3/22/2023

Deep Learning Assignment-3

Optimizers

Presented by:

1. R. Ramakrishna Kashyap – T22101
2. Anuj Kumar Shukla – T22103
3. Nishant Gupta- T22221

Index

1. Introduction

Backpropagation is a common technique used in neural networks for training models to make predictions. It works by adjusting the weights and biases of the network to minimize the difference between the predicted output and the actual output. This process involves calculating the gradient of the loss function with respect to the model parameters, and then updating those parameters in the direction of the negative gradient.

Optimizers are algorithms that are used to adjust the learning rate and direction of the gradient updates during backpropagation. They help to speed up the training process, avoid local minima, and improve the generalization of the model. Some of the most common optimizers used in deep learning include stochastic gradient descent, Batch Gradient Descent, Momentum based Gradient Descent, AdaGrad, RMSProp and Adam.

In this assignment, we will be training an FCNN using a variety of different optimizers for backpropagation. Our aim is to compare the performance of different optimizers and evaluate their strengths and weaknesses. By doing so, We hope to gain a better understanding of how optimizers can be used to improve the efficiency and accuracy of neural network training.

2. Fully Connected Neural Network

It is a type of neural network architecture where all the neurons in one layer are connected to all the neurons in the next layer. This type of neural network is also known as a "dense" network.

Architecture of a FCNN:

<u>Input layer</u> – Linear Neuron

<u>Hidden layer</u> – Sigmoidal Neuron

<u>Output layer</u> – Sigmoidal Neuron (For pattern classification task) or Linear Neuron (For Regression task)

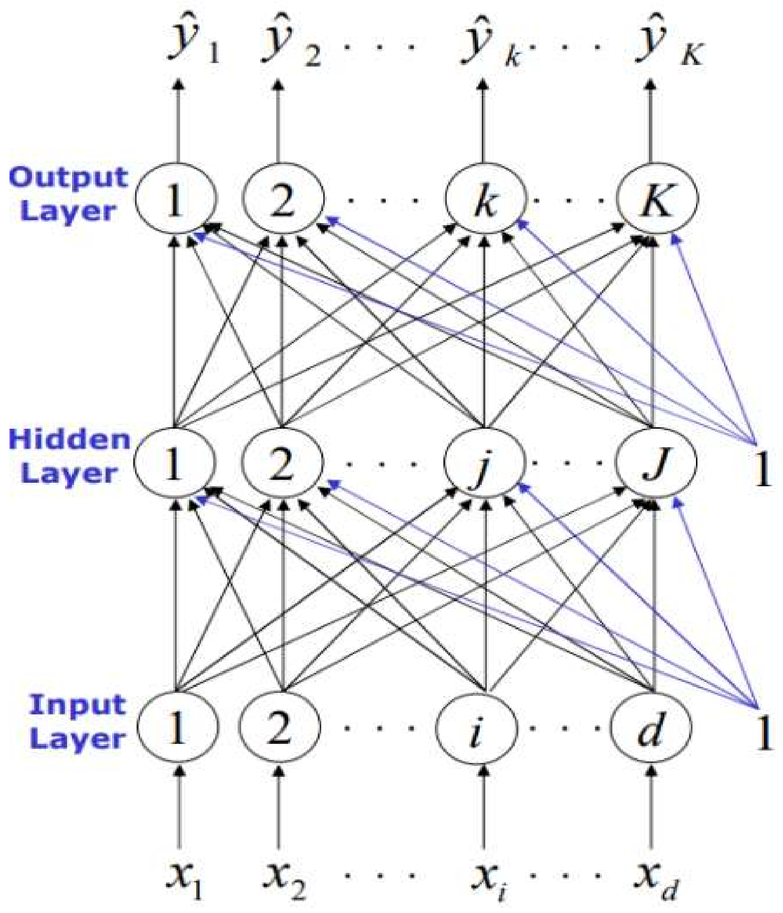

$\hat{y}_1$ $\hat{y}_2$ · · · $\hat{y}_k$ · · · $\hat{y}_K$
Output Layer
1 2 · · · k · · · K
Hidden Layer
1 2 · · · j · · · J 1
Input Layer
1 2 · · · i · · · d 1
x_1 x_2 · · · x_i · · · x_d

Figure 1: FCNN

Image source:
https://students.iitmandi.ac.in/moodle/mod/resource/view.php?id=47386

3. Different Optimizers for Backpropagation Algorithms

a) Stochastic gradient descent (SGD) algorithm

In SGD we present a single example, compute instantaneous error, compute the gradient, and update the weights based on instantaneous gradient values.

At mth iteration: Weights are updated after the presentation of each pattern.

Key Points:

- Faster than Batch mode.
- Here the update is based on the approximate gradient of the loss (loss is estimating based on a single data point)
- There will not be any guarantee that each step will decrease the loss.
- We see many oscillations, because we are making greedy decisions. Each point is trying to push the parameters in a direction most favourable to it.

b) Batch gradient descent algorithm (vanilla gradient descent)

Gradient values are collected over an entire epoch and the weights are updated at once. Weight change is decided using single resultant (local) gradients value.

