



CS-671 DEEP LEARNING AND APPLICATIONS

ASSIGNMENT 3

Implementing Optimizers for Backpropagation Algorithm

Submitted by:

Group 13

Prachi Sharma (V21078)

Ayush Tiwari (T22053)

Instructor:

Prof. A D Dileep

March 22, 2023

Table of Contents

1.1. Fully Connected Neural Network with more than 3 layers	2
1.2. Problems with Gradient Descent Algorithm	2
1.3. Brief introduction to better optimisation techniques for backpropagation algorithm	3
1.3.1. Stochastic gradient descent	3
1.3.2. Batch (Vanilla) gradient descent	3
1.3.3. Momentum-based gradient descent (Generalised Delta Rule)	3
1.3.4. Nesterov accelerated gradient descent	4
1.3.5. AdaGrad	5
1.3.6. RMSProp	5
1.3.7. Adaptive Moments (Adam)	5
1.4. Implementing various optimisation techniques	6
1.5. Comparing various optimisers	7
1.5.1. Architecture 1	7
1.5.2. Architecture 2	10
1.5.3. Architecture 3	13
1.5.4. Architecture 4	16
1.5.5. Architecture 5	19
1.6. Inferences	22
1.7. Results of best selected architecture	23

Introduction

1.1. Fully Connected Neural Network with more than 3 layers

- A fully connected neural network (FCNN) is a neural network of an input layer, two or more hidden layers and an output layer, such that each neuron in a layer is connected to each neuron in the next layer through weights.
- The input layer consists of d linear neurons which return the input value as output value. Here, the input pattern is of dimension d .
- The hidden layers consist of a varied number of nonlinear neurons which are connected to each neuron in the next layer.
- The output layer consists of K linear/sigmoidal/softmax neurons depending on the problem.
- The number of neurons in each layer and the number of layers are decided experimentally.
- A FCNN with at least three layers is considered as a deep network.
- While training a deep neural network, there are many challenges that can be encountered while convergence of training error. To avoid these challenges, different techniques are used to improve the model's learning process.

1.2. Problems with Gradient Descent Algorithm

Gradient descent is an optimization algorithm used to minimize error function after every iteration by adjusting model parameters. A FCNN uses backpropagation algorithm to apply gradient descent in order to calculate gradients of error function w.r.t. weight parameters at each layer. However, there are several challenges that can be encountered while using backpropagation. Few of them are:

- **Slow convergence:** Gradient descent may converge very slowly when the cost function is complex or has many local minima. It may also happen that gradient descent may oscillate before final convergence leading to slow convergence.
- **Overshooting minima:** Gradient descent may overshoot the minimum of the cost function and diverge.
- **Getting stuck in local minima:** Gradient descent can get stuck in a local minimum instead of reaching the global minimum of the cost function.
- Oscillations before convergence:
- **High computation cost:** Gradient descent can be computationally expensive when we have huge data or the model is too complex.

In this assignment, we will be implementing and comparing different optimisers which can be used to address above mentioned problems.

1.3. Brief introduction to better optimisation techniques for backpropagation algorithm

1.3.1. Stochastic gradient descent

- In stochastic gradient descent (SGD), parameters are updated after forward propagating every data point.
- In SGD, we do not have the true gradient of the error function but an approximation of the true gradient.
- It does not guarantee that each iteration will decrease the loss.
- This results in oscillations before convergence.

1.3.2. Batch (Vanilla) gradient descent

- In vanilla gradient descent (VGD), parameters are updated when all the patterns have been forward propagated once i.e. after one epoch.
- VGD considers the true gradient of the error function to make weight updates.
- Each step guarantees that there will be a decrease in error as we are dealing with true gradient.
- Huge number of calculations are done before single weight update leading to slow convergence.

