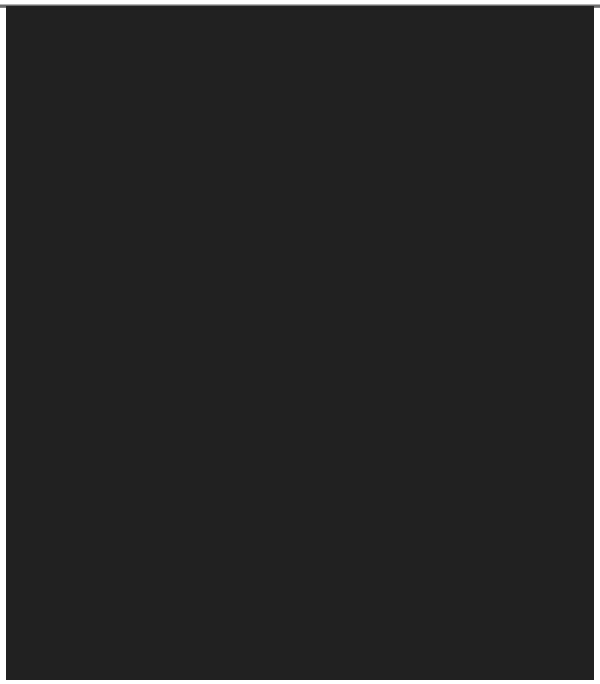


DL-Ops

Lab Assignment 3 - Report

Debonil Ghosh
Roll No: M21AIE225
Executive MTech
Artificial Intelligence
Indian Institute of Technology, Jodhpur

Problem Statement:

Question 1 [30 Marks]

Use ResNet18 pre-trained on ImageNet (Pre-trained models can be found online). Finetune the model on X dataset for classification task and plot curves for training loss and training accuracy. Report the final top-5 test accuracy. Perform the above task with any 3 optimizers from the following list.

1. Adam
2. Adagrad
3. Adadelta
4. RMSprop

X = STL 10, if last digit of your roll no. is odd

X = CIFAR100, if last digit of your roll no. is even

In the report, analyze the results of each optimizer and mention the reason why one is better than the other, or why not. Further, for each optimizer, you need to vary the value of attributes and show its effect on the overall performance. Provide reasons for your observations in the report. (For ex - In Adam you can change the values of attributes like beta_1, beta_2, epsilon, momentum, weight_decay etc.)

Note - Use of TensorFlow is not permitted, and marks shall not be given for the same.

Given Dataset:

STL 10 (as Roll is M21AIE225)

Training Set: 5000 (500×10 pre-defined folds) Colour (3 channels) 96×96 Images

Testing Set: 8000 (800×10 pre-defined folds) Colour (3 channels) 96×96 Images

Classes: 10 classes: airplane, bird, car, cat, deer, dog, horse, monkey, ship, truck

Data Samples:

Resnet18 Model Architecture:

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
    (downsample): Sequential(
      (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(layer4): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): BasicBlock(
    (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
(fc): Linear(in_features=512, out_features=10, bias=True)
)
```

- Augmented last fully connected layer as per problem given.
- Pretrained on ImageNet Data set.
- Loss Function: Cross Entropy Loss
- No of epochs used to train model for all optimizers: **10**
- Base Learning Rate: **0.001**

1. ADAM (Adaptive Moment Estimation) Optimizer:

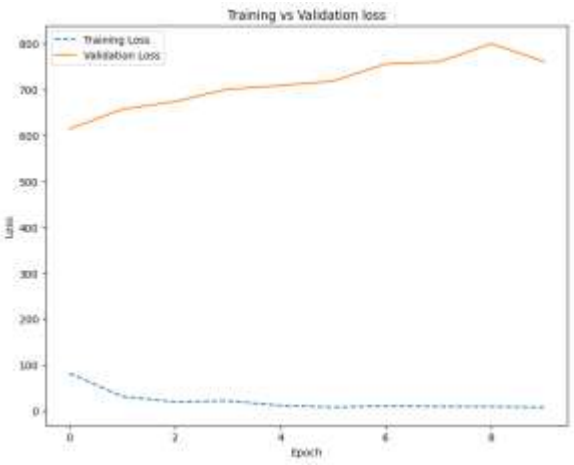
Adam is an optimization algorithm that computes adaptive learning rates for individual weights during training. It maintains a running estimate of the first and second moments of the gradients of the weights, denoted by m and v, respectively. These estimates are initialized as vectors of zeros and updated using a combination of a moving average and bias correction. The optimizer computes bias-corrected estimates of m and v as m_hat = m / (1 - β1^t) and v_hat = v / (1 - β2^t), where β1 and β2 are hyperparameters controlling the exponential decay rates of the estimates, and t is the iteration number. Finally, the optimizer updates the weights using a learning rate α and the bias-corrected estimates of m and v as follows: θ = θ - α * m_hat / (sqrt(v_hat) + ε), where ε is a small constant added to the denominator to avoid division by zero.

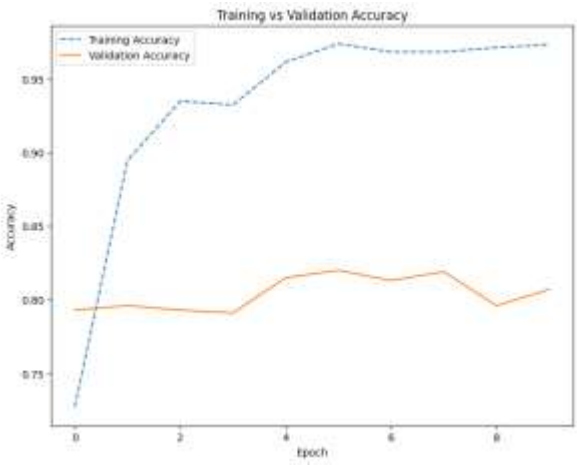
Training and Observations:

Model Training Log:

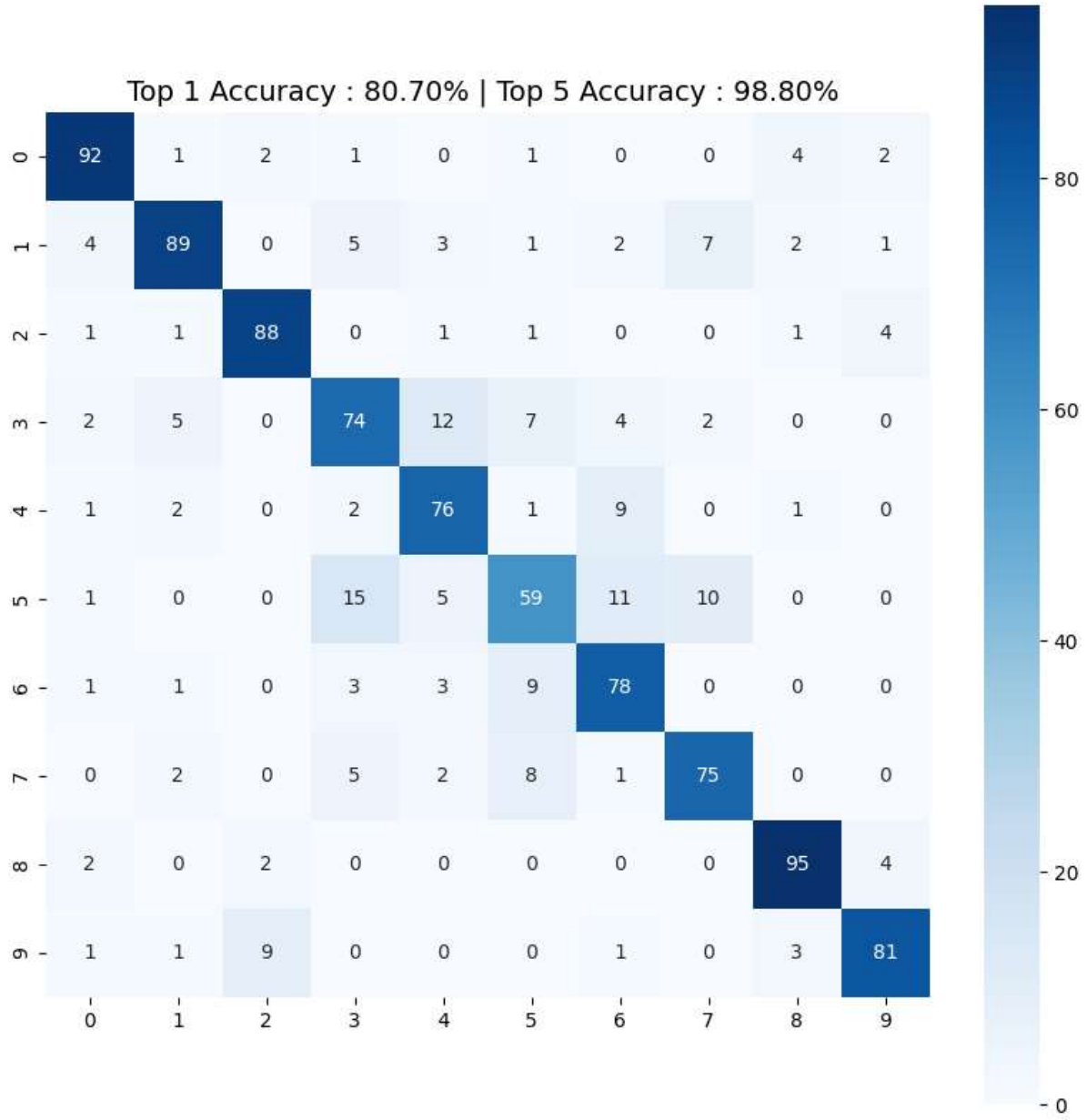