Update rule is based on true gradient of the loss where true gradient is sum of the gradients of the losses corresponding to each data point.

Key Points:

- Since update is based in true gradient, each step guarantees that the loss will decrease.
- If we have huge data points, this algorithm makes huge calculations to make one update.
- Leads to very slow convergence.

c) SGD with momentum (generalized delta rule)

Gradient descent algorithm takes a lot time to navigate regions having gentle slope because here the gradients is very small. We can introduce a momentum term in weight update procedure which increases the learning rate while maintaining the stability.

Weights are updated not just based on current gradient; it is also based on the history of weight change.

Weight update is based on two steps:

- First step is on the history
- Second step is on the gradient at that iteration

Key Points:

- In gentle slope region, this algorithm takes a large step because the momentum carries it along.
- If the minima is sufficiently deep then this algorithm oscillates in and out of the minimum valley as the momentum carries it out of the valley.
- Takes a lot of U-turns before finally converging.
- Although it is taking a lot of U-turns, it is faster than Normal Gradient Descent techniques.

d) SGD with momentum (NAG)

Weight update is based on two steps:

- First step is on the history
- Second step is on the gradient at the look ahead point at that iteration.

Key Points:

- Looking ahead helps NAG in correcting its course (error) quicker than Moment based Gradient Descent.
- Hence the oscillations are smaller and the chances of escaping the minimum valley also smaller.

e) AdaGrad

The basic idea for the momentum-based methods was to leverage the consistency in the gradient direction of certain parameters in order to speed up the updates.
This can also be achieved by Adapting Learning Rate explicitly for each parameter.
Goal of AdaGrad is to achieve speed up in parameter update by having different learning rate for different parameters.
In this method the learning rate is decayed for parameters in proportion to their update history.

Key Points:

- AdaGrad helps in moving faster towards the solution and proceed like accelerated SGD.
- For parameter associated with sparse feature there will be fewer updates as well as learning rate will be larger.
- It decays the learning rate aggressively, as a result after a while the frequent parameter will start receiving very small updates.

- Absolute movement along all component will tend to slow down over the time.
- It sometimes prematurely become very slow.

f) RMSProp

Goal of this method is also similar as AdaGrad. Alongside in this method we try to avoid aggressively decaying of learning rate.

Key Points:

- In this method according to update rule the exponential moving average of the squared gradient is not growing rapidly.
- So, learning rate is not decaying aggressively.
- This method overcame the problem of AdaGrad.
- One disadvantage for this method is, running estimate of squared gradient is initialized to 0. This causes some undesirable bias to second order moment in the early iterations, which disappears over longer iterations.

g) Adam

Goal of Adam is to do everything that RMSProp does to solve the decay problem of AdaGrad. Alongside use a cumulative history of the gradients.

It uses first order moment and second order moment to get adaptive learning rate hence it is called Adam.

Key Points:

- Adam finally adds bias correction and momentum to RMSProp.
- Adams seems to be more or less the default choice.

4. Details of Training Data

We have been given images of size 28*28 of numbers 0, 1, 2, 3, 7 with various orientations for training, validation and testing.
We need to train the fully connected neural network (FCNN) using different optimizers for the backpropagation algorithm and compare the number of epochs that it takes for convergence along with their classification performance.

Number of sample images

Numbers	Training	Validation	Testing
0	2277	759	759
1	2277	759	759
2	2277	759	759
3	2277	759	759
7	2277	759	759

5. Parameters Used

- Stopping criteria is used as absolute difference between average error of successive epochs falls below a threshold of 0.0001.
- Learning rate(η) is used as 0.001 for all the optimizers.
- Momentum parameter is considered as 0.9 for both generalized delta and NAG.
- For RMSProp, $\beta = 0.9$ and $\varepsilon = 10^{-8}$ are used.
- For Adam, $\beta 1 = 0.9$, $\beta 2 = 0.999$ and $\varepsilon = 10^{-8}$ are used.
- Weights are initialized using the keras initializer with a predefined seed value(in our case seed = 68).The RandomNormal initializer is used which generates a tensor with random values drawn from a normal distribution with mean 0, and the standard deviation 0.05. The seed parameter is set to 68, which ensures that the random

numbers generated are reproducible across different runs of the program.

- This initializer is commonly used to initialize the weights of a neural network. Initializing the weights randomly is important because it helps to break the symmetry of the model and ensures that each neuron learns to represent a different feature of the input data. By setting a non-zero standard deviation, the initializer also ensures that the weights are not too small, which can cause the gradients to vanish during training, or too large, which can cause the gradients to explode.

  Initial weights are as follows:

```
[[-0.01877342 -0.00252981 -0.04085181]
 [-0.00878202  0.02146926 -0.01751876]
 [-0.0138307   0.0181092   0.05117702]]
```

- Activation Function : Logistic Sigmoid
- Loss Function : Cross Entropy with metric as Accuracy.