1.3.3. Momentum-based gradient descent (Generalised Delta Rule)

- Momentum-based GD is an extension of the standard stochastic gradient descent algorithm that helps the optimization process converge faster and more reliably.
- In momentum-based stochastic gradient descent, an additional term called momentum is added to the update which accumulates the gradient values over previous iterations and smooths out the changes in the direction of the gradient .

$$\Delta w(m) = -\eta \Delta w(m) + \alpha \Delta w(m-1), \quad 0 < \alpha < 1 \text{ is the momentum.}$$

Here, $\Delta w(m)$ is gradient at m^{th} iteration and weight update rule is:

$$w(m+1) = w(m) + \Delta w(m) = w(m) - \eta \Delta w(m) + \alpha \Delta w(m-1), \quad 0 < \eta < 1 \text{ is the learning}$$

rate.

- If the gradient has the same sign on consecutive iterations then the net weight change increases over these iterations causing downward acceleration.

- If the gradient has different sign on consecutive iterations then the net weight change decreases over these iterations causing a downward deceleration.
- Pros:
 - Faster convergence as it takes large steps towards minima.
 - Helps escape local minima and saddle points by overshooting it.
- Cons:
 - More complex to implement and hard to tune than standard SGD.
 - Can oscillate in and out of global minima if momentum term is too high.

1.3.4. Nesterov accelerated gradient descent

- To reduce the oscillations caused in case of momentum-based SGD, we use nesterov accelerated gradient descent (NAG).
- Momentum-based GD considers two factors while updating weights: **current gradient and weight change history**.
- NAG uses a “**look ahead**” method to know where it expects to be in the next step and adjust its direction of weight change accordingly.

$$w(m)_{look\ ahead} = w(m) + \alpha \Delta w(m - 1), \quad 0 < \alpha < 1 \text{ is the momentum.}$$

- NAG first takes a step in the direction of momentum and calls this point as the “**look ahead**” point.
- It then calculates the look ahead gradient to see where it ends up after taking this step.
- This look ahead gradient and the momentum term is added to take a step forward in the direction of the updated gradient.

$$\Delta w(m) = -\eta \Delta w(m)_{look\ ahead} + \alpha \Delta w(m - 1), \quad 0 < \eta < 1 \text{ is the learning rate.}$$

- The weight update rule is

$$w(m + 1) = w(m) + \Delta w(m) = w(m) - \eta \Delta w(m)_{look\ ahead} + \alpha \Delta w(m - 1)$$

- Pros:
 - Looking ahead helps NAG to correct its course faster than momentum-based GD.
 - There are fewer oscillations.
 - NAG can navigate through valleys in the loss surface by using momentum term.

1.3.5. AdaGrad

- Uses different learning rates for different parameters to speed up the convergence
 - It tries to adapt learning rates explicitly for each of the parameters
 - Only suitable for batch and mini-batch gradient descent techniques. Not suitable for stochastic gradient descent
 - It decays the learning rate in proportion to the number of updates a parameter is getting. Parameters updated fewer times have larger learning rate compared to the parameters having more updates.
 - $v_l(m) = v_l(m - 1) + (\nabla w_l(m))^2$
 $v_l(m)$ is the update history variable for an l th parameter
- $$w_l(m + 1) = w_l(m) - \eta * \nabla w_l(m) / \sqrt{v_l(m) + \epsilon} \text{ where } \epsilon \text{ is small positive value such as } 1e-8$$
- Cons
 - Since the learning rate decays over time for the parameters and hence movement along all the component will tend to slow over time
 - It decays the learning rate very aggressively and due to which it becomes very slow before reaching the solution.

1.3.6. RMSProp

- It has the same concept of having different learning rates for different parameters as in AdaGrad. Over AdaGrad, it tries to avoid decaying of learning rate aggressively by introducing decay factor.
 - $v_l(m) = \beta v_l(m - 1) + (1 - \beta)(\nabla w_l(m))^2$ where $0 < \beta < 1$ is the decay factor.
- $$w_l(m + 1) = w_l(m) - \eta \nabla w_l(m) / \sqrt{v_l(m) + \epsilon}$$
- Cons
 - The running estimate of squared gradient is initialized to 0 which causes some kind of undesirable bias to the second order moment during the initial iterations. This bias disappears over longer iterations.