Epoch: 1 Training Loss: 82.078650, Test Loss: 614.433289, Training acc: 0.727000, Test acc: 0.793000,
Epoch: 2 Training Loss: 31.350838, Test Loss: 657.145262, Training acc: 0.894600, Test acc: 0.796000,
Epoch: 3 Training Loss: 19.408739, Test Loss: 673.955321, Training acc: 0.934800, Test acc: 0.793000,
Epoch: 4 Training Loss: 21.767490, Test Loss: 700.918734, Training acc: 0.932200, Test acc: 0.791000,
Epoch: 5 Training Loss: 11.681993, Test Loss: 709.099174, Training acc: 0.961400, Test acc: 0.815000,
Epoch: 6 Training Loss: 7.955046, Test Loss: 718.438625, Training acc: 0.973800, Test acc: 0.820000,
Epoch: 7 Training Loss: 10.227830, Test Loss: 755.967617, Training acc: 0.968200, Test acc: 0.813000,
Epoch: 8 Training Loss: 9.303071, Test Loss: 760.519803, Training acc: 0.968200, Test acc: 0.819000,
Epoch: 9 Training Loss: 9.134530, Test Loss: 800.109386, Training acc: 0.971200, Test acc: 0.796000,
Epoch: 10 Training Loss: 7.732154, Test Loss: 761.471570, Training acc: 0.973200, Test acc: 0.807000,

Model Training Losses and Accuracies:





Confusion Matrix and Accuracy:



Top 1 Accuracy: **80.700%**

Top 5 Accuracy: **98.8%**

Class wise Accuracy Score:

[0.89320388 0.78070175 0.90721649 0.69811321 0.82608696 0.58415842

0.82105263 0.80645161 0.9223301 0.84375   ]

2. Adagrad Optimizer:

This optimizer changes the learning rate. It changes the learning rate **'η'** for each parameter and at every time step **'t'.** It's a type second order optimization algorithm. It works on the derivative of an error function.

$$g_{t,i} = \nabla_\theta J(\theta_{t,i}),$$

A derivative of loss function for given parameters at a given time t.

