Approaches to Effective Hyperparameter Tuning in Adam, RMSProp and AdaGrad Optimization Algorithms

GitHub Repo: https://github.com/anirudhmadhusudan/Effective-Hyperparameters-Tuning-For-Adaptive-Optimizers

Anirudh Madhusudan, University of Illinois, Urbana-Champaign

Abstract: Tuning the learning rates is an expensive process, and a lot of research has gone into adaptively tuning the learning rates, and even do so per parameter. Three algorithms in particular have become common place in Deep Learning optimization namely - Adam, RMSProp and AdaGrad. While these methods help with tuning learning rates, there are still other hyper-parameter settings within these algorithms that are required. This paper will explore and explain the three optimizers and how they achieve convergence. This paper will also present experimental results on popular datasets (CIFAR10 & MNIST) by tuning the hyper-parameters of these optimizers and suggesting effective strategies for performan

Introduction

Three algorithms namely – AdaGrad, RMSprop, Adam are quite popular in most Deep Learning modules that are used today. TensorFlow, Keras, PyTorch, Caffe and other modules have incorporated these algorithms for optimization. While using these optimizers, the default arguments, (or default hyperparameters) are usually used without necessarily tuning the hyperparameters. This is because these optimizers are well-behaved for a broad range of hyperparameter values compared to a constant learning rate. However, that doesn't infer that they cannot be optimized further. This paper will first review the three algorithms and why the perform optimally. It will them talk about experiments that were carried out to come up with strategies for tuning the hyperparameters of these optimizers. Finally it will present good strategies that can to help choose these optimizers and tune their hyperparameters.

Review of Concepts

Stochastic Gradient Descent

The Stochastic Gradient Descent (SGD), is a stochastic approximation of the gradient descent optimization and iterative approach for minimizing the objective function. The SGD is a popular algorithm for training a wide range of models in Machine Learning specifically in Support Vector Machines, Logistic Regression and Graphical Models. In combination with back-propagation, it is the standard algorithm for training Artificial Neural Networks. Many improvements on the basic SGD algorithm have been proposed and used so far. In particular, the need to set a specific learning rate or step size has often been recognized as an issue.

While SGD is a fundamental algorithm for optimization in Machine Learning and Artificial Neural Networks, it comes with some problems in actual practice. SGD progresses very slowly along shallow dimensions and jitters along steeper directions. Additionally, due to a zero gradient at local minima and at saddle points (which are common in high dimension datasets), SGD gets stuck. A simple trick of adding a momentum term, addresses these problems. The following codes illustrate these two:-

```
#SGD

while True:

      dx = compute_gradient(x)

      x = x + learning_rate*dx


#SGD + Momentum

vx = 0

while True:

      dx = compute_gradient(x)

      vx = rho*vx + dx
```

```
        x = x + learning_rate*vx
```

The idea of SGD + Momentum is simple – we maintain a velocity over time and add our gradient estimates to the velocity. We step in the direction of velocity rather than stepping in the direction of the gradient. The hypermeter **rho** corresponds to "friction". For every time step we take in the direction of the current velocity, we decay it by friction and add in the gradient. A physical interpretation of this would be a ball rolling down a hill that picks up speed as it comes down. Now, even if we pass the point of local minima, or saddle point, because we have velocity, we will pass the area of zero gradient. The resulting convergence with SGD + Momentum is quicker and smoother than vanilla SGD.

AdaGrad

AdaGrad (or Adaptive Gradient Algorithm) is a modified SGD with per-parameter learning rate. In simple terms, the idea behind AdaGrad is to increase the learning rate for sparse parameters and decrease the learning rate for non-sparse parameters. This method improves convergence when data is sparse, and these sparse are informative.

The code for AdaGrad is given as:

```
#AdaGrad
grad_squared = 0
while True:
        dx = compute_gradient(x)
        grad_squared = Grad_squared + dx^2
        x - = learning_rate * dx/(sqrt(grad_squared) + e)
```

During the optimization, we keep summing the squared of gradients that is seen during training. Rather than having a velocity term, we have a gradient squared term and during training we keep adding the squared gradient to the grad_squared term. When we update our parameter vector, we divide by the grad_squared term and a small value (e, to prevent division by zero). As the grad_squared gets larger monotonically, the step size gets smaller over time. The advantage with AdaGrad is, it works really well in a convex-case, where the local minima is the global minima. We want the step size to get smaller over time. However, in a non-convex case, the algorithm gets stuck at a saddle point, which is problematic. Given these reasons, in most non-convex cases AdaGrad is generally avoided.

RMSprop

RMSprop is a slight variation of AdaGrad that addresses its inefficiencies. The idea of RMSprop is to divide the learning rate for a weight by running average of magnitudes of recent gradients for that weight. To explain further, with RMSprop, we still keep grad_squared, but instead of letting the grad_squared accumulate over training, we let the squared estimate decay. So this seems similar to the momentum update, except that we have a momentum over grad_squared, rather than the gradients itself. The code for RMSprop is given as:

```
#RMSProp

grad_squared = 0

while True:

      dx = compute_gradient(x)

      grad_squared = decay_rate*grad_squared +

            (1-decay_rate)*dx*dx

      x - = learning_rate * dx/(sqrt(grad_squared) + e)
```

So Now with RMSProp after we compute our gradient, we take our current estimate of the grad_squared, we multiply it with a decay_rate which is commonly between 0.9 and 0.999 then we add this (1-decay_rate) over squared gradient. When we make our step, it looks like AdaGrad where we have accelearating movement along one dimension and slowing down movement on the other. Given that the estimates are leaky, it addresses the problem of slowing down where we might not want to. RMSprop performs well for online and non-stationary problems.