1.3.7. Adaptive Moments (Adam)

- It works similar to RMS prop to solve the decay problem of AdaGrad.
- It updates the parameters by combining the momentum and the adaptive learning rate.
- Weight update rule:

$$w(m + 1) = w(m) - \eta \widehat{u_l(m)} / \sqrt{\widehat{v_l(m)} + \epsilon}$$

where $\widehat{u_l(m)} = u_l(m) / (1 - \beta_1^m)$ and $\widehat{v_l(m)} = v_l(m) / (1 - \beta_2^m)$

$$\text{and } v_l(m) = \beta_2 v_l(m-1) + (1 - \beta_2)(\nabla w_l(m))^2 \quad \text{eqn. 1}$$

$$u_l(m) = \beta_1 u_l(m-1) + (1 - \beta_1)(\nabla w_l(m)) \quad \text{eqn. 2}$$

- Eqn (1) takes care of decaying learning rate and Eqn (2) takes care of momentum.
- Current SOTA for the choice of optimisers in general.

1.4. Implementing various optimisation techniques

We are given a subset of the MNIST handwritten digit dataset having 5 classes (digits) : 1, 2, 3, 6, 9. The task is to implement and analyze the speed of convergence in different gradient descent optimisers. Each model is trained using images as inputs, each of size (28*28).

Architecture	Input layer	Hidden layer 1	Hidden layer 2	Hidden layer 3	Hidden layer 4	Hidden layer 5	Output
1	784	128	64	128	-	-	10
2	784	256	128	256	-	-	10
3	784	512	256	128	-	-	10
4	784	512	256	128	64	-	10
5	784	512	256	128	64	32	10

The output layer has 10 neurons representing 10 classes {0,1,2,3,4,5,6,7,8,9}. The input set contains only 5 classes {1,2,3,6,9}. Hence, the model should not classify any input out of these classes.

The data is normalized using MinMax normalization to scale all input features in the range of [0,1]. Convergence criterion for all models is the difference between successive epochs is less than or equal to 10^{-4} .

1.5. Comparing various optimisers

1.5.1. Architecture 1

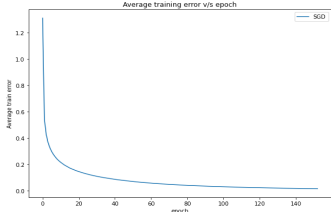
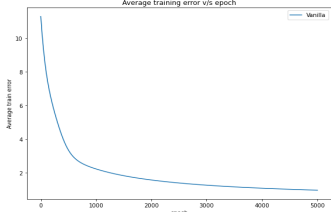
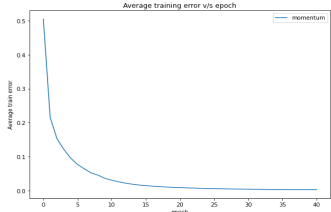
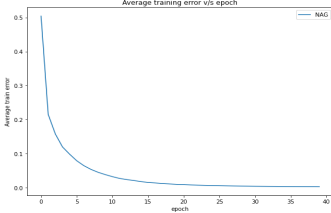
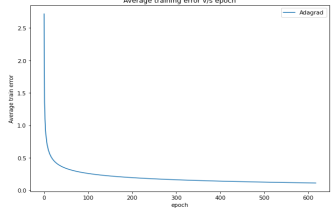
Model: "sequential_1"		
Layer (type)	Output Shape	Param #
input_layer (Flatten)	(None, 784)	0
hidden_layer_1 (Dense)	(None, 128)	100480
hidden_layer_2 (Dense)	(None, 64)	8256
hidden_layer_3 (Dense)	(None, 128)	8320
output_layer (Dense)	(None, 10)	1290
Total params: 118,346		
Trainable params: 118,346		
Non-trainable params: 0		

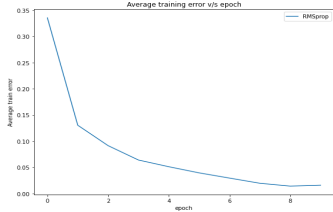
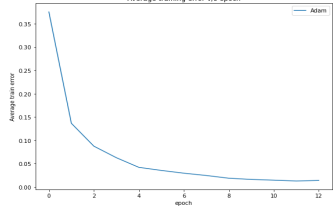
Fig: The figure shows the architecture summary including number of layers, number of units in each layer and total number of parameters.