6. Architectures Used

- We have used four architectures for analysing the optimizers performance.
- There are three hidden layers for all the architectures.
- Output layer is having 10 nodes

Architectures	No of nodes in 1st Hidden Layer	No of nodes in 2nd Hidden Layer	No of nodes in 3rd Hidden Layer
256-128-64-10	256	128	64
512-256-128-10	512	256	128
32-64-128-10	32	64	128
128-256-512-10	128	256	512

7. Results and Inferences

Number of epochs for convergence in each architecture

Optimizer	256-128-64-10	512-256-128-10	32-64-128-10	128-256-512-10
SGD	13	16	9	14
Batch-SGD	1499	1545	908	460
Generalized delta	2803	2176	4595	2883
NAG	5	6	5	4
AdaGrad	242	196	205	182
RMSProp	11	6	5	5
Adam	5	10	10	13

Inference:

- ➢ The architectures have different number of layers and nodes in each layer, and hence may have different computational complexity.
- ➢ The choice of optimization algorithm affects the convergence speed and the number of epochs required to reach convergence.
- ➢ It can be inferred from above that SGD is faster than Batch-SGD, it is happening because in batch-SGD (Given that batch size is number of training examples) Gradient values are collected over an entire epoch and the weights are updated at once but at the same time in SGD we update weights for all the examples.
- ➢ Among the optimization algorithms listed, Adam generally converges the fastest, while Generalized delta converges the slowest. However, the number of epochs required for convergence can vary significantly depending on the architecture and optimization algorithm used.
- ➢ It seems that Generalized delta have taken a lot of U-turns to reach final solution because update rule in this case directs to have larger steps (due to momentum), NAG is having less oscillations due to introduction of lookahead point.
- ➢ While comparing AdaGrad RMSProp and Adam, we can infer that as AdaGrad decays the learning rate aggressively it is taking more number of epochs to converge, in RMSProp we resolve this problem.

- ***Plots of average training error (y-axis) vs. epochs (x-axis)***

  ➢ <u>First architecture (256-128-64)</u>

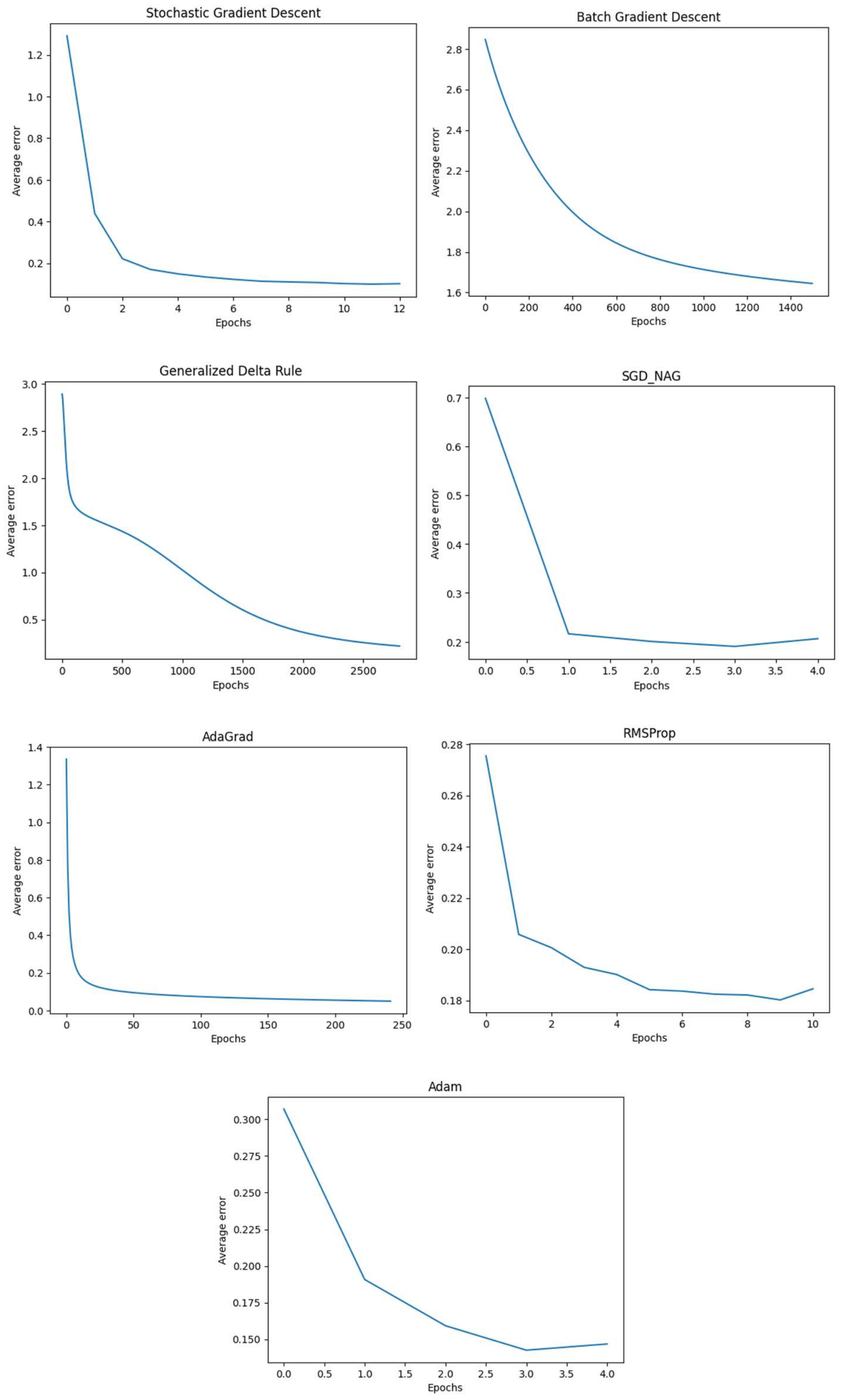
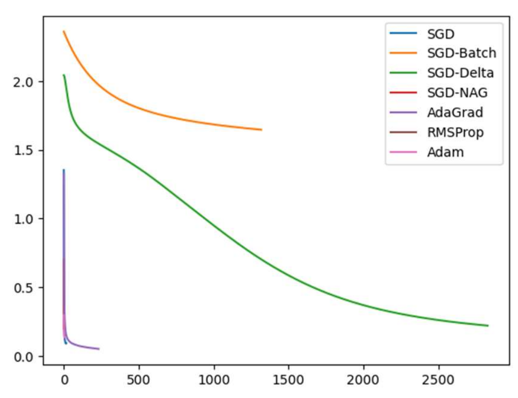
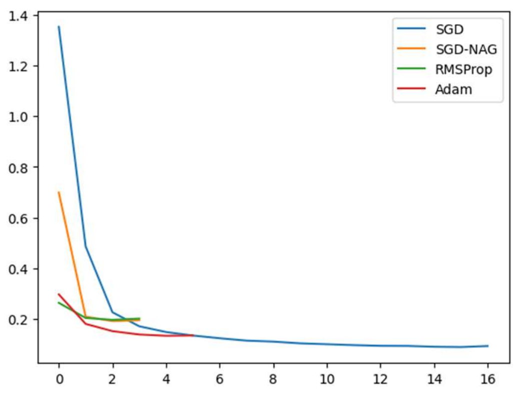