Adam

In momentum we saw the concept of velocity, where we build up velocity, by adding in gradients and moving in the direction of velocity. Similarly, in AdaGrad & RMSprop we build up estimates of square gradients and then divide by the square gradients. Adam combines the best features of both momentum and RMSprop. The code for Adam is given as follows:

```
#Adam

first_moment = 0

second_moment = 0

for t in range(num_iter):

      dx = compute_gradient(x)

#momentum equivalent step

      first_moment = beta1*first_moment + (1-beta1)*dx

      second_moment = beta2*second_moment + (1-beta2)*dx*dx

#bias-correction
```

```
      first_unbias = first_moment/(1-beta1**t)

      second_unbias = second_moment/(1-beta2**t)

#AdaGrad/RMSprop equivalent step

      x  - =  lr*first_unbias/(sqrt(second_unbias)+e))
```

As can be seen in the code block above, the first moment is very similar to the momentum. The second moment and step size update is similar to AdaGrad/RMSprop. Given this combination Adam performs very well in non-convex cases while being computationally efficient and well-suited for different types of data/parameters.

Hyperparameter Tuning

The three algorithms namely – AdaGrad, RMSprop, Adam are quite popular in most Deep Learning modules that are used today. TensorFlow, Keras, PyTorch, Caffe and other modules have incorporated these algorithms as functions used during optimization. While using these optimizers, the default arguments, (or default hyperparameters) are usually used without necessarily tuning the hyperparameters. This is because these optimizers are well-behaved for a broad range of hyperparameter values compared to a constant learning rate. However, that doesn't infer that they cannot be optimized further. To illustrate this fact, take the Adam Optimizer on TensorFlow. The default arguments are `lr  = 0: 001, beta1 = 0: 9, beta2 = 0: 99, epsilon  = 1e -  08`

In the following sections, we explore experiments carried out to generalize approaches that can be adopted to tune the hyperparameters of these algorithms. Keras, an open source neural library written in Python is used to carry out this study. The following functions from the keras.optimizers shall be used in this study

`keras. optimizers.Adagrad (lr=0.01, epsilon=None, decay=0.0)`

Arguments:

lr: float>=0. Learning Rate

epsilon: float>=0. If None, defaults to K.epsilon()

decay: float>=0. Learning rate decay over each update

`keras. optimizers.RMSprop (lr=0.01, rho = 0.9,`
`epsilon=None ,decay=0.0)`

Arguments:

lr: float>=0. Learning Rate

rho: float>=0

epsilon: float>=0. If None, defaults to K.epsilon()

decay: float>=0. Learning rate decay over each update

```
keras. optimizers.Adam (lr=0.01, beta_1 = 0.9, beta_2 = 0.999
epsilon=None, decay=0.0)
```

Arguments:

lr: float>=0. Learning Rate

beta_1: float, 0<beta<1, generally close to 1

beta_2: float, 0<beta<1, generally close to 1

epsilon: float>=0. If None, defaults to K.epsilon()

decay: float>=0. Learning rate decay over each update

Experiments

We tune hyperparameters of the CIFAR10 and MNIST dataset to find the optimal values of all arguments for each function. We then try to generalize the approaches to tuning these parameters. We use a convolution neural network for the CIFAR10 dataset. The architecture is summarized as follows:

- Convolutional input layer, 32 feature maps with a size of 3×3 and a rectifier activation function.
- Dropout layer at 20%.
- Convolutional layer, 32 feature maps with a size of 3×3 and a rectifier activation function.
- Max Pool layer with size 2×2.
- Convolutional layer, 64 feature maps with a size of 3×3 and a rectifier activation function.
- Dropout layer at 20%.
- Convolutional layer, 64 feature maps with a size of 3×3 and a rectifier activation function.
- Max Pool layer with size 2×2.
- Convolutional layer, 128 feature maps with a size of 3×3 and a rectifier activation function.
- Dropout layer at 20%.
- Convolutional layer,128 feature maps with a size of 3×3 and a rectifier activation function.
- Max Pool layer with size 2×2.
- Flatten layer.
- Dropout layer at 20%.
- Fully connected layer with 1024 units and a rectifier activation function.
- Dropout layer at 20%.
- Fully connected layer with 512 units and a rectifier activation function.
- Dropout layer at 20%.
- Fully connected output layer with 10 units and a softmax activation function.

We use another simple convolution neural network for the MNIST dataset. The architecture is summarized as follows:

- Convolutional layer with 30 feature maps of size 5×5.
- Pooling layer taking the max over 2*2 patches.
- Convolutional layer with 15 feature maps of size 3×3.
- Pooling layer taking the max over 2*2 patches.
- Dropout layer with a probability of 20%.
- Flatten layer.
- Fully connected layer with 128 neurons and rectifier activation.
- Fully connected layer with 50 neurons and rectifier activation.
- Output layer.

Experiments on AdaGrad

CIFAR10: AdaGrad has 2 main hyperparameters – lr and decay. Upon several exploratory experiments, it was found that epsilon has little to no difference on the accuracy and loss function of the model. The learning rate, lr was first tune to find the optimal value, that gives least loss (fig 2) and highest accuracy (fig1) for 30 epochs.

