CS 169 FALL 2020: FINAL PROJECT

Performance of different optimization methods on neural networks

Team Members:

- Shivan Vipani (SID: 17173295)
- Adarsh Shankar (SID: 11972805)
- Alan Yuen (SID: 49623790)
- William Yzaguirre (SID: 48275576)

Tasks:

- Problem Statement (reword and expand) - Alan
- Previous Work (Adam Paper, find 2 or 3 more on the other optimizers?) - Alan + Adarsh + Will
    - SGD, RMSProp, Momentum (Will)
    - Adam, Adagrad, Adadelta (Adarsh)
- Coding Experience - Shivan + Alan
- Results (Charts and Graphs from Code) - Shivan
- Conclusion
    - Decision off Results (Shivan),
    - Significance of Measurements (Will),
    - Application in Domain (generalization out of multiple datasets for one opt to rise above) (All of Us)
- Bibliography (Previous Work) - Alan + Adarsh
- Bibliography (Code) - Shivan
- Poster Slide - All of Us

Problem Statement: Are there particular optimization methods that perform better under different image classification scenarios for NNets, MLPs, and Logistic Regression?

- Measurements: training speed (ms), accuracy (percentage correct on validation), test loss

Codebase: Keras

List of optimizers: SGD, RMSProp, Adam, Adagrad, Adadelta

Abstract: Many papers and online resources treat optimizers in neural networks as a black box, and rank them hierarchically across all possible use cases. We wanted to look deeper into the workings of the optimizers, and find out if there are particular optimization methods that perform better under scenarios with different data shapes. In order to do this, we first looked into the math behind the algorithms, to try to preemptively find any potential strengths or weaknesses that they might have depending on the data or type of neural network. We then trained three different types of neural networks—linear regression, multi-layer perceptron, convolutional—with five different optimizers—SGD, AdaGrad, AdaDelta, RMSprop, Adam—on three different datasets—MNIST, fashion MNIST, CIFAR-100. We then looked at the training time, training loss, and validation accuracy, across all of these neural networks, and used what we learned about the algorithms strengths and weaknesses to try to figure out the best performing algorithms in different situations, and why they performed better.

Optimizers are used in supervised training of neural networks to minimize a cost—sometimes called a loss function—function for each input. This cost function is used to give you the magnitude of error between the output of the neural network for an input, and the actual training label—a training label is the given correct output—for that same input. The optimizers we are researching are known as descent algorithms, because they are variations on basic gradient descent optimization. Gradient descent in neural networks minimizes the cost function, C, by repeatedly updating the model's parameters, θ, to get more accurate outputs for each input until you reach a minimum. It does this by updating the parameters in the direction of the negative gradient, with respect to the parameters, $-\nabla_{\theta} C(\theta)$. The direction is then multiplied by the learning rate, η, which serves as the step size, and then added to the parameters to update them. As you do this for every input, your neural network should get a smaller error for each input, causing it to output the correct answer for each input more frequently, even with input that it was not trained with. In our research, we look into the efficacy of six common descent optimizing functions used in neural networks: Stochastic Gradient Descent, Momentum, ADAM, AdaGrad, AdaDelta, and RMSProp.

I. Stochastic Gradient Descent - [1]

Stochastic **G**radient **D**escent, or *SGD*, works very similarly to gradient descent, except the training inputs are evaluated in a random order. This attempts to solve the problem that gradient descent always converges to a single local minimum for a given set of parameters. With SGD, though, because inputs are evaluated randomly, as you optimize, you move around the possible parameter space, leading to a possibly better minimum. For each training input $x^{(i)}$ and label $y^{(i)}$, the parameters are updated with:

$$\Theta^{(i+1)} = \theta^{(i)} + \eta * (-1) * \nabla_{\theta} C(\theta; x^{(i)}; y^{(i)})$$

or, in pseudocode:

```
Function: Stochastic Gradient Descent($\eta$)
 training_data ← random_shuffle(training_data)
for input in training_data:
        parameter_gradient ← evaluate_gradient(parameters)
        parameters ← parameters + $\eta$ * (-1) * parameter_gradient
```

Advantages:

- One of the least computationally intensive optimizers. Simply find the negative gradient of the parameters with respect to the cost function, , multiply this by the learning rate η, and add this back to the parameters.

Disadvantages:

- The random fluctuation makes it harder to converge to an exact minimum, meaning much longer convergence times.

[1] Ruder, Sebastian. *An Overview of Gradient Descent Optimization Algorithms*, Sebastian Ruder, 2016, ruder.io/optimizing-gradient-descent/.

