no extra tricks were used to improve optimization in this initial experiment with CIFAR-10 classification (e.g., learning rate annealing, momentum decay, etc.). Learning rate were again selected with grid search. These results are further improved in Section 3.3 with some extra optimization tricks.

Observation and Discussion

- As seen in Fig. 5, vanilla SGD converges quite quickly w.r.t. loss, but has a very poor final validation accuracy.
- Each of the optimization methods seem to eventually converge to similar loss values, as seen in Fig. 5. However, the fastest convergence seems to come with AdaDelta. RMSprop and Adam also converge quite quickly (but slower than AdaDelta).
- Interestingly, AdaGrad seems to converge the slowest of all methods w.r.t. loss. This observation demonstrates the fact that AdaGrad is sensitive to the initial conditions of training. If the first gradient values during training are large, learning will be slowed down for the entire optimization process. Additionally, learning becomes increasingly slow as the L2-norm

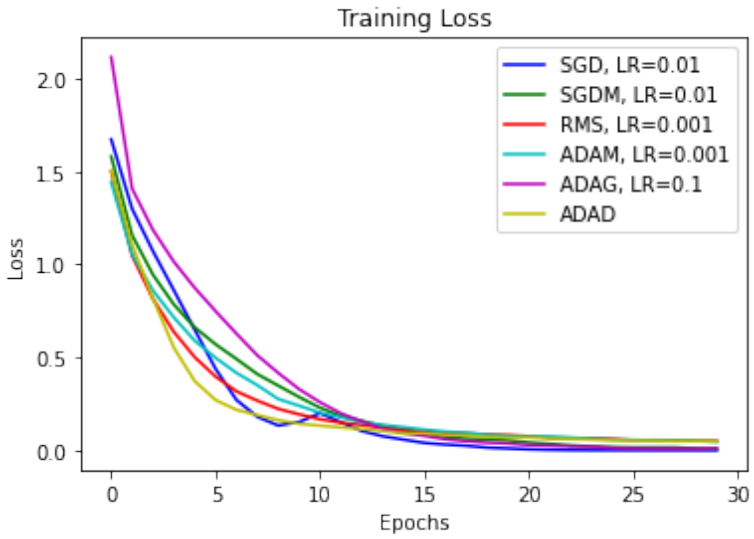

Figure 5: Training losses across epochs for optimization on CIFAR-10 with each optimization algorithm.

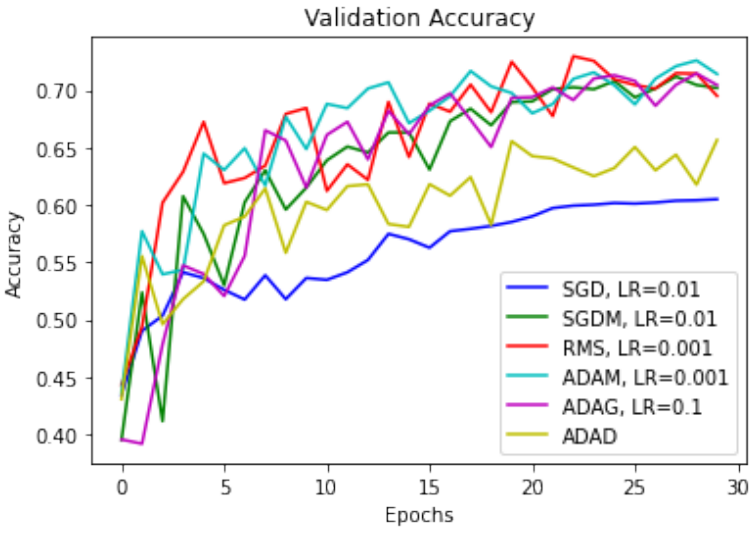

Figure 6: Validation accuracies across epochs on CIFAR-10 with each optimization algorithm.

of previous gradients increases and inversely scales the learning rate of each parameter.

- AdaDelta converges to a lower validation accuracy compared to the rest of the optimization algorithms.
- Adam seems to yield the highest final validation performance, as seen in Fig. 6. However, RMSprop, SGD with Momentum, and AdaGrad are not far behind in terms of final validation accuracy.
- It can be seen in Fig. 6 that SGD with Momentum initially has very large fluctuations in validation accuracy, but eventually (i.e., around epoch 8) has the most stable convergence in terms of validation accuracy. The curve fluctuates significantly in the first few epochs, but eventually becomes smooth and stops having rapid fluctuations, thus highlighting the effect of the momentum term in creating a stable convergence.