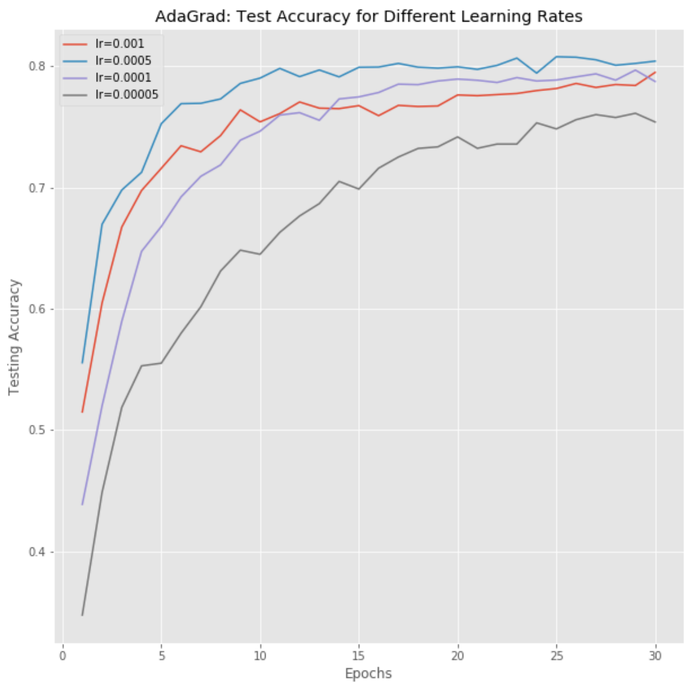


FIG 1

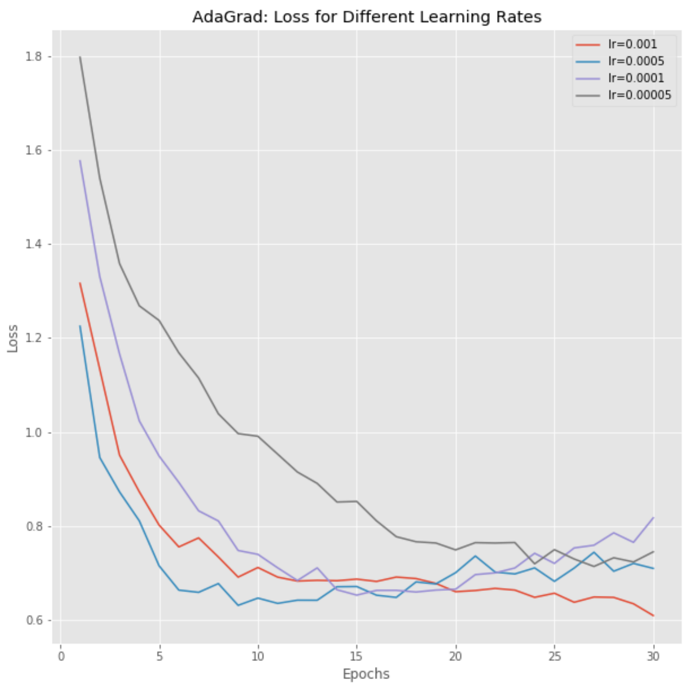


FIG 2

As can be seen in Fig 2. For all lr besides lr = 0.001, the loss goes up, indicating that the optimizer is trying to fight overfitting. Only lr = 0.001 seems to achieve a good loss and accuracy. However, as a sanity check, we also consider lr = 0.0005 which gave best accuracy. We move ahead with our analysis, by fixing decay = 0.00001 and testing out lr = 0.001 and lr = 0.0005.