Parameter values for various optimisers:

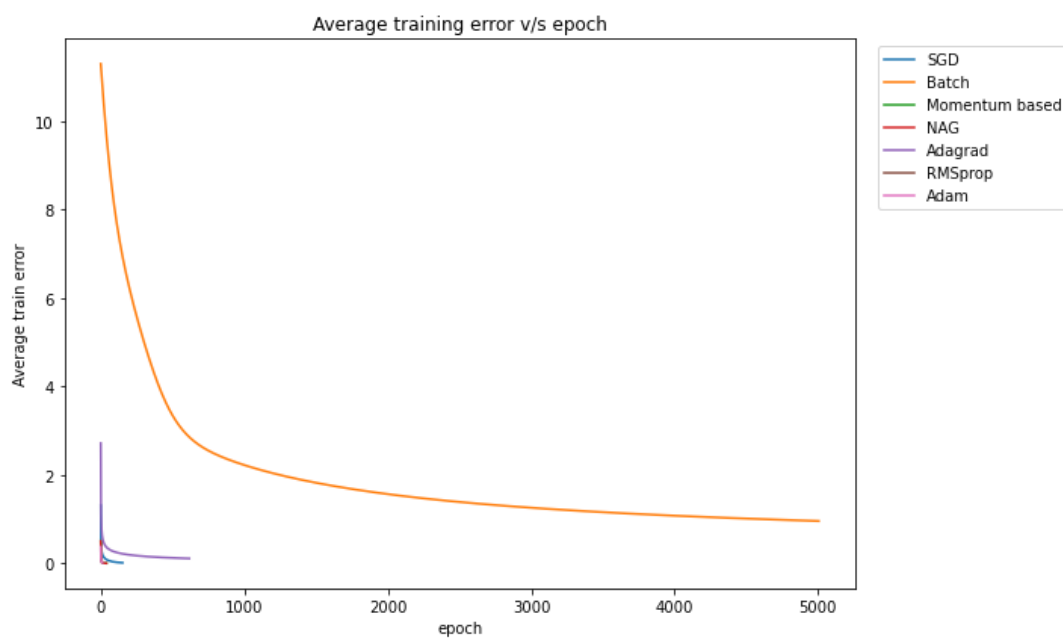
Hyperparameters	Optimiser	Values
Hidden layer activation function	All	Logistic
Output layer activation function	All	Softmax
Loss function	All	Cross Entropy
Learning rate (η)	All	0.001
Alpha (α)	Momentum based GD, NAG	0.9
Beta (β)	RMSprop	0.99
Beta 1 (β_1)	Adam	0.9
Beta 2 (β_2)	Adam	0.999
Epsilon (ϵ)	Adagrad, RMSprop, Adam	10^{-8}

Results and Observations:

Comparison of various optimiser performance				
Optimiser	No. of epochs	Error vs epoch	Training accuracy	Validation Accuracy
SGD (batch_size=1)	153		0.9987	0.9568
Batch GD (batch_size=N)	5005		0.6661	0.6672
Momentum based GD (batch_size=1)	41		1.0000	0.9615
NAG (batch_size=1)	40		1.0000	0.9599
Adagrad (batch_size=1)	618		0.9683	0.9428