Figure 2: Average error vs epochs



(a)



(b)

Figure 3: Error plots of all the optimizers on 1st Architecture

Inference:

- Figure 3(a) shows average error vs number of epochs for all the optimizers, it can be inferred that except SGD-batch and SGD-Delta all are converging with less number of epochs compare to them.
- Figure 3(b) shows optimizers which are taking lesser number of epochs for converging, it can be inferred that SGD is converging with minimum average error whereas Adam is converging with a smaller number of epochs compare to SGD and it is also having less average error compare to SGD-NAG and RMSProp (It was expected, that Adam will be good).

➢ <u>Second architecture (512-256-128)</u>

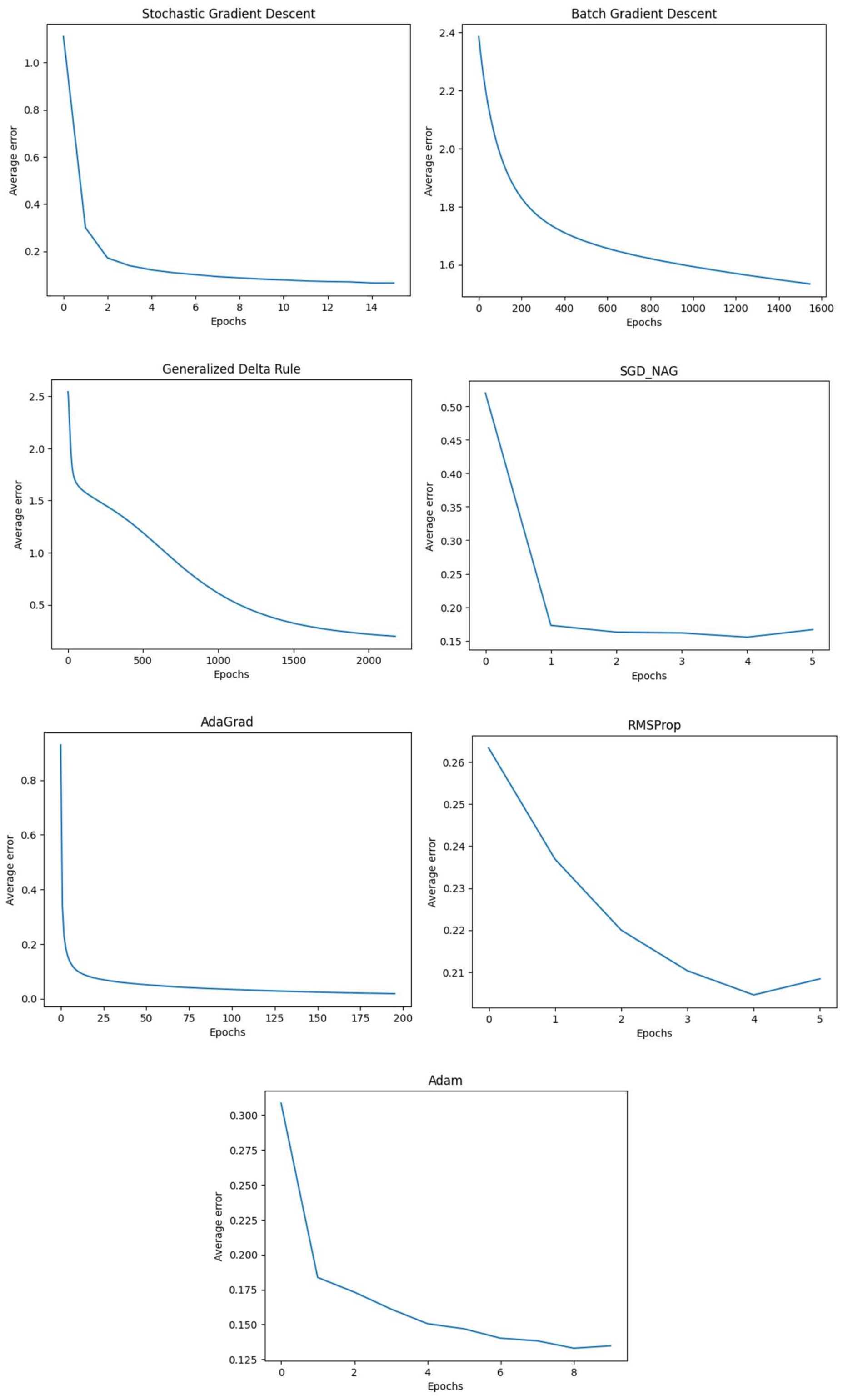
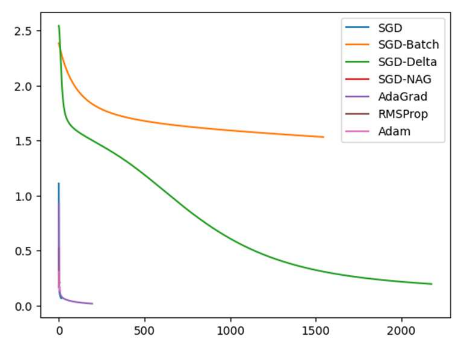
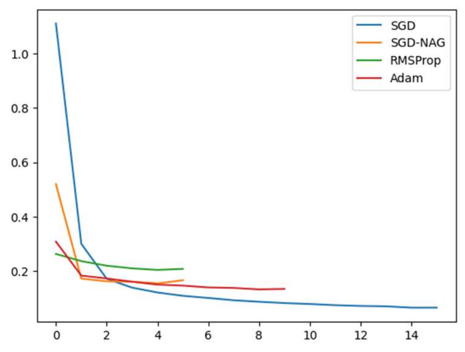