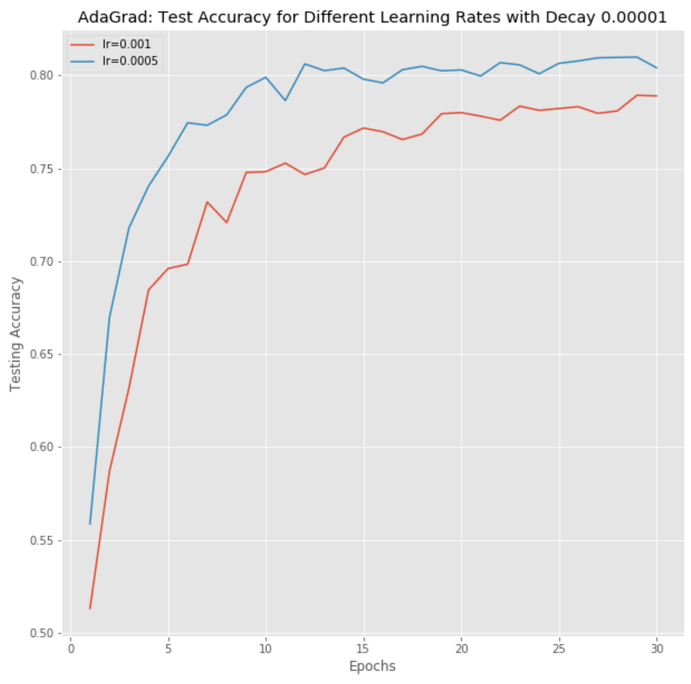


FIG 3

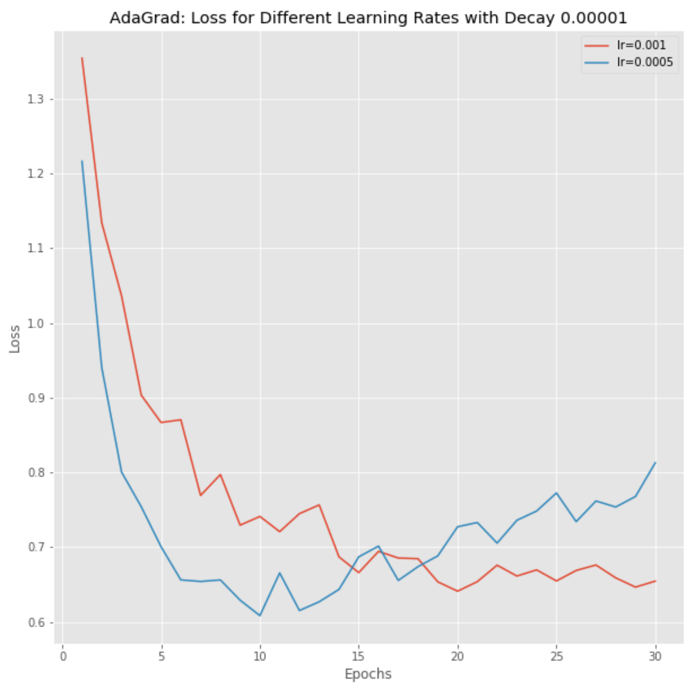


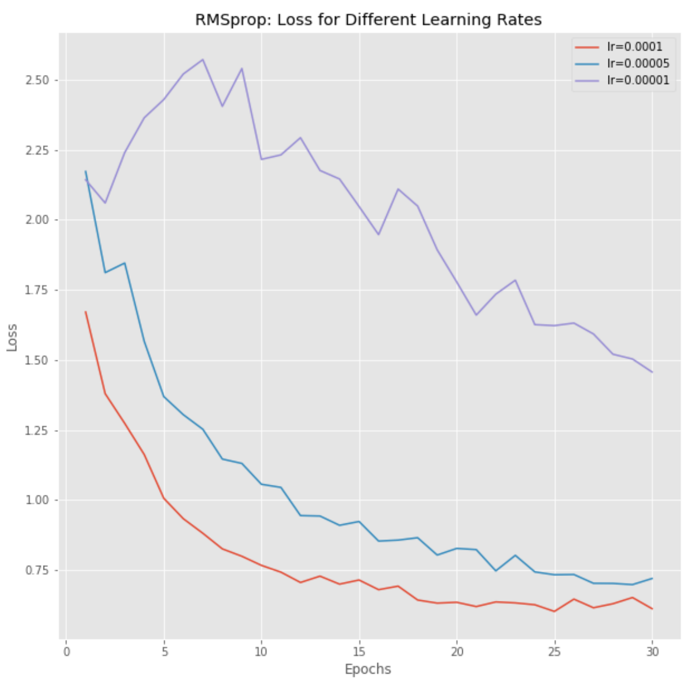
FIG 4

We see that loss reduces for lr=0.001 as opposed to lr=0.0005 in Fig 4. This clearly shows lr=0.001 with decay is the best option. Therefore with a careful choice of lr and decay an optimized model can be obtained. More details on approach towards algorithm shall be discussed in the results section.

MNIST: This dataset performed best with lr=0.001 and decay = 0.0001.

Experiments with RMSProp

RMSprop was tested first on learning rate, lr = [0.0001, 0.00005, 0.00001] for 30 epochs. All lower lr values didn't give a good accuracy or loss during exploratory analysis due to large step size. Lr = 0.0001 gave the best performance, as demonstrated in Fig 5 and Fig 6, so lr=0.0001 was chosen for the next steps of evaluation. We also consider lr = 0.00005, and test our performance of different rho values on these two lrs. The rho values are chosen according to the formula (1-1/k), where k=i*20 (roughly).



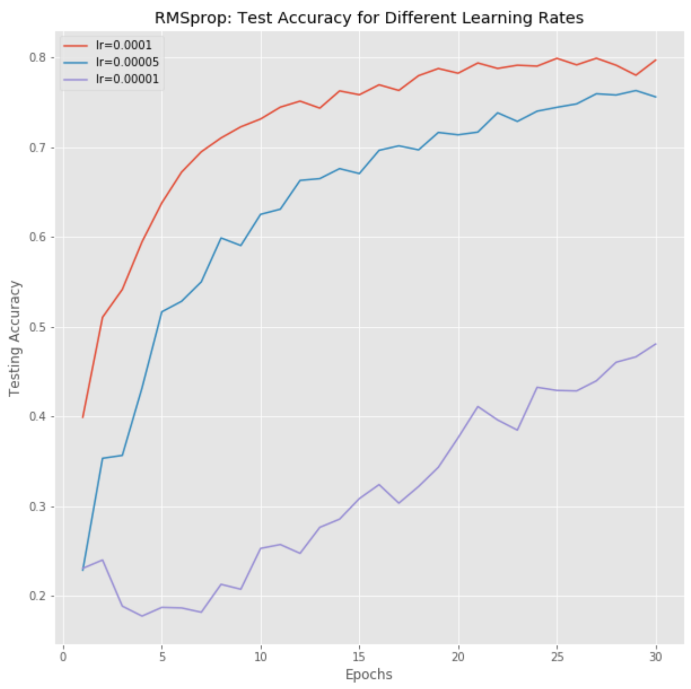
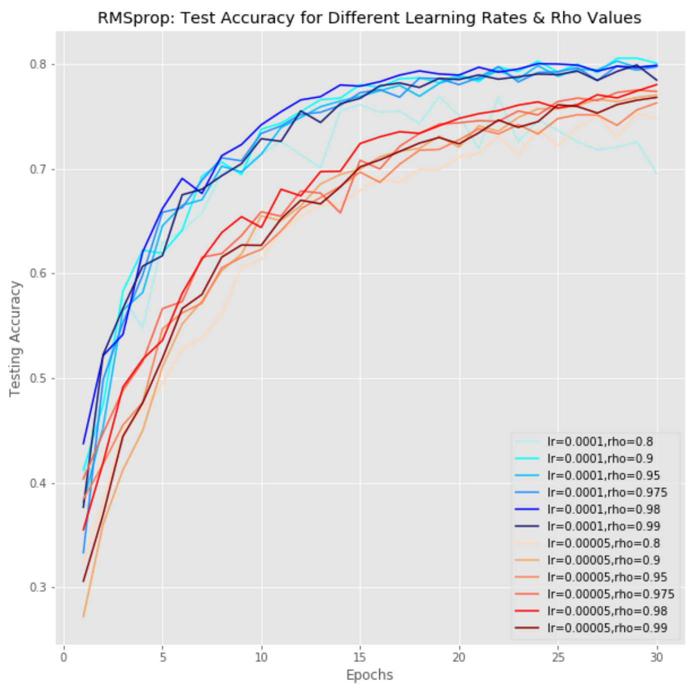


FIG 5

FIG 6

The following rho values were tested out: rho = [0.8, 0.9, 0.95, 0.975, 0.98, 0.99]. We observed that the following hyperparameter combinations do well:

Lr = 0.00005, rho = 0.98 | lr = 0.0001, rho = 0.9 | lr = 0.00005, rho = 0.98 | lr = 0.0001, rho = 0.9. This is observed on Fig 7 and Fig 8.