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

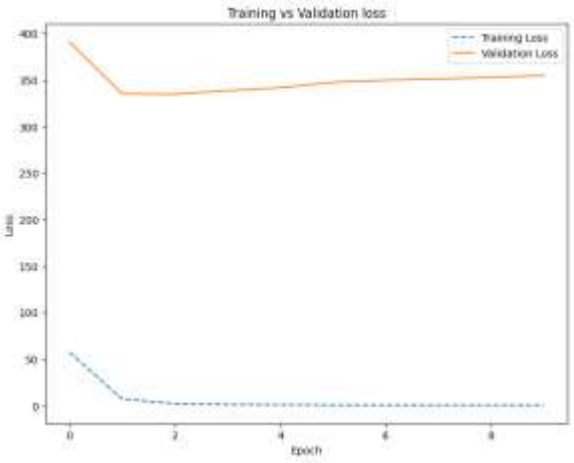
Update parameters for given input i and at time/iteration t
η is a learning rate which is modified for given parameter **θ(i)** at a given time based on previous gradients calculated for given parameter **θ(i).**

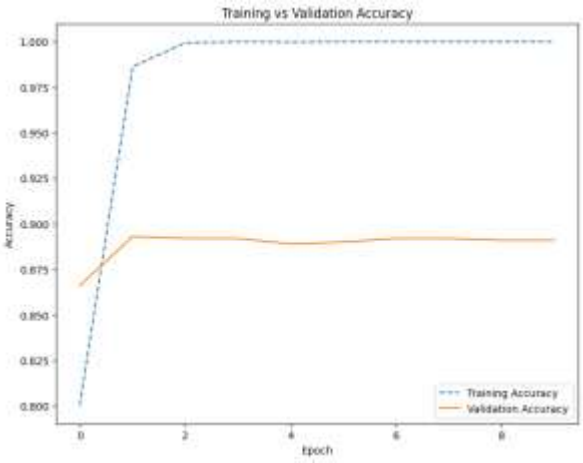
Training and Observations:

Model Training Log:

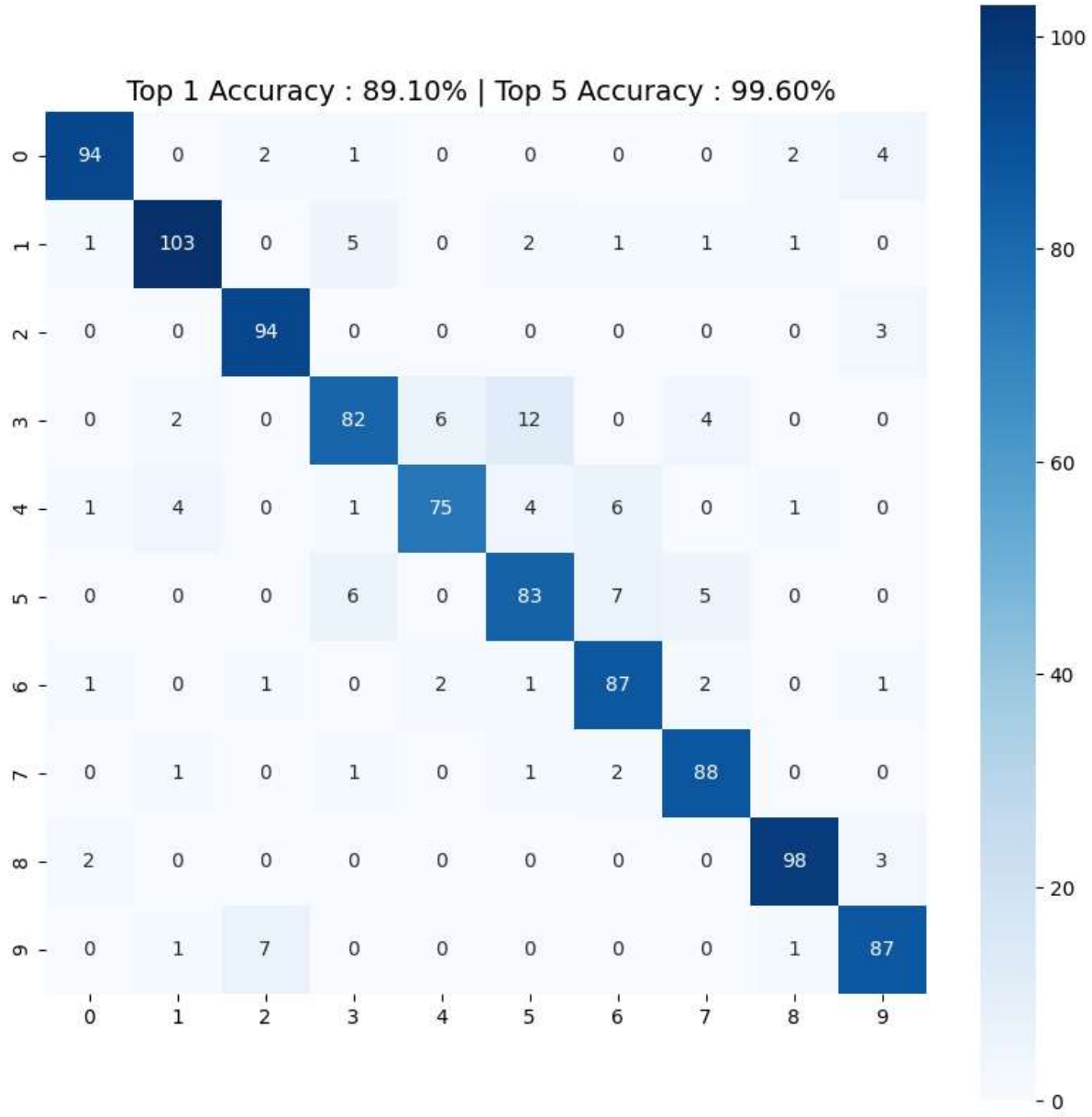
Epoch: 1 Training Loss: 57.339447, Test Loss: 391.216218, Training acc: 0.800400, Test acc: 0.866000,
Epoch: 2 Training Loss: 7.036367, Test Loss: 335.199594, Training acc: 0.986200, Test acc: 0.893000,
Epoch: 3 Training Loss: 2.061597, Test Loss: 334.692627, Training acc: 0.999400, Test acc: 0.892000,
Epoch: 4 Training Loss: 1.033667, Test Loss: 338.609815, Training acc: 1.000000, Test acc: 0.892000,
Epoch: 5 Training Loss: 0.771824, Test Loss: 341.696292, Training acc: 0.999800, Test acc: 0.889000,
Epoch: 6 Training Loss: 0.554433, Test Loss: 347.637147, Training acc: 1.000000, Test acc: 0.890000,
Epoch: 7 Training Loss: 0.439116, Test Loss: 349.825233, Training acc: 1.000000, Test acc: 0.892000,
Epoch: 8 Training Loss: 0.407432, Test Loss: 351.264626, Training acc: 1.000000, Test acc: 0.892000,
Epoch: 9 Training Loss: 0.375267, Test Loss: 352.931619, Training acc: 1.000000, Test acc: 0.891000,
Epoch: 10 Training Loss: 0.298735, Test Loss: 354.932994, Training acc: 1.000000, Test acc: 0.891000,

Model Training Losses and Accuracies:





Confusion Matrix and Accuracy:



Top 1 Accuracy: **89.100%**

Top 5 Accuracy: **99.6%**

Classwise Accuracy Score:

[0.91262136 0.90350877 0.96907216 0.77358491 0.81521739 0.82178218