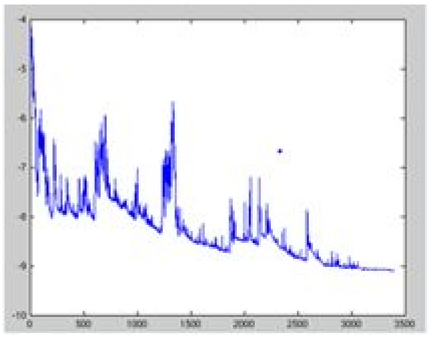


Figure 1: Fluctuations in cost function as parameters are updated with SGD[2]

- Fixed learning rate means you will most likely use a value that will be too small and take longer to converge, or use too large a value and will overshoot.
- Tends to get stuck in saddle points, which tend to be near plateaus where the gradient is close to zero.

Performance on Neural Networks:

SGD will generally perform worse than other more modern optimization algorithms. It really serves as the building block for more modern optimization algorithms, like the ones we discuss after this. As such, it is generally used more often now as a benchmark for testing other optimizers as a baseline, and has very little use in real neural network applications.

II. Momentum - [3]

Momentum is the colloquial term for SGD with momentum. Momentum tries to speed up SGD by adding a fraction of the previous step, to the current. If SGD is analogous to dropping a ball down a hill and letting it roll to a local minimum at a constant speed, momentum is analogous to that ball having a constant acceleration as it rolls towards a local minimum.

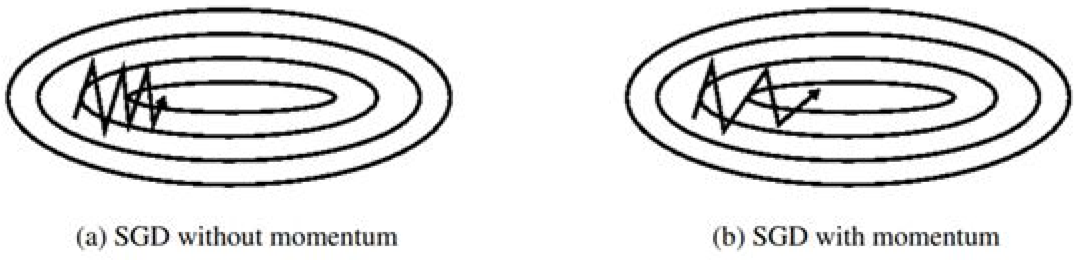


Figure 2[4]

[2] "Stochastic Gradient Descent." *Wikipedia*, Wikimedia Foundation, 13 Dec. 2020, en.wikipedia.org/wiki/Stochastic_gradient_descent.

[3] Ruder, Sebastian. *An Overview of Gradient Descent Optimization Algorithms*, Sebastian Ruder, 2016, ruder.io/optimizing-gradient-descent/.

[4] Ruder, Sebastian. *An Overview of Gradient Descent Optimization Algorithms*, Sebastian Ruder, 2016, ruder.io/optimizing-gradient-descent/.

The new update value is calculated with a momentum term γ, and has the form:

$$v^{(t)} = \gamma v^{(t-1)} + \eta * \nabla_{\theta} C(\theta)$$

$$\theta^{(t+1)} = \theta^{(t)} - v^{(t)}$$

or, in pseudocode:

```
**Function:** Momentum($\eta$, $\gamma$)
 training_data ← random_shuffle(training_data)
update_value[0] ← 0
**for** input **in** training_data:
        parameter_gradient ← evaluate_gradient(parameters)
        update_value[input] ← update_value[input – 1] * $\gamma$ + $\eta$ * parameter_gradient
        parameters ← parameters – update_value[input]
```

Advantages:

- Faster convergence than SGD
- Less oscillation than SGD
- Moves through plateaus faster, so less likely to get stuck on a saddle point.

Disadvantages:

- Momentum keeps building, even near minimum. This can cause overshooting once you reach the minimum.
- All the parameters get the same update momentum, which can cause over-shooting and ensures oscillation, though less than SGD, towards minimum (this problem is addressed in the adaptive optimizers discussed below)

Performance on Neural Networks:

Momentum solves SGD's slow convergence rate and tendency to get trapped in saddle points well, but also almost too well. Figure three, below, illustrates this quite well:

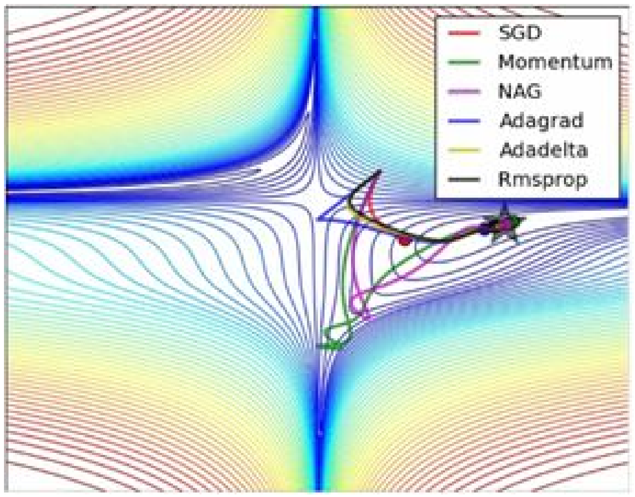


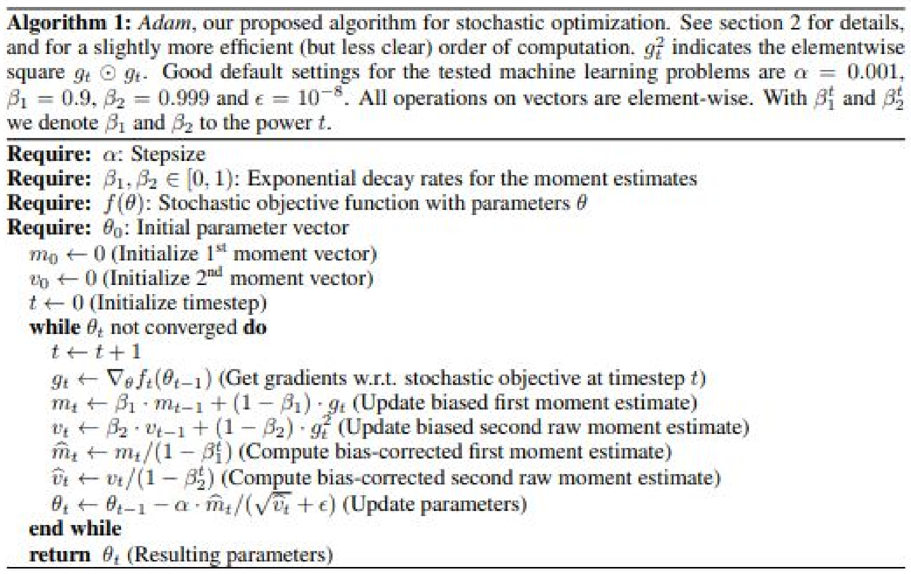
Figure 3[5]

This figure shows the “contours of a loss surface and time evolution of different optimization algorithms”. It is important to note how far momentum over shot the minimum on it’s descent. So while it does generally converge faster than SGD, you can see it had yet to converge in this test while all the other optimizers had, it has some major drawbacks that are addressed in more modern algorithms, such as Nesterov Momentum.

[5] “CS231n Convolutional Neural Networks for Visual Recognition.” *Stanford.edu*, Stanford University, cs231n.github.io/neural-networks-3/.

III. ADAM – [6]

Adam or **<u>Ada</u>**ptive **<u>M</u>**oment estimation is another method for stochastic optimization that just requires the first-order gradients with a lower space complexity. This method computes the adaptive learning rates for each individual different parameter from estimates of the first and second moments of the gradients.



Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

Advantages:-

- Combines the advantages of RMSProp which works well in nonstationary settings and Adagrad which works well on sparse gradients.
- The magnitudes of parameter updates do not affect the rescaling of the gradient
- The step-sizes are bounded by the step-size given in the hyperparameter
- This method does not require a stationary objective making this more efficient than others in terms of performance and memory requirement.

We can compute the decaying averages of the first and second moment based on the hyperparameters $\beta_1, \beta_2 \epsilon [0, 1)$ as follows: (shown in the pseudocode above)

$m_t = \beta_1 m_{t-1} - (1 - \beta_1) g_t$ [estimate of the first moment]

$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$ [estimate of the second moment]

As m_t and v_t are initialized to 0, it is mentioned these estimates are biased towards zero especially during the initial time steps. So, to counteract these biases we compute the bias-corrected first and second moment estimates as follows:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Which are then used to update the parameters similar to AdaDelta or RMSProp, yielding

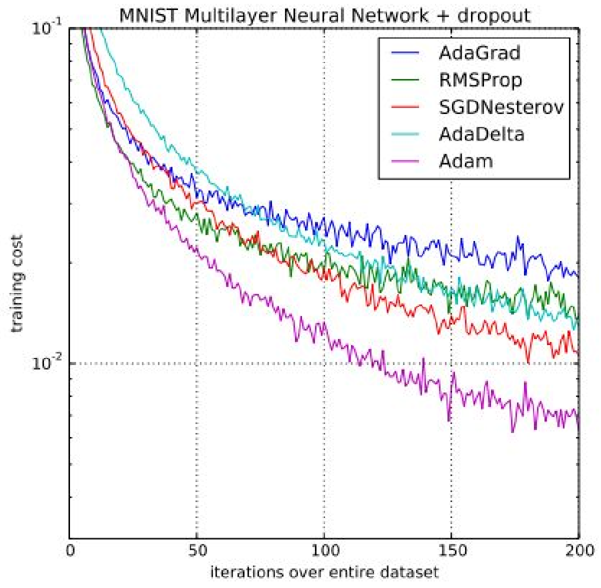
[6] Kingma, D. P., & Ba, J. L. (2015). Adam: a Method for Stochastic Optimization. International Conference on Learning Representations

the Adam update rule:

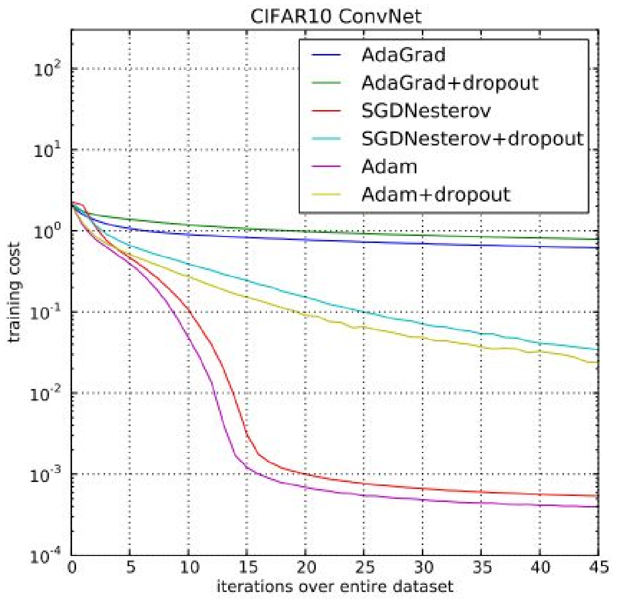
$$\theta_{t+1} = \theta_t - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}$$

<u>Performance on Neural Networks:</u>

- Multi-Layer Neural Networks:
    - In the experiment described in Section 6.2[7], a neural network with two fully connected hidden layers with 1000 hidden units each.
    - They use 5 different optimization methods for this model and as we can see in the graph below, Adam makes faster progress in terms of both the number of iterations and wall-clock time. [SHOWN ON THE RIGHT]



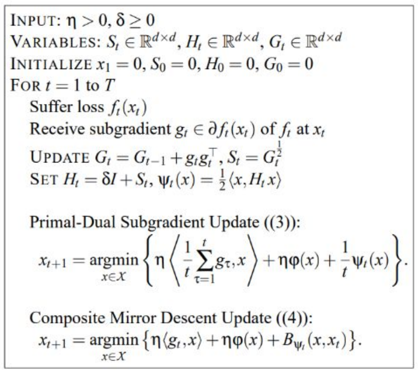
- Convolutional Neural Networks: (CNN)
    - The CNN architecture used in this experiment has three alternating stages of 5x5 convolution filters and 3x3 max pooling with stride of 2 that are followed by a fully connected layer of 1000 rectified linear hidden units.
    - Although both Adam and Adagrad make quick progress decreasing the cost in the initial training stages, for CNNs Adam and SGD converge much faster than Adagrad.[SHOWN ON THE RIGHT]
    - Adam proves to be marginally better than SGD with momentum since it adapts the learning rate scale for different layers as opposed to manually hand picking in SGD.



<u>Conclusion:</u>

Adam is easy and efficient to implement and is a robust method due to its low space complexity, well-suited to a wide variety of non-convex optimization problems in machine learning.

[7] Kingma, D. P., & Ba, J. L. (2015). Adam: a Method for Stochastic Optimization. International Conference on Learning Representations



```
INPUT: $\eta > 0, \delta \geq 0$
VARIABLES: $S_t \in \mathbb{R}^{d \times d}, H_t \in \mathbb{R}^{d \times d}, G_t \in \mathbb{R}^{d \times d}$
INITIALIZE $x_1 = 0, S_0 = 0, H_0 = 0, G_0 = 0$
FOR $t = 1$ to $T$
    Suffer loss $f_t(x_t)$
    Receive subgradient $g_t \in \partial f_t(x_t)$ of $f_t$ at $x_t$
    UPDATE $G_t = G_{t-1} + g_t g_t^\top, S_t = G_t^{\frac{1}{2}}$
    SET $H_t = \delta I + S_t, \psi_t(x) = \frac{1}{2}\langle x, H_t x \rangle$

    Primal-Dual Subgradient Update ((3)):
      $x_{t+1} = \operatorname*{argmin}_{x \in \mathcal{X}} \left\{ \eta \left\langle \frac{1}{t}\sum_{\tau=1}^{t} g_\tau, x \right\rangle + \eta\varphi(x) + \frac{1}{t}\psi_t(x) \right\}.$

    Composite Mirror Descent Update ((4)):
      $x_{t+1} = \operatorname*{argmin}_{x \in \mathcal{X}} \{ \eta\langle g_t, x \rangle + \eta\varphi(x) + B_{\psi_t}(x, x_t) \}.$
```

Figure 2: ADAGRAD with full matrices

IV. ADAGRAD – [8]

AdaGrad or **Ada**ptive **Grad**ient algorithm is an update to the stochastic gradient descent algorithm with per-parameter learning rate making this algorithm an adaptive learning rate method. This method performs smaller updates on the more frequent parameters and larger updates on the less frequent parameters.