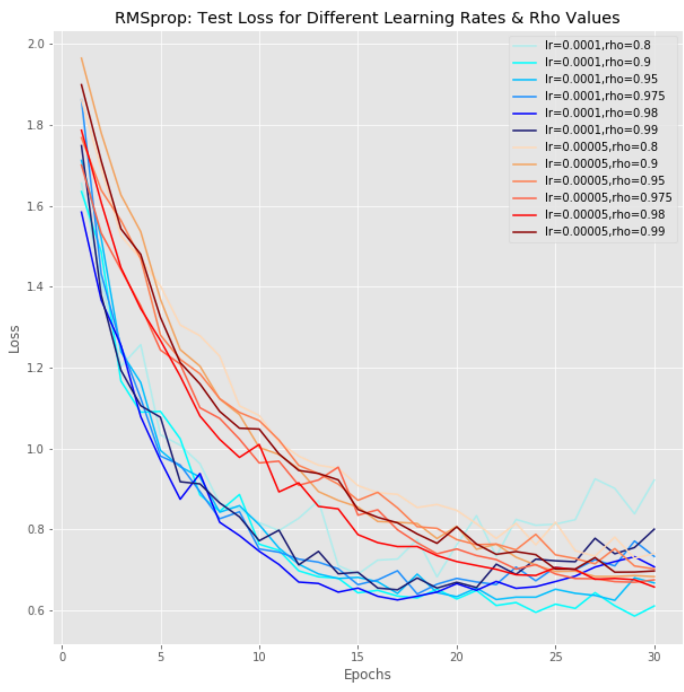


FIG 7

FIG 8

For the next steps we consider these selected hyperparameter combinations and run 50 epochs instead of 30. Fig 9 and Fig 10 present the performance. All of the cases have the loss function go up with after one stage which implies a possible overfitting case of training data.

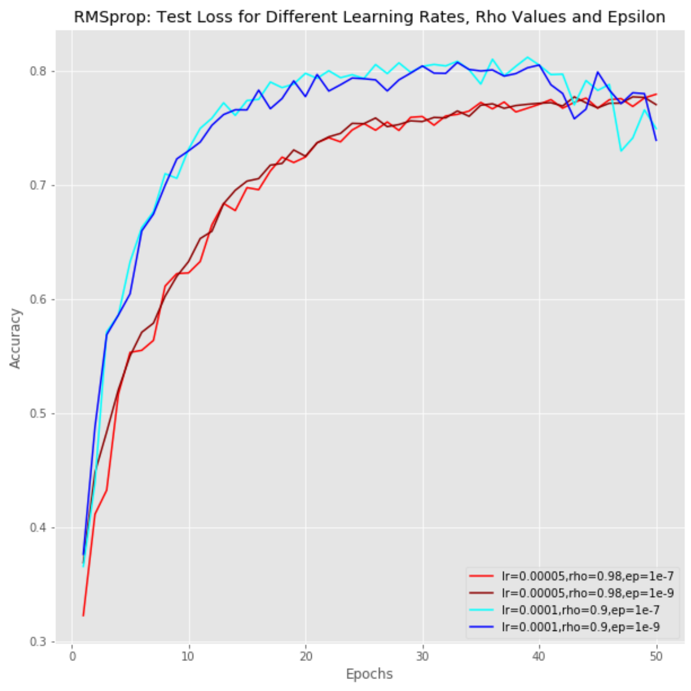


FIG 9

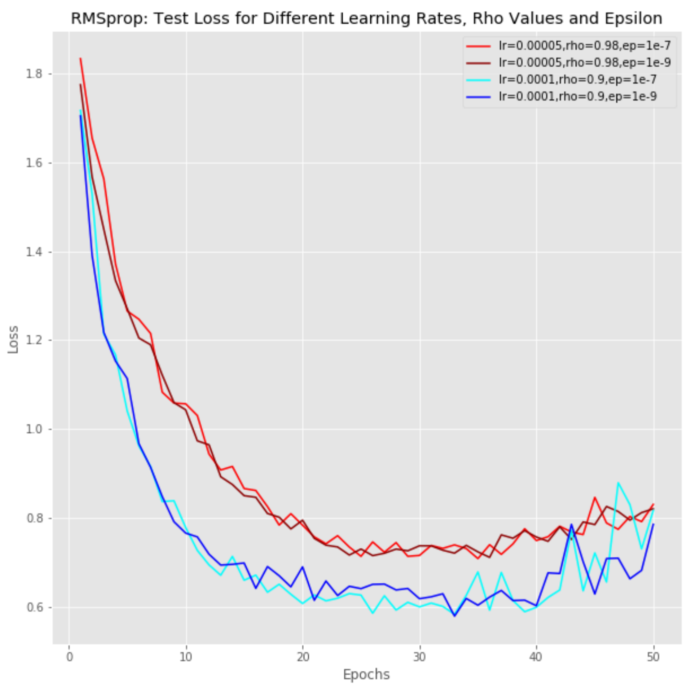


FIG 10

So, to compensate this we decay the learning rate finally as shown in Fig 11 and Fig 12. A stable descent is seen which indicates we have achieved a good generalization.