RMSprop (batch_size=1)	10		0.9960	0.9773
Adam (batch_size=1)	13		0.9961	0.9797

Plot of training error for all optimisers



1.5.2. Architecture 2

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
input_layer (Flatten)	(None, 784)	0
hidden_layer_1 (Dense)	(None, 256)	200960
hidden_layer_2 (Dense)	(None, 128)	32896
hidden_layer_3 (Dense)	(None, 256)	33024
output_layer (Dense)	(None, 10)	2570

```

Total params: 269,450
Trainable params: 269,450
Non-trainable params: 0

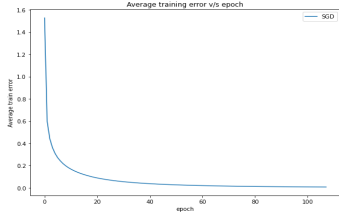
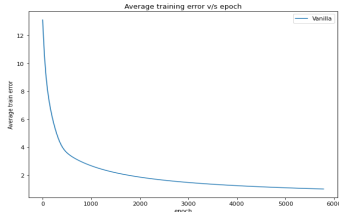
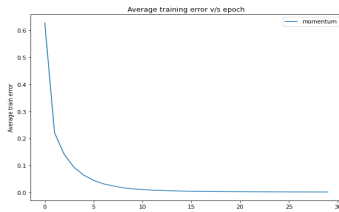
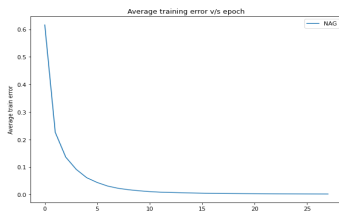
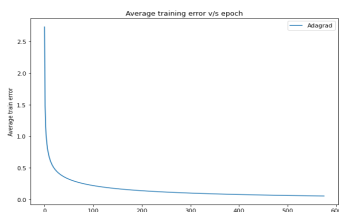
```

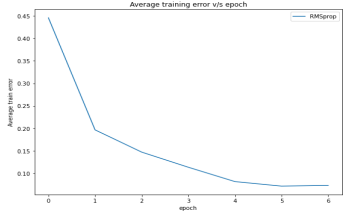
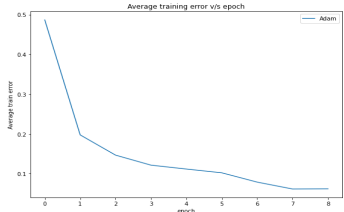
Fig: The figure shows the architecture summary including number of layers, number of units in each layer and total number of parameters.

Parameter values for various optimisers:

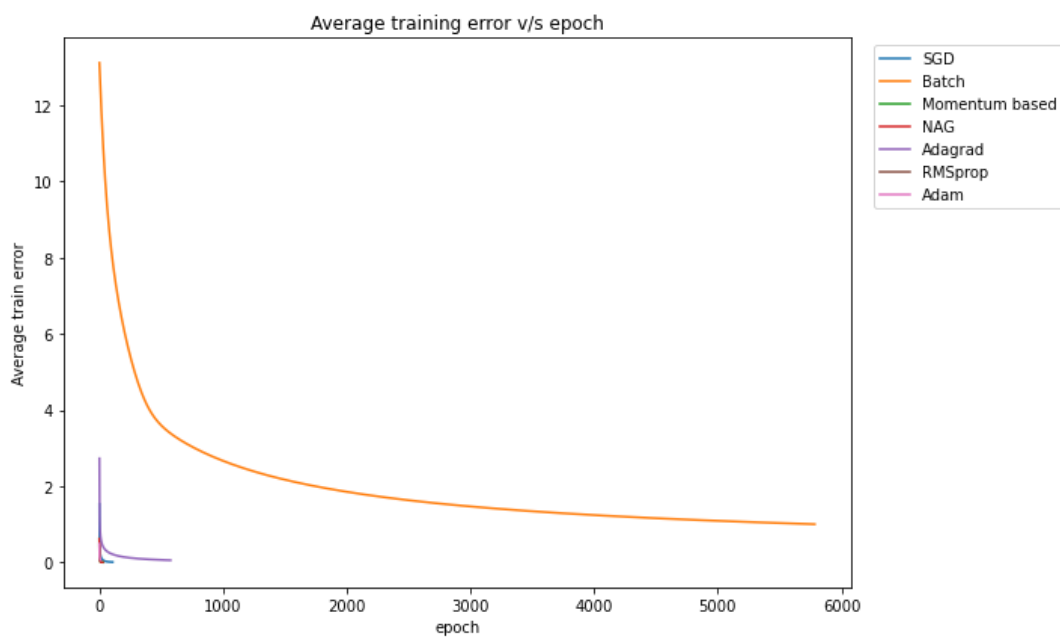
Hyperparameters	Optimiser	Values
Hidden layer activation function	All	Logistic
Output layer activation function	All	Softmax
Loss function	All	Cross Entropy
Learning rate (η)	All	0.001
Alpha (α)	Momentum based GD, NAG	0.9
Beta (β)	RMSprop	0.99
Beta 1 (β_1)	Adam	0.9
Beta 2 (β_2)	Adam	0.999
Epsilon (ϵ)	Adagrad, RMSprop, Adam	10^{-8}