Advantages:-

- Experiments have been done to show that AdaGrad has greatly improved the robustness of stochastic gradient descent and was used for training large-scale neural networks at Google.[9] So this method is well suited for large scale neural networks with sparse data.
- Eliminates the need to manually tune the learning rate since we use a different learning rate for every parameter ϴ for every time step t given by:

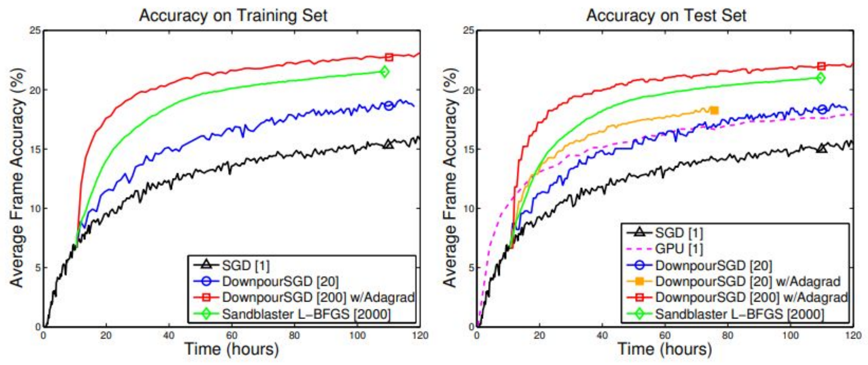
$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \varepsilon}} \otimes g_t$ - G_t is the sum of squares of past gradients w.r.t all Θ_t(s)

Disadvantage:-

The denominator in the above equation, the accumulation of the sum of squared gradients keeps on growing. The more we run this algorithm, the learning rate η becomes infinitesimally small to the point where the algorithm is no longer learning and not able to acquire new information. (This problem is resolved in AdaDelta and Adam)

Performance on Neural Networks:

In the



experiments previously mentioned, the graphs above show the performance of different optimization methods for both the training and testing sets. Sandblaster L-BFGS is the more optimal method without AdaGrad. But we can see that the fastest method is the Downpour-SGD method with AdaGrad. Hence, Downpour-SGD method with AdaGrad can train models much faster than a high-end GPU as shown above. (assuming we are given access to sufficient CPU resources)

Conclusion:

AdaGrad automatically stabilizes volatile parameters and naturally adjusts learning rates to the requirements of different layers in the neural network and can perform better than a standard GPU.

[8] Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12, 2121–2159

[9] Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V, ... Ng, A. Y. (2012). Large Scale Distributed Deep Networks. NIPS 2012: Neural Information Processing Systems, 1–11

V. ADADELTA – [10]

AdaDelta is a modification of the AdaGrad algorithm that aims to decrease its monotonic, aggressively decreasing learning rate (as mentioned in the disadvantages before). This is resolved by restricting the window of past accumulated gradients to a fixed size *w* as opposed to accumulating all past squared gradients.

Algorithm 1 Computing ADADELTA update at time t

```
Require: Decay rate $\rho$, Constant $\epsilon$
Require: Initial parameter $x_1$
1: Initialize accumulation variables $E[g^2]_0 = 0$, $E[\Delta x^2]_0 = 0$
2: for $t = 1 : T$ do %% Loop over # of updates
3:     Compute Gradient: $g_t$
4:     Accumulate Gradient: $E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2$
5:     Compute Update: $\Delta x_t = -\frac{\text{RMS}[\Delta x]_{t-1}}{\text{RMS}[g]_t} g_t$
6:     Accumulate Updates: $E[\Delta x^2]_t = \rho E[\Delta x^2]_{t-1} + (1-\rho)\Delta x_t^2$
7:     Apply Update: $x_{t+1} = x_t + \Delta x_t$
8: end for
```

Advantage:-

The main advantage of AdaDelta is that it is not required to set a default learning rate since we are taking the ratio of the running average ($E[g^2]_t$) of the previous steps to that of the current gradient, so:

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

$$\Rightarrow \Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} \cdot g_t$$

$\rightarrow$ where, $RMS[f]_t = \sqrt{E[f^2]_t + \epsilon}$

And $\theta_{t+1} = \theta_t + \Delta\theta_t$

$\rightarrow$ AdaDelta update rule

So, with Adadelta we do not need a default learning rate since it has been eliminated from the update rule.

Performance on Neural Networks:

The experiments conducted by Zeiler[5] for speech data was done by setting up a neural network where the inputs 26 frames of audio and the outputs were 8000 senone labels and each layer of the hidden network had 2560 hidden units.

Fig. 3 on the right shows the comparison between AdaGrad and AdaDelta when using 100 network replicas. We can see that AdaDelta outperforms AdaGrad and initially converges faster too throughout the training in terms of frame classification accuracy on the test set.