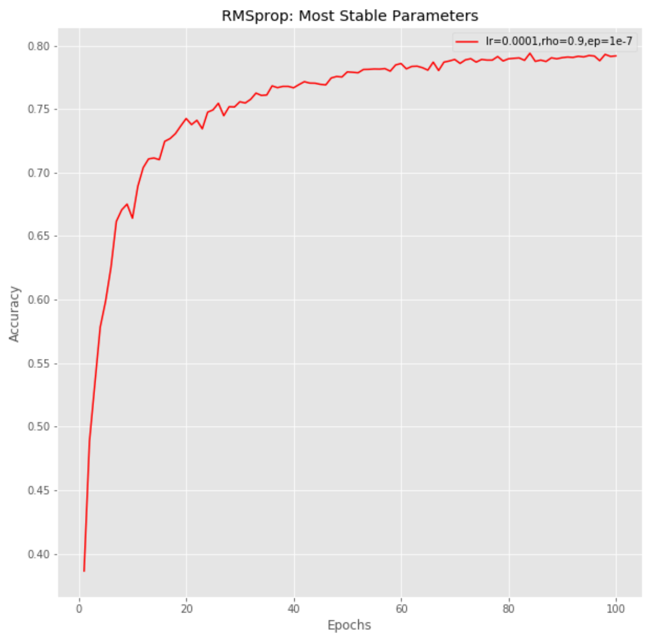


FIG 11

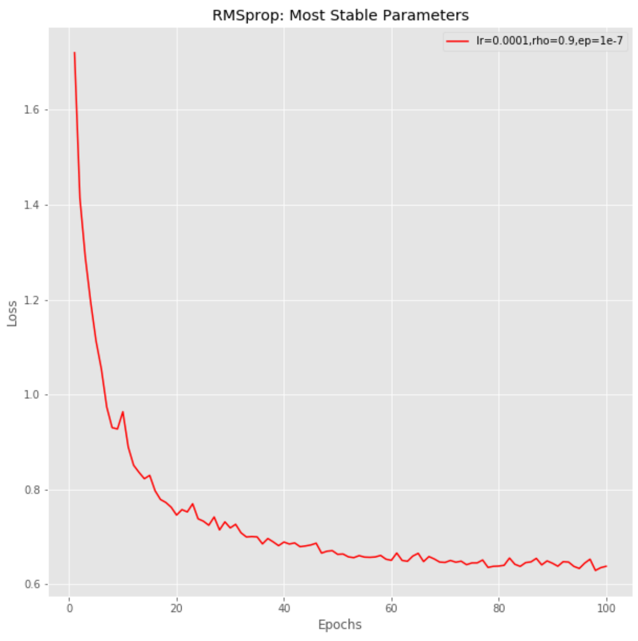


FIG 12

MNIST: The dataset performed best for lr = 0.001 and rho = 0.9

Experiments with Adam

The Adam Optimizer was tested on different lr values and gave the rest result for lr = 0.001. Fig 13 and Fig 14. However, a smaller lr = 0.0001 with decay of 0.0001 was chosen and tested on various beta1 values = [0.8,0.9,0.95,0.975,0.98,0.99]. The result of beta1 = 0.9, 0.98 and 0.99 were close and comparable as see on Fig 15, 16.