0.91578947 0.94623656 0.95145631 0.90625]

3. Adadelta Optimizer:

It is an extension of AdaGrad which tends to remove the decaying learning Rate problem of it. Instead of accumulating all previously squared gradients, **Adadelta** limits the window of accumulated past gradients to some fixed size **w**. In this exponentially moving average is used rather than the sum of all the gradients.

$$\mathbf{E[g^2](t)=\gamma.E[g^2](t-1)+(1-\gamma).g^2(t)}$$

We set **γ** to a similar value as the momentum term, around 0.9.

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

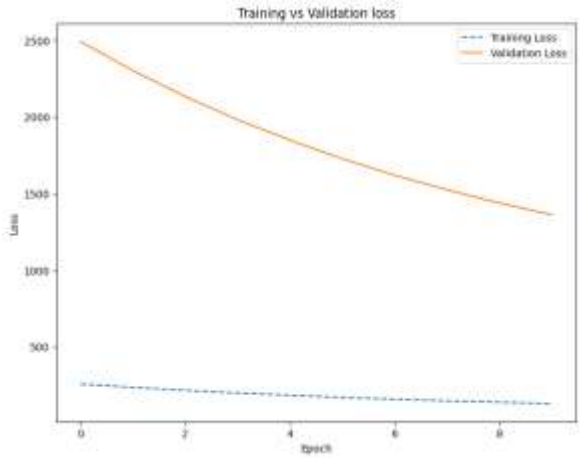
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\mathbb{E}[g^2]_t + \epsilon}} \cdot g_t.$$

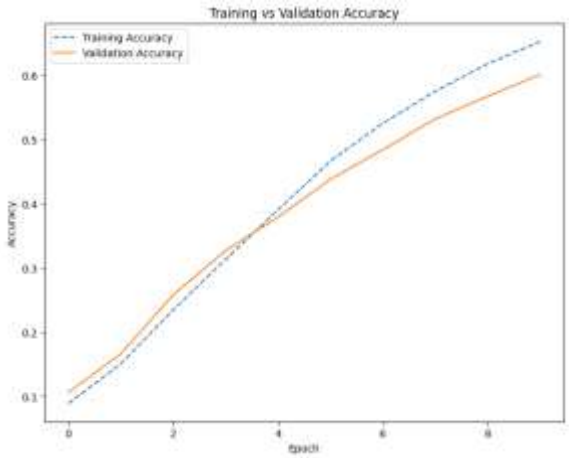
Training and Observations:

Model Training Log:

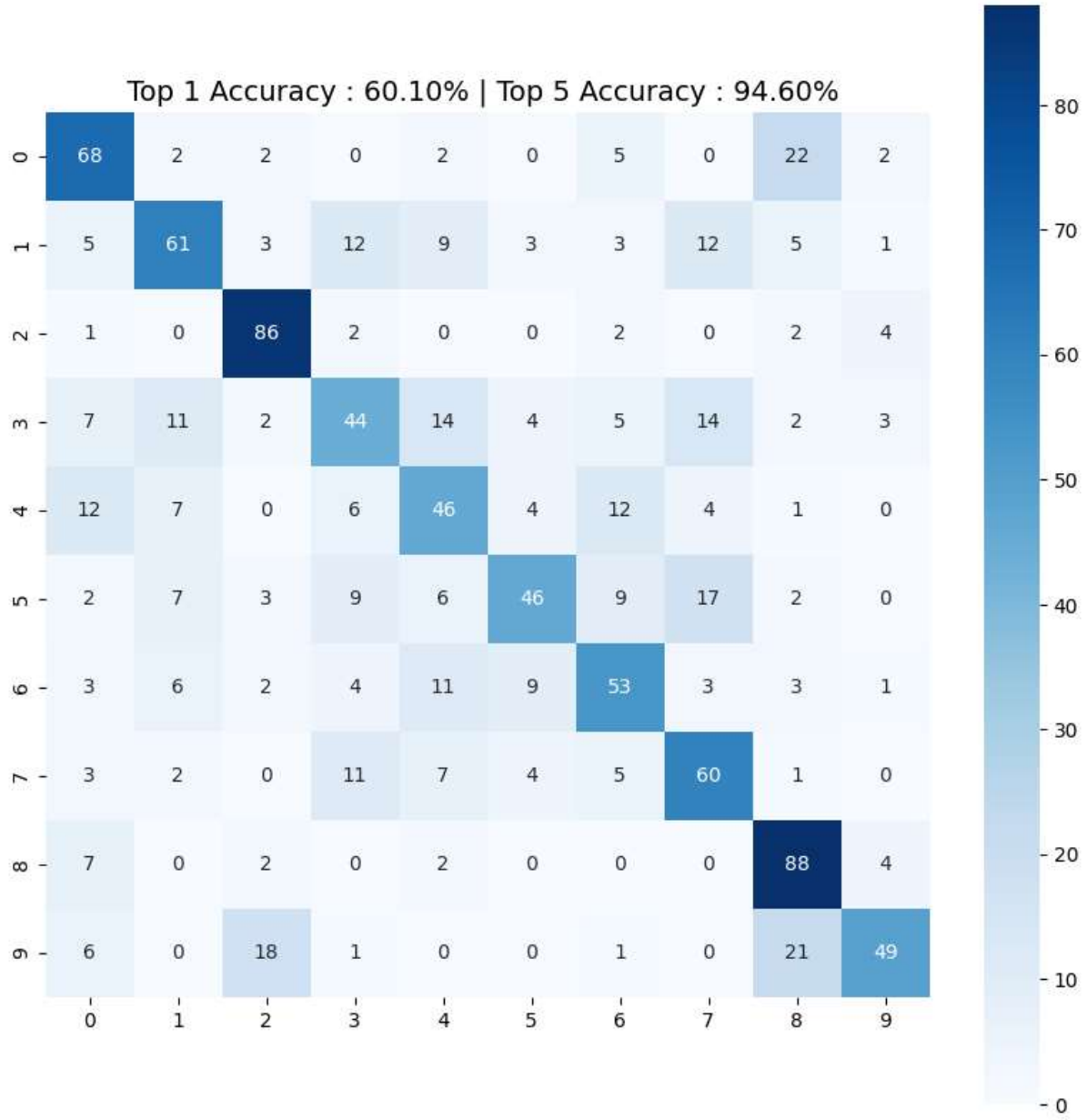
Epoch: 1 Training Loss: 256.169827, Test Loss: 2491.599083, Training acc: 0.088800, Test acc: 0.106000,
Epoch: 2 Training Loss: 234.733407, Test Loss: 2301.831961, Training acc: 0.150600, Test acc: 0.166000,
Epoch: 3 Training Loss: 215.314672, Test Loss: 2132.591963, Training acc: 0.234600, Test acc: 0.258000,
Epoch: 4 Training Loss: 198.007964, Test Loss: 1981.519461, Training acc: 0.313200, Test acc: 0.327000,
Epoch: 5 Training Loss: 183.040474, Test Loss: 1846.614122, Training acc: 0.390400, Test acc: 0.379000,
Epoch: 6 Training Loss: 169.069120, Test Loss: 1725.814939, Training acc: 0.466200, Test acc: 0.438000,
Epoch: 7 Training Loss: 157.340216, Test Loss: 1618.575931, Training acc: 0.525200, Test acc: 0.484000,
Epoch: 8 Training Loss: 146.545958, Test Loss: 1522.536397, Training acc: 0.575000, Test acc: 0.532000,