3.3 Extra Optimization Tricks

In this section, I continue classification of CIFAR-10 images with the ResNet18 without pre-training. However, I employ extra tricks within the optimization process to improve the final performance of resulting models. I use both SGD with Momentum and Adam (i.e., the most commonly used optimizers in practice) to see how much I can improve the final performance of the CNN compared to the results seen in Sec. 3.2

Learning Rate Finder

In this section, I will use a learning rate finder to select the optimal learning rate to use during training. This method works by employing the following procedure:

- Obtain the model and optimizer being used for the problem in question (e.g., ResNet18 and an Adam optimizer).
- Select a single batch of data from your dataset.
- Create a sorted list of possible learning rates (in ascending order) that is evenly spaced from low to high learning rates (e.g., 1e-5 to 1.0).
- Cycle through each of the learning rates (from low to high). At each learning rate, obtain the loss over your batch of data and perform an update with the current learning rate in the list. This is repeated for every learning rate in the sorted list of learning rates.
- Plot the loss that was obtained at each possible learning rate as a function of the associated learning rate.

- Select the learning rate for which the loss curve has the most negative slope. **It should be noted that, in general, more preference should be given to stable, negative slopes (i.e., as opposed to negative slopes that occur in areas of drastic fluctuation in the loss curve).**

This process allows an exact learning rate to be selected without the use of an expensive grid search. Additionally, the learning rate finder allows the learning rate to be selected with more precision because it identifies the optimal learning rate exactly instead of running tests with learning rates at different orders of magnitude. Every time I use a learning rate finder, I typically like to train the model for about 30-40 iterations with a conservative learning rate (i.e., on the lower side of reasonable learning rates) before running the learning rate finder. I have found that running these few iterations of training prior to running the learning rate finder (i.e., as opposed to using a model with randomly initialized weights) makes the resulting graph significantly less noisy and easier to read.

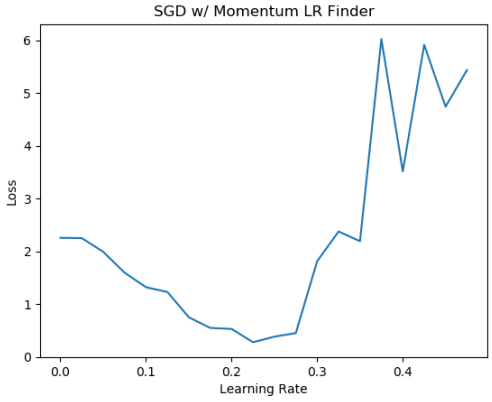

Figure 7: The learning rate finder plot that results when applied to the SGD with Momentum optimizer on the CIFAR-10 dataset.

Before the learning rate finder was employed, learning rates of 0.01 and 0.001 were used for the SGD with Momentum and Adam optimizers, respectively. As can be seen in Fig. 7, the optimal learning rate for SGD with Momentum occurs around 0.05. This is known because it is around a learning rate of 0.05 that the associated graph has its most negative slope (i.e., on the far left side of the graph). Similarly, the optimal learning rate for the Adam optimizer, according to Fig. 8, occurs around 0.008 (i.e., anything between 0.005 and 0.015 seems reasonable). It can be seen that these learning rates are quite close to those obtained with grid search (a bit higher), but were obtained within minutes as opposed to the significant time taken to perform a grid search.

As can be seen in Fig. 9 and Fig. 10, the performance of models trained with the learning rates selected from the learning rate finder is identical, if not slightly better, than models trained with learning rates selected using grid

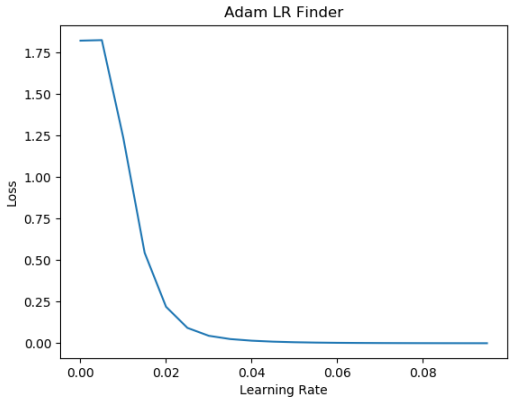

Figure 8: The learning rate finder plot that results when applied to the Adam optimizer on the CIFAR-10 dataset.