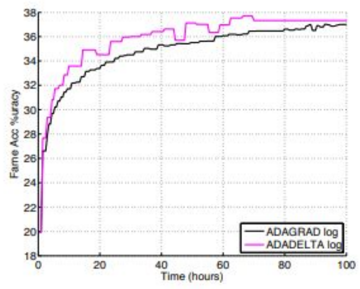


Fig. 3. Comparison of ADAGRAD and ADADELTA on the Speech Dataset with 100 replicas using logistic nonlinearities.

Fig. 4 on the right shows the comparison between AdaDelta and AdaGrad when using 200 network replicas. Even here we can see that AdaDelta still marginally outperforms AdaGrad and also rapidly converging to the same accuracy as the other methods.

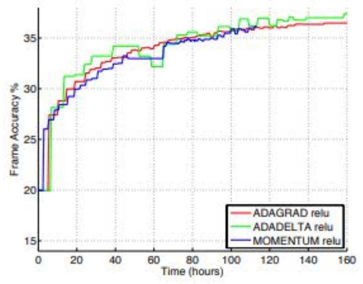


Fig. 4. Comparison of ADAGRAD, Momentum, and ADADELTA on the Speech Dataset with 200 replicas using rectified linear nonlinearities.

Conclusion:

[10] Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method

In spite of all the nonlinearities, the number of replicas, the number of hidden units or even the variety of input data types, the hyperparameters for this algorithm did not need to be tuned. Making AdaDelta a robust learning rate method that somewhat fixes the issue that AdaGrad faces and can be applied in a multitude of optimization problem.

VI. RMSprop - [11]

RMSprop is an adaptive optimizer that was developed around the same time as AdaDelta, and, despite being created completely separately from AdaDelta, is created to fix the same problem in a similar manner. Just like AdaDelta, it was created to combat the shrinking learning rate of AdaGrad. Similarly to AdaDelta, it keeps a running average of all past squared gradients, . However, unlike AdaDelta which does not use a fixed learning rate, RMSprop is focused on updating a constant learning rate, at each step, using:

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

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}} g_t$$

Or, in pseudocode:

```
**Function:** RMSprop($\eta$, $\gamma$):
 training_data ← random_shuffle(training_data)
 $E[g^2]_0 \leftarrow 0$
**for** input **in** training_data:
        parameter_gradient ← evaluate_gradient(parameters)
        $E[g^2]_{input} \leftarrow \gamma E[g^2]_{input-1}$ + (1 - $\gamma$ )parameter_gradient$^2$
        parameters ← parameters – $\frac{\eta}{\sqrt{E[g^2]_t + \varepsilon}}$ * parameter_gradient
```

Advantages:

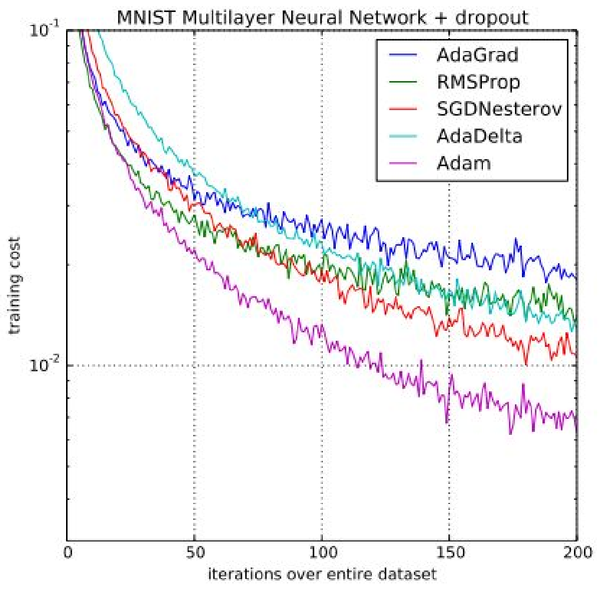
- Eliminates AdaGrad's diminishing learning rate issue.
- Adaptively computes an ideal step size for each parameter.
- Slightly less computationally and memory intensive than AdaDelta.

Disadvantages:

- Has to use a set learning rate, which needs to be tested to find ideal value.

Performance on Neural Networks:

Looking at the figure to the right, from the experiment described in Section 6.2—which uses a typical multilayer neural network to identify handwritten images—you can see both the similarity in performances between RMSprop and AdaDelta, and their clear advantage over AdaGrad.