Epoch: 9 Training Loss: 137.393952, Test Loss: 1436.814308, Training acc: 0.617800, Test acc: 0.567000,
Epoch: 10 Training Loss: 129.242647, Test Loss: 1360.561848, Training acc: 0.652200, Test acc: 0.601000,

Model Training Losses and Accuracies:





Confusion Matrix and Accuracy:



Top 1 Accuracy: **60.100**%

Top 5 Accuracy: **94.6**%

Classwise Accuracy Score:

[0.66019417 0.53508772 0.88659794 0.41509434 0.5 0.45544554

0.55789474 0.64516129 0.85436893 0.51041667]

3. RMSprop Optimizer:

RMSprop is an optimization algorithm that maintains a running average of the squared gradients of each weight. It uses this average to scale the learning rate for each weight. Specifically, RMSprop maintains a running estimate of the second moment of the gradients, denoted by v, for each weight. The optimizer then computes an adaptive learning rate α for each weight as $\alpha = \eta / (\text{sqrt}(v) + \varepsilon)$, where η is the initial learning rate, ε is a small constant added to the denominator to avoid division by zero, and sqrt denotes the element-wise square root. Finally, the optimizer updates the weights using the adaptive learning rate and the gradient of the loss function with respect to the weights.

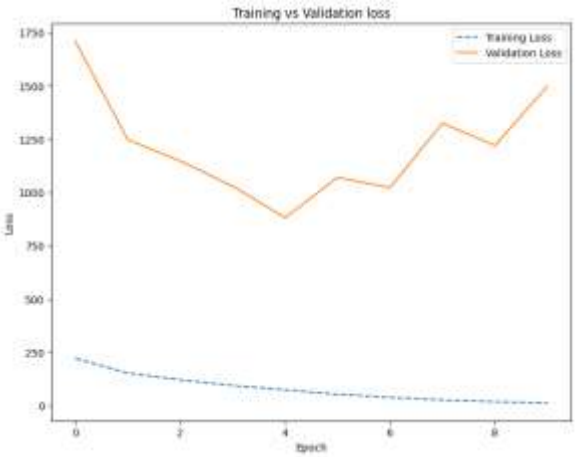
Compared to the Adagrad optimizer, RMSprop restricts the accumulation of past squared gradients to a fixed window, which makes it more suitable for long-term optimization. RMSprop has been shown to work well in practice for a wide range of problems and network architectures.

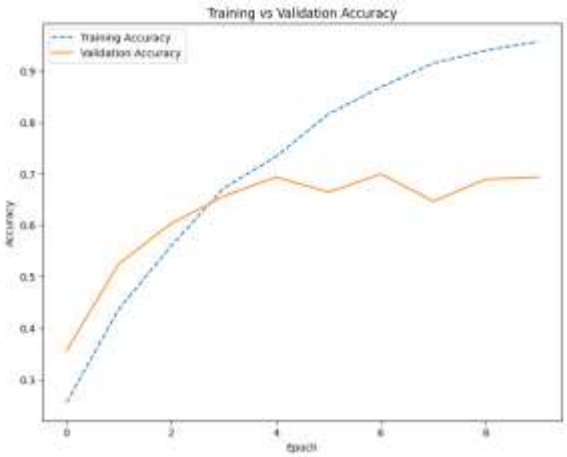
Training and Observations:

Model Training Log:

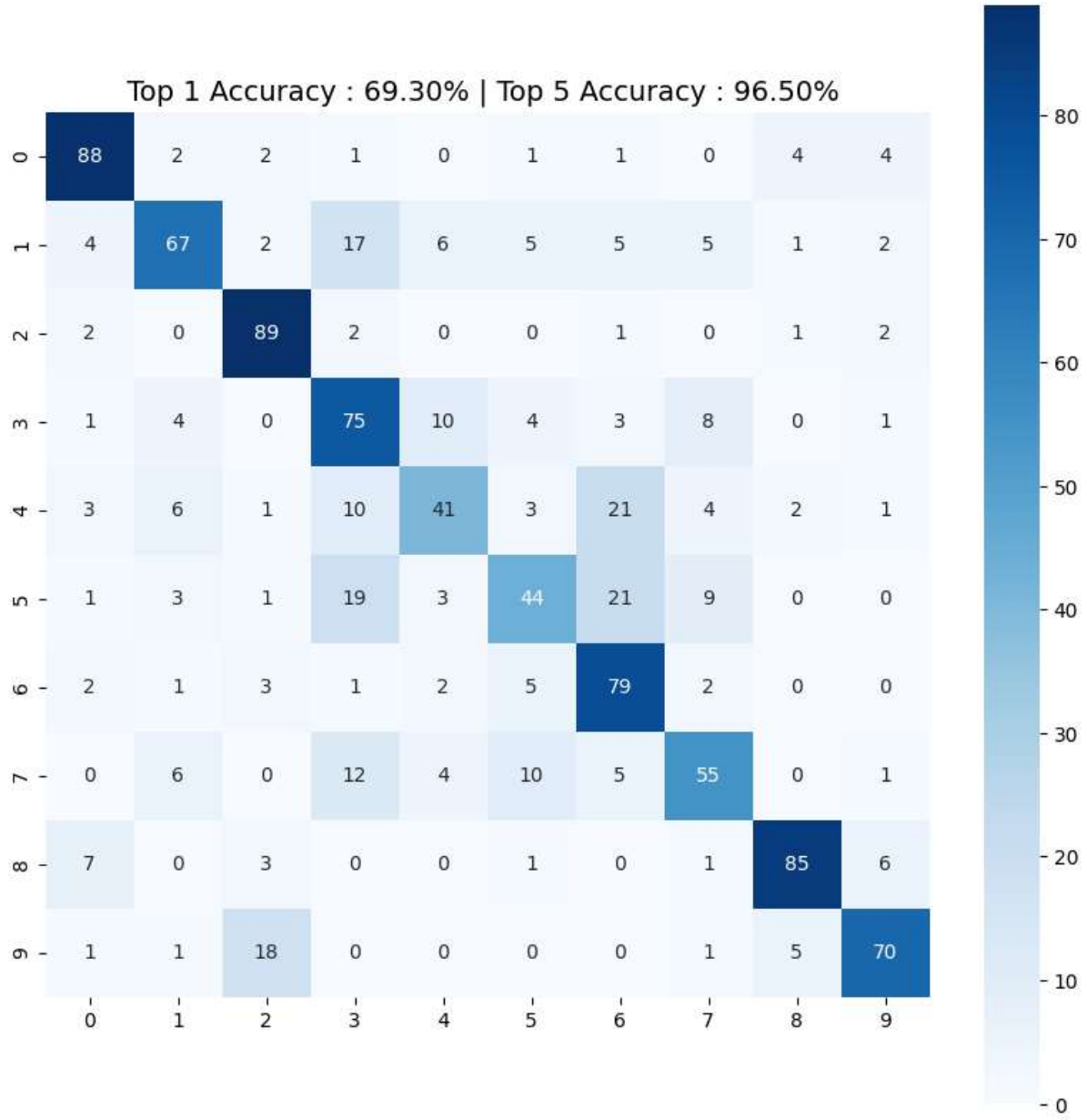
Epoch: 1 Training Loss: 221.198801, Test Loss: 1708.335161, Training acc: 0.255200, Test acc: 0.356000,
Epoch: 2 Training Loss: 152.504709, Test Loss: 1247.624278, Training acc: 0.436800, Test acc: 0.525000,
Epoch: 3 Training Loss: 120.680466, Test Loss: 1147.314548, Training acc: 0.559800, Test acc: 0.603000,
Epoch: 4 Training Loss: 94.099264, Test Loss: 1029.482007, Training acc: 0.671600, Test acc: 0.656000,
Epoch: 5 Training Loss: 73.839946, Test Loss: 881.439209, Training acc: 0.733200, Test acc: 0.693000,
Epoch: 6 Training Loss: 52.886699, Test Loss: 1070.078611, Training acc: 0.815600, Test acc: 0.664000,
Epoch: 7 Training Loss: 38.459189, Test Loss: 1022.760391, Training acc: 0.867800, Test acc: 0.699000,
Epoch: 8 Training Loss: 26.465256, Test Loss: 1324.641109, Training acc: 0.914400, Test acc: 0.646000,
Epoch: 9 Training Loss: 17.766749, Test Loss: 1220.268846, Training acc: 0.938800, Test acc: 0.689000,
Epoch: 10 Training Loss: 13.847231, Test Loss: 1498.154998, Training acc: 0.956400, Test acc: 0.693000,

Model Training Losses and Accuracies:





Confusion Matrix and Accuracy:



Top 1 Accuracy: **69.300**%

Top 5 Accuracy: **96.5**%

Classwise Accuracy Score:

[0.85436893 0.5877193 0.91752577 0.70754717 0.44565217 0.43564356

0.83157895 0.59139785 0.82524272 0.72916667]

Analysis of obtained results:

Adam reached optima very fast, then overfitting into train data. From the beginning of the training, validation loss keeps increasing and training loss was decreasing. After training produced a good result of top 1 Accuracy of 80.70% and top 5 Accuracy of 98.8%.