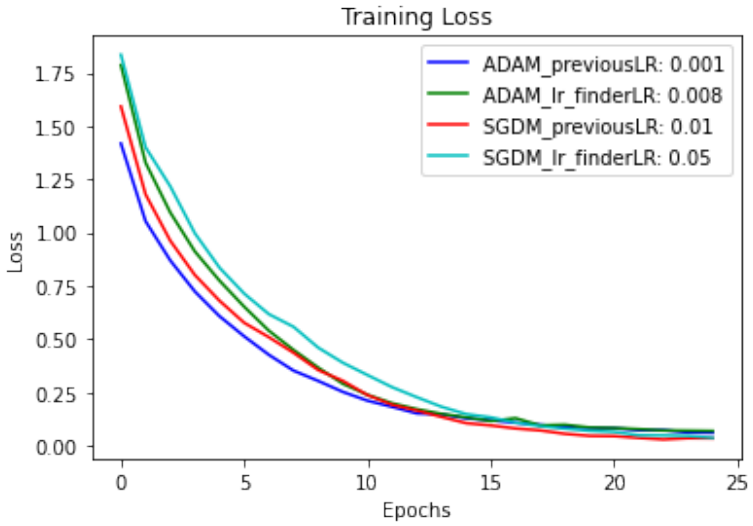

Figure 9: Loss across epochs for models trained with previous learning rates (obtained with grid search) and learning rates selected using a learning rate finder.

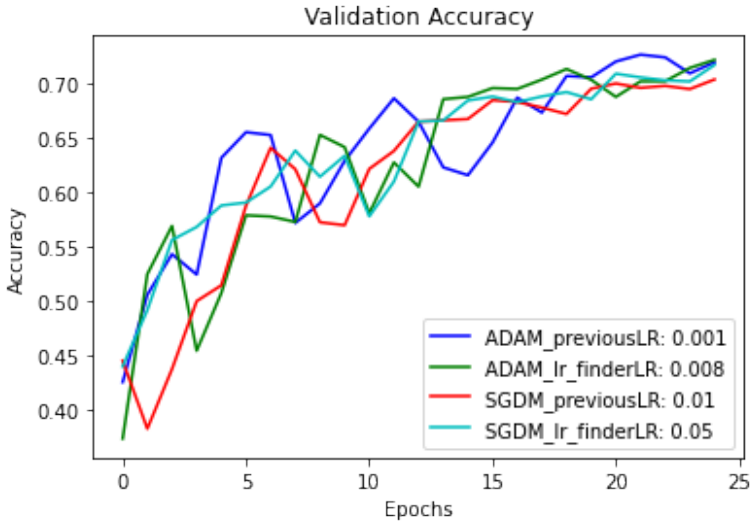

Figure 10: Validation Accuracies across epochs for models trained with previous learning rates (obtained with grid search) and learning rates selected using a learning rate finder.

search. Therefore, it is evident that the learning rate finder provides a reasonable estimate of the optimal learning rate for training a neural network. Additionally, the learning rate finder requires significantly less time compared to performing grid search (i.e., this requires training multiple models with different learning rates).

Learning Rate Schedules

One simple trick to yield improved performance is employing some kind of learning rate schedule in your optimizer. Learning rate schedules have numerous different forms. For example, learning rate annealing slowly decreases the learning rate throughout training so that a lower learning rate is used in later epochs. Step schedules are one of the most common forms of learning rate schedules, which decrease the learning rate at predefined spots in the training loop. In practice, schedulers that decrease the learning rate at predefined intervals during training (e.g., step schedulers), as opposed to smoothly over time, have been shown to perform favorably. Additionally, cyclical learning rate schedules, in which the learning rate fluctuates smoothly between low and high values throughout training, have become quite popular in recent years (e.g., https://arxiv.org/pdf/1506.01186.pdf). In this experiment, I implemented three different types of simple learning rate schedules to analyze their performance compared to models trained with a constant learning rate. The details of these learning rate schedules are as follows:

- **Inverse Annealing**: This learning rate schedule calculates the learning rate at each epoch as $\alpha_e = \frac{\alpha_0}{e}$, where α_e is the learning rate at the current epoch, α_0 is the initial/global learning rate, and e is the current epoch number. Therefore, the learning rate will slowly decrease at each epoch during training.
- **Root Inverse Annealing**: This learning rate schedule is quite similar to Inverse Annealing. The learning rate at each epoch is calculated as $\alpha_e = \frac{\alpha_0}{\sqrt{e}}$. The learning rate decays more slowly due to the square root term in the denominator.
- **Step Schedule**: Step schedulers are one of the simplest but most effective learning rate schedules. The step schedule works by decreasing the learning rate at specified epochs during training. In this experiment, I crafted the step scheduler such that it divides the original learning rate by a factor of 10 60% of the way through training and again by a factor of five 80% of the way through training. Step schedulers can also be made dynamic such that the learning rate is decreased when the loss or validation accuracy plateaus instead of at a hard-coded epoch number.