Figure 4: Average error vs epochs





(a) (b)

Figure 5: Error plots of all the optimizers on 2nd Architecture

Inference:

- Figure 5(a) shows average error vs number of epochs for all the optimizers, it can be inferred that except SGD-batch and SGD-Delta all are converging with a smaller number of epochs compare to them.
- Figure 5(b) shows optimizers which are taking lesser number of epochs for converging, it can be inferred that SGD is converging with minimum average error whereas Adam is converging with a smaller number of epochs compare to SGD and it is also having less average error compare to SGD-NAG and RMSProp (It was expected, that Adam will be good).

➢ Third architecture (32-64-128)

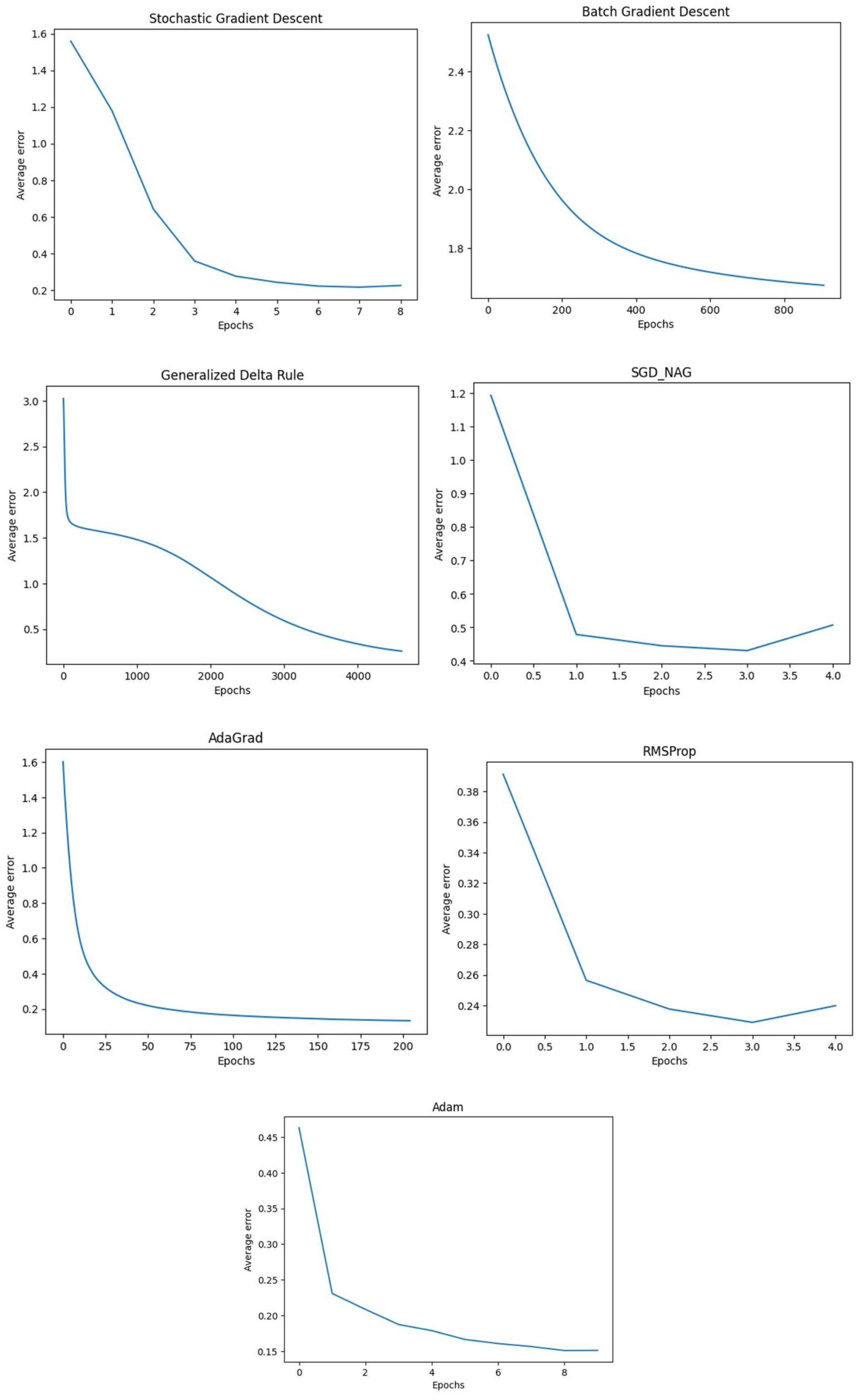
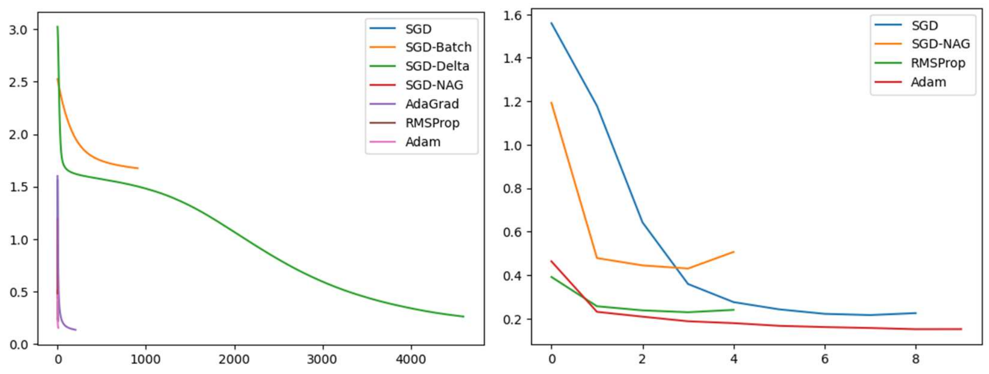


Figure 6: Average error vs epochs



(a) (b)

Figure 7: Error plots of all the optimizers on 3rd Architecture

Inference:

- Figure 7(a) shows average error vs number of epochs for all the optimizers, it can be inferred that except SGD-batch and SGD-Delta all are converging with a smaller number of epochs compare to them.

- Figure 7(b) shows optimizers which are taking lesser number of epochs for converging, it can be inferred that Adam is converging with minimum average error although it is taking some more number of epochs.