[11] Ruder, Sebastian. *An Overview of Gradient Descent Optimization Algorithms*, Sebastian

Significance of measurements

In order to test the efficacy of different optimizers, it is important to define metrics with which to measure our results. In order to do this, we chose to compare the training speed of each optimizer, the percent correct while testing on validation data, and the loss while training of each optimizer across all tests. We will now present the significance of these metrics:

Training speed: Each day, 2.5 quintillion bytes of data is created[12], and this number keeps growing each day. That is a lot of data we could be processing and adding to neural networks to train. Unfortunately, though, it seems like Moore's Law is beginning to slow down. This means that our ability to process large amounts of information from a hardware perspective is currently growing slower than our production of new data. As such, it is important to prioritize speed on the software side, and develop algorithms that perform quick enough to keep up with the insane growth of our data-driven economy.

Validation Score: A supervised neural network's job is to be able to correctly categorize data it has not seen before. As such, a neural network with a poor percentage of correctly identified validation inputs is pretty much worthless. It might be tempting then to assume that whichever optimizer produces the best validation score is the best optimizer. However, if the neural network with the highest score also took a year to train due to slow algorithms, that neural network is also basically worthless. This shows the importance of having a good balance between speed and accuracy.

Training Loss: The training loss is the direct value that the optimizers try to lower. As such, it is very important to keep track of this value throughout the training process, so that you can more accurately evaluate the efficacy of an optimizer. It also can be used to identify interesting patterns in your optimizer that you would not be able to see otherwise. For example, look back at figure 3. From this, we are able to clearly see how the momentum-based algorithms (momentum and NAG) overshot before moving back towards the minimum. This is a very interesting metric that would otherwise be missed if you were to not analyze the training loss.

[12] Bulao, Jacquelyn. "How Much Data Is Created Every Day in 2020? [You'll Be Shocked!]." *TechJury*, 10 Sept. 2020, techjury.net/blog/how-much-data-is-created-every-day/.

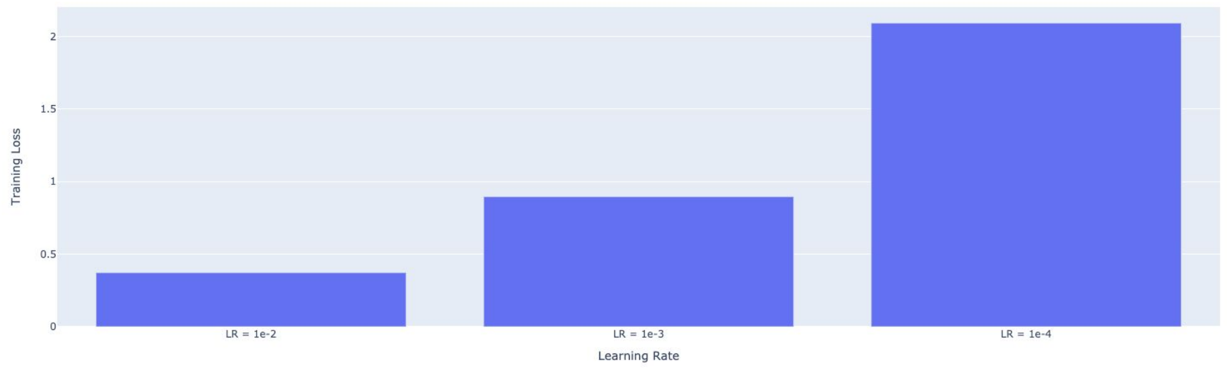
Experience with Coding the Experiment

For the evaluation of different optimizers on machine learning models, we used Google Colaboratory to implement Tensorflow/Keras, Numpy, Pandas, and Plotly. With Keras' libraries, we were able to pull up our three datasets to download locally to process and reshape according to the needs of each machine learning model. After setting up evaluation methods that can measure the training conditions of a single training attempt on one dataset for a model, additional methods were written to simulate testing the optimizers with different learning rates and determine the best hyperparameter configuration for that given optimizer and model such that the optimizers can then be ready to be compared amongst each other. The hyperparameter selection process was visualized for us using Plotly bar graphs to see which learning rates the optimizers were ending up with. After hyperparameter selection is graphed and complete, the trajectory of the optimizer's training loss is graphed over the twenty epochs that the models have been trained with for our provided example. Because we wanted to evaluate and consolidate the performance of each optimizer on these three models in different data scenarios, we split our testing into three separate notebooks after drafting up our functions necessary for the experiment. Each notebook was dedicated to a different dataset (MNIST, Fashion MNIST, and CIFAR-100) to distribute the testing execution among different machines.

Results

As mentioned above, evaluations and comparisons of the optimizers on various models and datasets required us to first narrow down the optimal hyperparameter configuration for each optimizer in that given situation. To do so, a line search was performed, assessing training loss against adjustments to the learning rate. To visualize this process as we automated the selection of the hyperparameter, we constructed bar graphs for each optimizer.