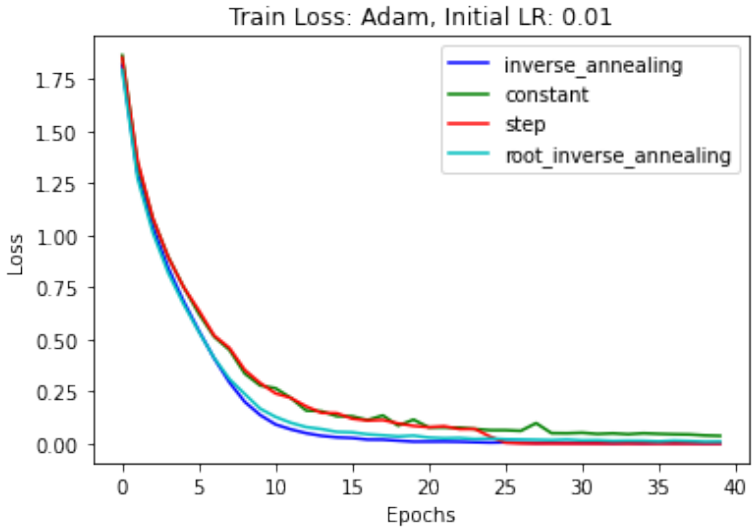

Figure 11: Loss across epochs for models trained with an Adam optimizer and different learning rate schedules on the CIFAR-10 dataset.

Observation and Discussion

- In Fig. 11, it can be seen that all models trained with learning rate schedules converge to a slightly lower loss than the model trained with a

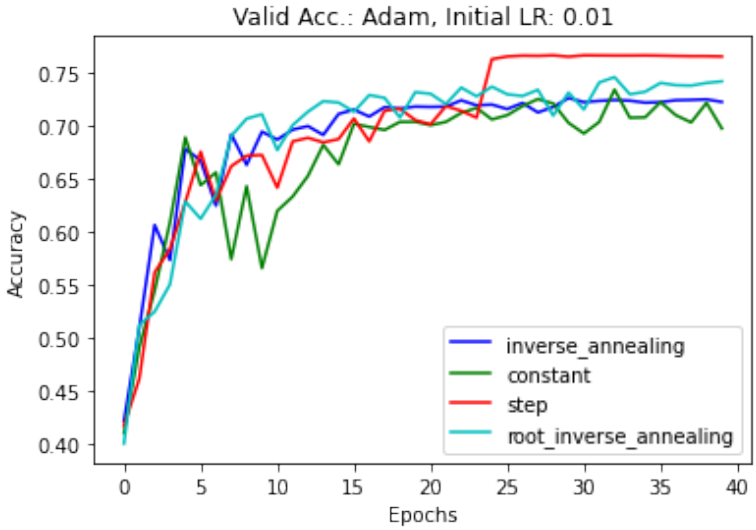

Figure 12: Validation accuracy across epochs for models trained with an Adam optimizer and different learning rate schedules on the CIFAR-10 dataset.

constant learning rate. Intuitively, this makes sense because they utilize much lower learning rates towards the end of training, thus allowing a slightly better loss value to be reached.

- The models trained with each of the proposed learning rate schedules converge to nearly identical loss values.
- It can be seen in Fig. 12 that the model trained with a step scheduler converges to a significantly higher validation accuracy than other models. The sharp increase of accuracy around epoch 25 is characteristic of a step scheduler, as this is the epoch in which the learning rate was divided by a factor of 10. This example demonstrates the simplicity and effectiveness of the step scheduler.
- The models trained with inverse annealing and root inverse annealing yield slightly higher validation accuracy compared to the model trained with a constant learning rate. These models also seem to have a more stable convergence in validation accuracy compared to the model trained with a constant learning rate (i.e., the validation accuracy does not fluctuate as much), which is caused by their usage of lower learning rates in the later epochs of training.
- The model trained with the root inverse annealing schedule converges to a slightly higher validation accuracy than the model trained with the inverse annealing schedule.