➢ <u>Fourth architecture (128-256-512)</u>

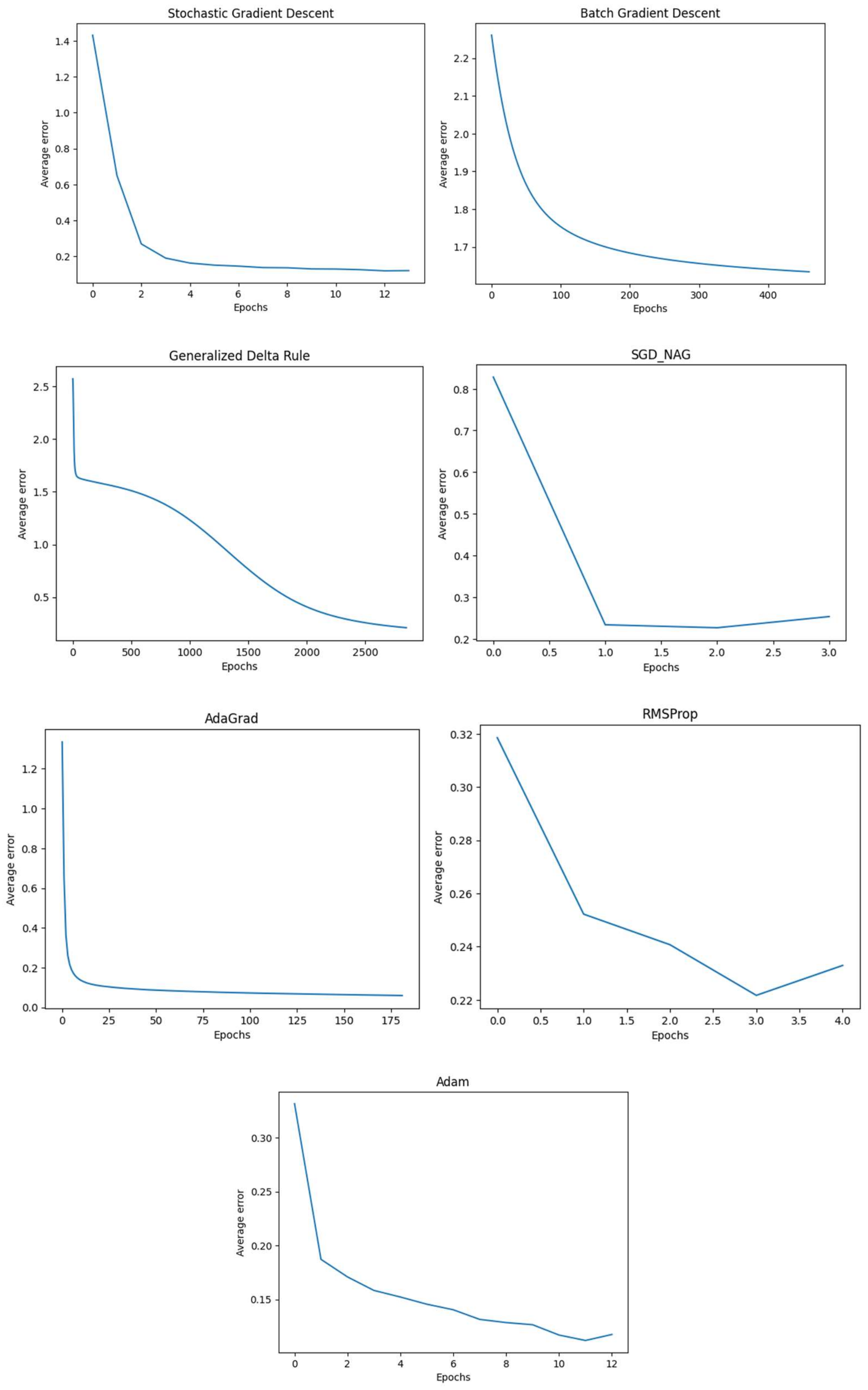
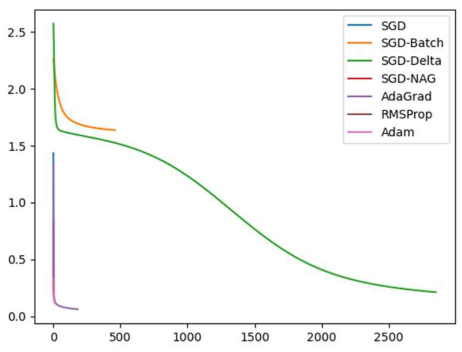
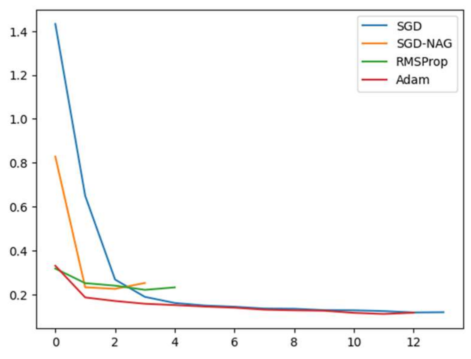


Figure 8: Average error vs epochs





(a) (b)

Figure 9: Error plots of all the optimizers on 4th Architecture

Inference:

- Figure 9(a) shows average error vs number of epochs for all the optimizers, it can be inferred that except SGD-batch and SGD-Delta all are converging with a smaller number of epochs compare to them.

- Figure 9(b) shows optimizers which are taking lesser number of epochs for converging, it can be inferred that SGD and