Adagrad also achieved fast and did not much change for last few epochs with an excellent performance of 89.1% top 1 and 99.6 top 5 accuracy.

For Adadelta, it was reducing loss steadily, though steep ness of the curve indicates it needs more epochs to converge to a good accuracy score. With present epoch count of 10, it reached a decent top 5 accuracy level of 94% (top 1 of 60%).

RMSProp started overfitting after fourth epoch, finally training accuracy reached 95% but validation accuracy reached only 69% . Top 5 accuracy reached 96.5.

From the observations above, its is found that **Adagrad is outperforming** all other three in terms of accuracy and stability of learning process, both. It adapts the learning rate for each weight based on the historical sum of squared gradients of that weight. It allows different weights to have different learning rates and for this reason it is particularly useful for sparse data. However, it can accumulate the sum of gradients indefinitely, which may lead to a very small or even zero learning rate as training progresses.

All four optimization algorithms have their strengths and weaknesses, and the choice of which algorithm to use depends on the specific problem and the neural network architecture being trained. Adam and RMSprop are often considered to be more robust and computationally efficient than Adagrad and Adadelta. However, Adagrad and Adadelta can be useful for handling sparse data and for reducing the number of hyperparameters that need to be tuned.

References:

1. Lectures of DL Ops class by Prof Pratik Majumder
2. Implementation Session by Teaching Assistants
3. Blogs from Internet