Momentum Decay

Momentum decay is recently proposed (https://arxiv.org/pdf/1910.04952.pdf), simple to implement, and has been shown to be quite effective in improving model performance. In most applications, the coefficients of momentum (e.g., β_1 in Adam or γ in SGD with Momentum) are typically set to their default values (i.e., 0.9 in most cases) and rarely tuned. Recent work has shown, however, that model performance can be improved by paying better attention to the value of the momentum term. Additionally, decaying the momentum term throughout training can yield improved performance. For SGD with Momentum, the momentum term can be decayed by using the following recurrence:

$$\beta_t = \beta_0 \frac{1 - \frac{t}{T}}{(1 - \beta_0) + \beta_0(1 - \frac{t}{T})}$$

In this equation β_0 is the initial value of the momentum term, β_t is the momentum term at epoch t, t is the current epoch, and T is the total number of epochs. This equation can easily be applied to SGD with Momentum (i.e., just reset the value of the momentum term at the beginning of every epoch). However, this idea can also be extended to the Adam optimizer. The update rule for the Adam optimizer remains nearly constant. However, the calculation for the first moment approximation becomes the following:

$$m_t = \beta_t m_{t-1} + \bigtriangledown_\theta L(\theta)_t$$

In the above equation, the β_t term is calculated using the momentum decay equation given above. All other aspects of the Adam optimizer's update rule remain the same (i.e., only the calculation of the first moment is altered).

Observation and Discussion

- As seen in Fig. 13, the loss values of models trained with and without momentum decay converge to similar values and do not differ much throughout training.
- The final validation accuracy of the model trained with the Adam optimizer with momentum decay is the best of all models, as seen in Fig. 14. The difference is validation accuracy is slight but noticeable.

All of the optimization tricks combined result in a final validation accuracy of 76.7%, compared to an initial validation performance of about 70%. The accuracy could be further improved by using a larger model, initializing with pre-trained weights, or even training an ensemble of classification models. These further improvements are not presented because they do not improve optimization, but rather the model itself.

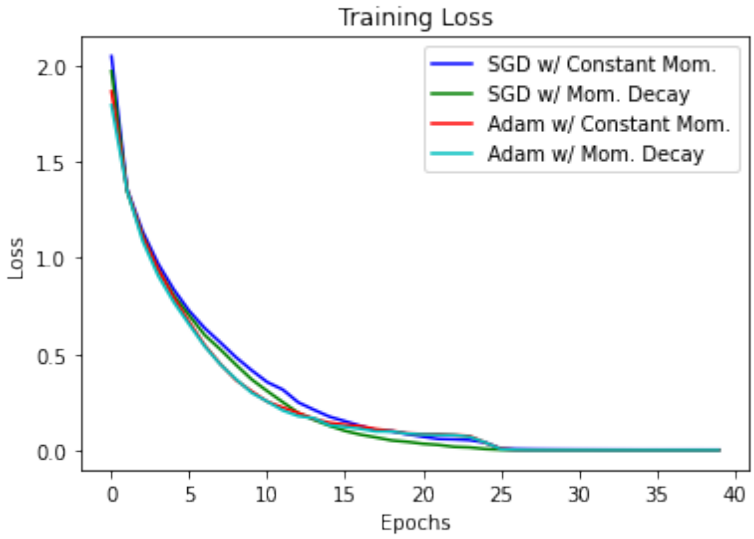

Figure 13: Loss across epochs for models trained with and without momentum decay using both Adam and SGD with momentum.

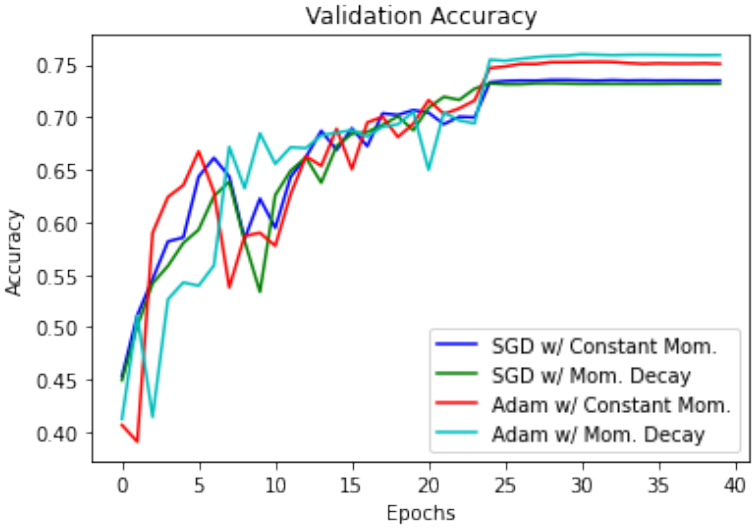

Figure 14: Validation accuracy across epochs for models trained with and without momentum decay using both Adam and SGD with momentum.