Adam are converging with approximately similar minimum average error, SGD-NAG and RMS-Prop are converging with a smaller number of epochs but with greater average error compare to SGD and Adam (It was expected also that Adam will be good).

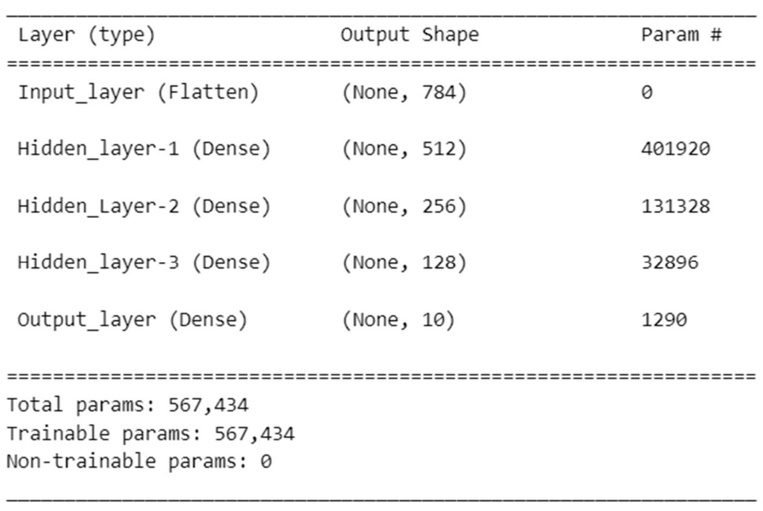
Training accuracy for each architecture

Optimizer	256-128-64-10	512-256-128-10	32-64-128-10	128-256-512-10
SGD	97.39	98.39	93.43	96.32
Batch-SGD	69.01	80.29	43.59	37.17
Generalized delta	95.84	95.69	95.18	95.4
NAG	94.13	94.78	76.88	92.6
AdaGrad	98.63	99.55	96.16	98.21
RMSProp	96.67	97.06	96.03	96.10
Adam	95.99	95.92	95.6	97.08