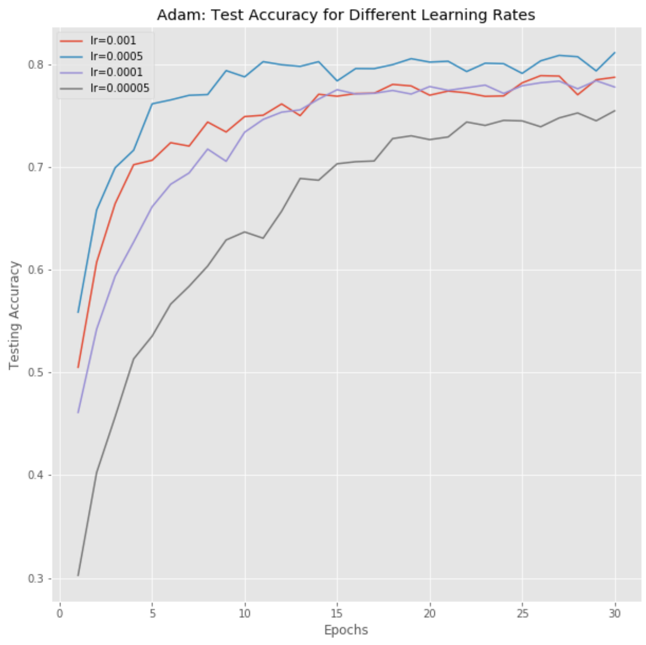


FIG 13

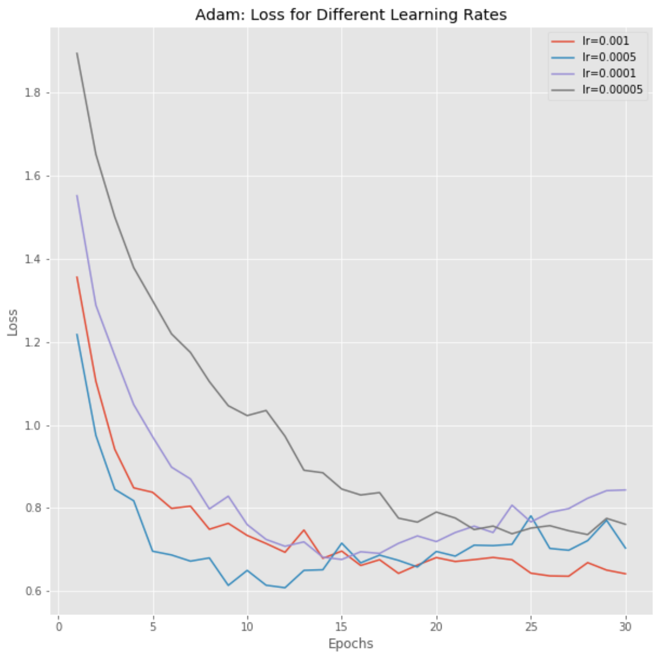


FIG 14

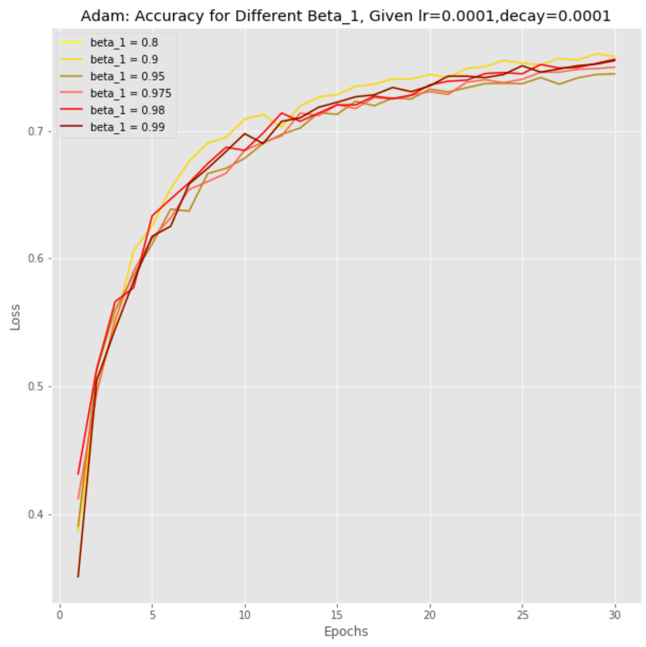


FIG 15

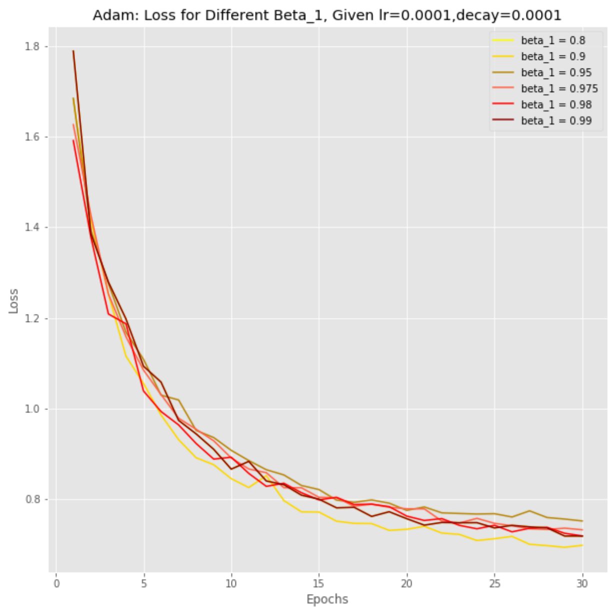


FIG 16

Given these comparable rho values, we test them out for different beta2 values as shown in Fig17 and Fig 18. Finally a stable version of the Adam hyperparameter combination is shown in Fig 19 and Fig 20. For lr = 0.0001, decay = 0.0002,