Hyperparameter Line Searching for Learning Rate of Optimization Functions



Ex: Bar graph of Adagrad evaluation to determine its optimal learning rate for a CNN on MNIST data

MNIST

Model	Optimizer	Training Time	Training Loss	Validation Accuracy
CNN	Adam	36.243	0.008	**0.988**
	SGD	**35.451**	0.093	0.976
	RMSProp	39.315	**0.007**	**0.988**
	Adadelta	36.278	0.224	0.945
	Adagrad	35.906	0.053	0.983
LR	Adam	20.143	**0.235**	**0.927**
	SGD	**19.493**	0.433	0.896
	RMSProp	20.763	0.251	**0.927**
	Adadelta	19.987	1.050	0.804
	Adagrad	19.839	0.373	0.906
MLP	Adam	26.056	0.398	**0.959**
	SGD	**24.950**	1.207	0.936
	RMSProp	29.163	**0.373**	0.957
	Adadelta	25.225	2.257	0.919
	Adagrad	25.203	0.435	0.952

Fashion MNIST

Model	Optimizer	Training Time	Training Loss	Validation Accuracy
CNN	Adam	29.535	**0.109**	0.887
	SGD	28.460	0.507	0.806
	RMSProp	**24.194**	0.134	**0.890**
	Adadelta	29.966	0.692	0.755
	Adagrad	28.991	0.389	0.849
LR	Adam	20.990	**0.367**	**0.840**
	SGD	**20.484**	0.564	0.805

	RMSProp	21.699	0.386	**0.840**
	Adadelta	20.843	0.914	0.691
	Adagrad	20.715	0.510	0.819
MLP	Adam	25.898	0.615	**0.844**
	SGD	26.110	1.365	0.833
	RMSProp	28.267	**0.592**	0.841
	Adadelta	26.192	2.784	0.820
	Adagrad	**25.279**	0.637	0.836

CIFAR-100

Model	Optimizer	Training Time	Training Loss	Validation Accuracy
CNN	Adam	38.773	**1.479**	**0.428**
	SGD	**38.001**	3.623	0.179
	RMSProp	42.040	1.508	0.404
	Adadelta	39.299	4.232	0.077
	Adagrad	38.834	3.027	0.275
LR	Adam	22.441	**3.360**	**0.173**
	SGD	**21.558**	3.800	0.143
	RMSProp	22.671	3.390	0.171
	Adadelta	21.692	4.138	0.095
	Adagrad	22.248	3.632	0.165
MLP	Adam	27.880	3.938	**0.158**
	SGD	**26.670**	5.607	0.139
	RMSProp	34.530	**3.921**	0.128
	Adadelta	29.066	8.569	0.116
	Adagrad	27.545	3.946	0.152

Conclusion

It can be intractable to do hyperparameter search in every experiment, especially when models and datasets are larger. Having a default optimizer that has good general performance on various models and datasets can save time and compute resources. The application of the results of this project is finding and making use of such an optimizer.

For our 9 experiments, Adam had the best average performance in training loss and validation accuracy. RMSProp surprisingly had comparable performance to Adam, having the second best average performance in these categories. In terms of training time, SGD had the lowest training time on average. Since we often care about model convergence and classification performance more than training time, we also compared training times between Adam and RMSProp. Adam was the clear winner in the 9 experiments, having a better training time in 8 of them. From these experiments, Adam has proven itself to be a very solid and failsafe baseline optimizer to use in many if not machine learning tasks.

Evaluating our approach, we were limited by the amount of time and compute and were only able to run 9 experiments with mostly default hyperparameters. Future work includes also looking at convergence rates in experiments, rather than just last epoch training loss, test a wider range of learning rate values and other hyperparameters, and run each experiment multiple times with random initial parameters and use average as the quantitative metric.

BIBLIOGRAPHY:

Bulao, Jacquelyn. “How Much Data Is Created Every Day in 2020? [You'll Be Shocked!].” *TechJury*, 10 Sept. 2020, techjury.net/blog/how-much-data-is-created-every-day/.

“CS231n Convolutional Neural Networks for Visual Recognition.” *Stanford.edu*, Stanford University, cs231n.github.io/neural-networks-3/.

Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V, … Ng, A. Y. (2012). Large Scale Distributed Deep Networks. NIPS 2012: Neural Information Processing Systems, 1–11

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12, 2121–2159

Khandelwal, Renu. “Overview of Different Optimizers for Neural Networks.” *Medium*, Data Driven Investor, 4 Feb. 2019, medium.com/datadriveninvestor/overview-of-different-optimizers-for-neural-networks-e0ed119440c3.

Kingma, D. P., & Ba, J. L. (2015). Adam: a Method for Stochastic Optimization. International Conference on Learning Representations

Ruder, Sebastian. *An Overview of Gradient Descent Optimization Algorithms*, Sebastian Ruder, 2016, ruder.io/optimizing-gradient-descent/.

“Stochastic Gradient Descent.” *Wikipedia*, Wikimedia Foundation, 13 Dec. 2020, en.wikipedia.org/wiki/Stochastic_gradient_descent.

Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method