Validation accuracy for each architecture

Optimizer	256-128-64-10	512-256-128-10	32-64-128-10	128-256-512-10
SGD	96.86	97.79	93.7	96
Batch-SGD	69.46	79.66	41.92	38.50
Generalized delta	94.91	94.99	94	94.6
NAG	94.02	94.04	76.73	92
AdaGrad	96.79	97.39	94.99	96.15
RMSProp	96.18	96.60	95.55	95.81
Adam	96.23	95.86	95.2	96.47

Best architecture based on validation accuracy



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Input_layer (Flatten)       (None, 784)               0

 Hidden_layer-1 (Dense)      (None, 512)               401920

 Hidden_Layer-2 (Dense)      (None, 256)               131328

 Hidden_layer-3 (Dense)      (None, 128)               32896

 Output_layer (Dense)        (None, 10)                1290

=================================================================
Total params: 567,434
Trainable params: 567,434
Non-trainable params: 0
_________________________________________________________________
```

• *Test Confusion Matrix & Test Classification accuracy for Best Architecture*

Based on validation accuracy 512-256-128-10 architecture was performing better compared to other architectures considered. This could be because the other architectures we tested maybe overfitting the training data. Overfitting occurs when a model is too complex for the amount of available data, and it starts to fit the noise in the training data instead of the underlying patterns. The 512-256-128 architecture may have just the right amount of capacity to avoid overfitting while still capturing the relevant features of the data.

Adam:

		Actual Class				
Predicted Class		**Class 1**	**Class 2**	**Class 3**	**Class 4**	**Class 5**
	Class 1	742	0	3	13	1
	Class 2	0	737	7	13	2
	Class 3	14	0	701	39	5
	Class 4	6	4	14	725	10
	Class 5	3	8	16	11	721

Accuracy – 95.55%

RMSProp:

		Actual Class				
Predicted Class		**Class 1**	**Class 2**	**Class 3**	**Class 4**	**Class 5**
	Class 1	732	0	16	7	4
	Class 2	0	736	11	7	5
	Class 3	5	1	733	11	9
	Class 4	3	1	20	722	13
	Class 5	1	1	16	7	734

Accuracy– 96.36%

AdaGrad:

		Actual Class				
		Class 1	Class 2	Class 3	Class 4	Class 5
Predicted Class	**Class 1**	751	0	3	2	3
	Class 2	1	749	4	3	2
	Class 3	8	5	733	7	6
	Class 4	6	3	16	724	10
	Class 5	2	4	13	6	734

Accuracy– 97.26%

NAG:

		Actual Class				
		Class 1	**Class 2**	**Class 3**	**Class 4**	**Class 5**
Predicted Class	**Class 1**	747	0	9	2	2
	Class 2	0	742	16	0	1
	Class 3	22	5	709	12	11
	Class 4	30	3	62	637	27
	Class 5	6	5	26	1	721

Accuracy – 93.68%

Delta:

		Actual Class				
		Class 1	**Class 2**	**Class 3**	**Class 4**	**Class 5**
Predicted Class	**Class 1**	746	0	9	2	2
	Class 2	0	724	16	0	1
	Class 3	22	5	709	12	11
	Class 4	30	3	62	637	27
	Class 5	6	5	26	1	721

Accuracy– 94.81%

Batch:

	Actual Class					
Predicted Class		Class 1	Class 2	Class 3	Class 4	Class 5
	Class 1	735	2	4	5	13
	Class 2	1	708	18	8	24
	Class 3	119	56	413	73	98
	Class 4	102	52	37	517	51
	Class 5	65	65	21	12	596

Accuracy – 78.23%

SGD:

	Actual Class					
Predicted Class		Class 1	Class 2	Class 3	Class 4	Class 5
	Class 1	750	0	5	2	2
	Class 2	0	749	3	5	2
	Class 3	9	4	731	10	5
	Class 4	4	5	13	734	3
	Class 5	4	3	14	6	732

Accuracy – 97.39%

Inference:

- As far as Validation accuracy is concerned, we can infer that Second Architecture (512-256-128-10) is best among all the architectures.
- Above confusion matrices are for Second architecture, it can be inferred using these confusion matrices that accuracy wise SGD is best (97.39%).
- The model is converging faster when using the Tanh activation function is because of its symmetric nature and the range of its output values.The Tanh activation function outputs values between -1 and 1, which is a range that is centered around zero. This means that the outputs of the Tanh activation function are symmetric around zero.