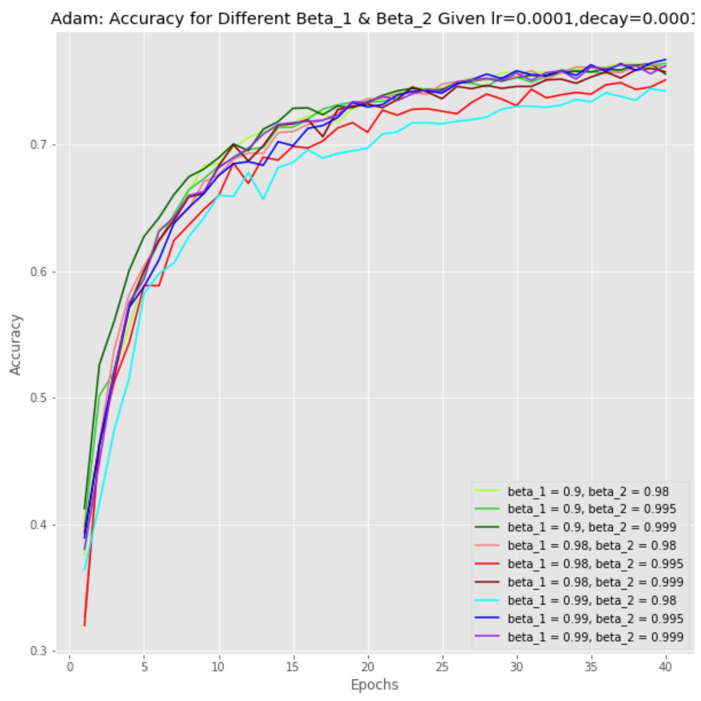


FIG 17

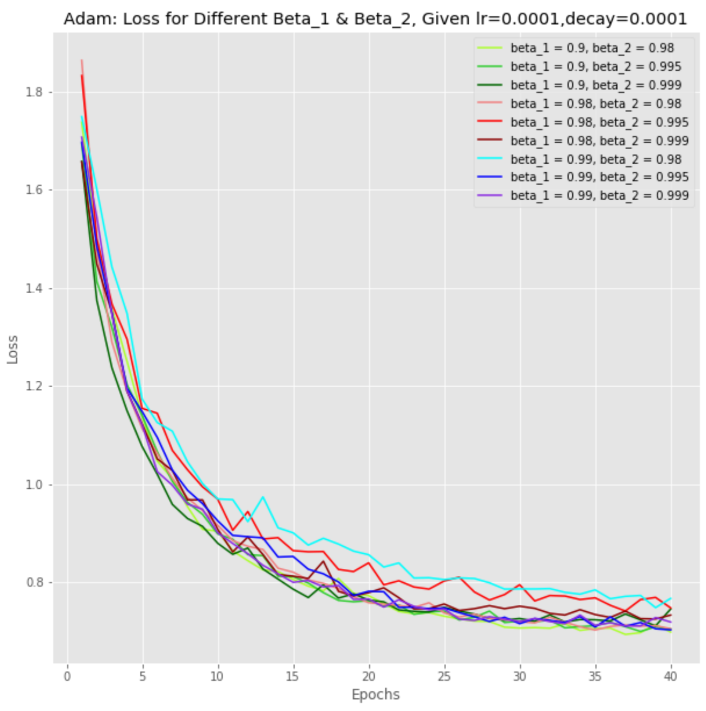


FIG 18

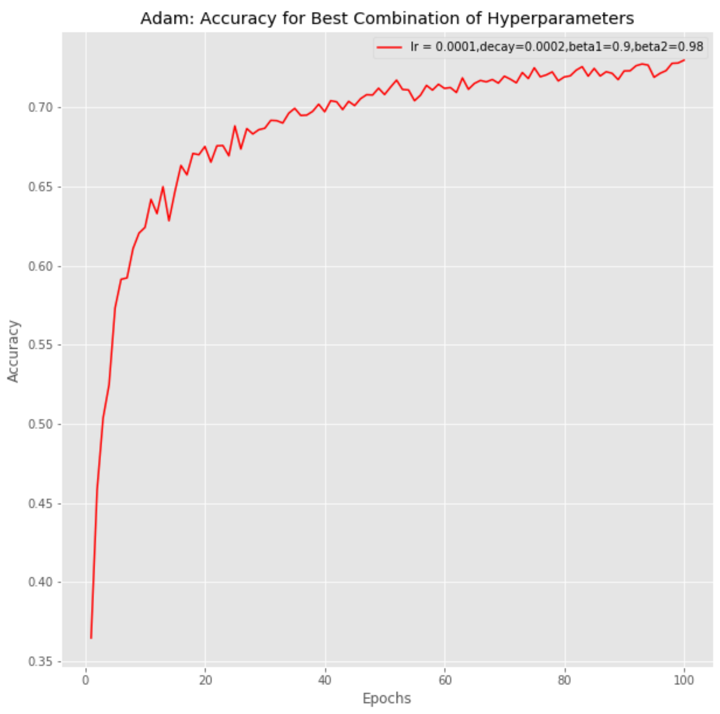


FIG 19

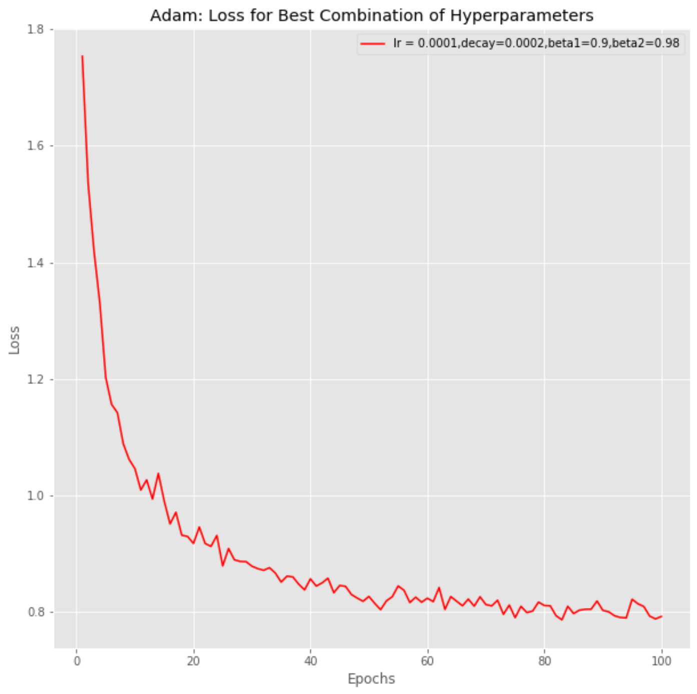


FIG 20

MNIST: beta1 = 0.98, beta2 = 0.995, lr = 0.001 gave best result.

Results and Quick Take-Aways

All these experiments really helped understand methodologies to choose based on our needs:-

AdaGrad

- Gives high accuracy on convex cases, but might overfit on training data
- Inefficient for non-convex case because of diminishing step-size
- Test on lr = 1e-3, 5e-4, 1e-4
- Include learning rate decay if necessary (this might be counterintuitive, since the step size is diminishing, but it sometimes works)
- Epsilon value can be fixed at 1e-7. Little to no effect
- Avoid AdaGrad if you want to experiment less. Might need a lot more training and tuning than other algorithms

RMSprop

- Includes a momentum-like step update
- Gives high accuracy too, but needs to iteratively tested on smaller epochs first to find the best parameter combination
- Test on lr = 1e-3, 5e-4, 1e-4, with default rho = 0.9
- Choose best lr and test rho for [0.9, 0.95, 0.975, 0.98], use (1-1/k) for k = 20,40,60,80,100
- Select the best rho and update your decay if necessary
- Epsilon value can be fixed at 1e-7. Little to no effect
- RMSprop converges faster than AdaGrad so if you want to quickly achieve convergence, RMSprop is better

Adam

- Suitable for convex & non-convex problems
- Might not give the best accuracy among the three algorithm, but performs fairly close to best accuracy for a wide array of hyperparameters
- Quite robust to poor choice of hyperparameters
- Ideally test on lr = 1e-3, 5e-4, 1e-4
- Test on beta1 & beta2 combinations together
- Epsilon value can be fixed at 1e-7. Little to no effect
- Experiment with beta1 higher than 0.9, be sure to try beta1 = [0.95, 0.975, 0.98]
- Beta2 must be close to 0.99 as possible. 0.98, 0.995 have shown better performance than 0.999. So experiment with them as well

While Adam might be the choice of most Data Scientists, one observation I made is it doesn't give the best accuracy compared to properly tuned RMSprop or properly tuned AdaGrad. But because tuning is a costly process Adam is preferred since it performs almost close to the best accuracy and it is quite robust to poor choices of hyperparameters. Even Adam could be made more optimized, but it is a costly process given the number of parameters it has to tune.