Results and Observations:

Comparison of various optimiser performance				
Optimiser	No. of epoch	Error vs epoch	Training accuracy	Validation Accuracy
SGD (batch_size=1)	108		0.9997	0.9397
Batch GD (batch_size=N)	5785		0.7053	0.6901
Momentum based GD (batch_size=1)	30		1.0000	0.9599
NAG (batch_size=1)	28		1.0000	0.9510
Adagrad (batch_size=1)	576		0.9877	0.9194

RMSprop (batch_size=1)	7		0.9850	0.9755
Adam (batch_size=1)	9		0.9859	0.9752

Plot of training error for all optimisers



1.5.3. Architecture 3

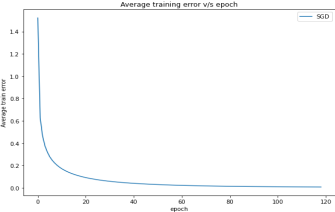
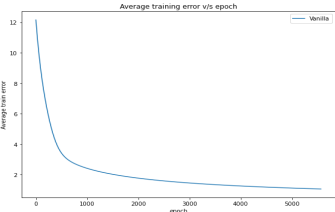
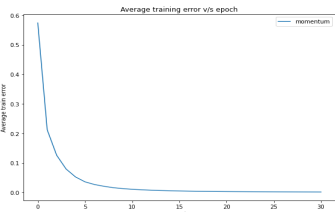
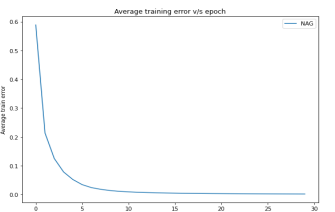
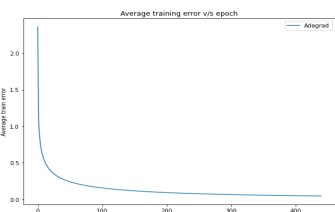
Model: "sequential"		
Layer (type)	Output Shape	Param #
input_layer (Flatten)	(None, 784)	0
hidden_layer_1 (Dense)	(None, 512)	401920
hidden_layer_2 (Dense)	(None, 256)	131328
hidden_layer_3 (Dense)	(None, 128)	32896
output_layer (Dense)	(None, 10)	1290
Total params: 567,434		
Trainable params: 567,434		
Non-trainable params: 0		

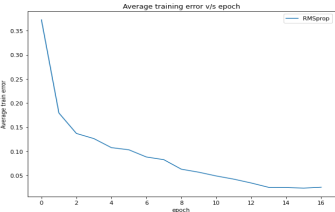
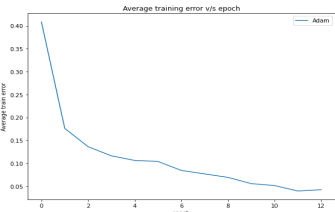
Fig: The figure shows the architecture summary including number of layers, number of units in each layer and total number of parameters.

Parameter values for various optimisers:

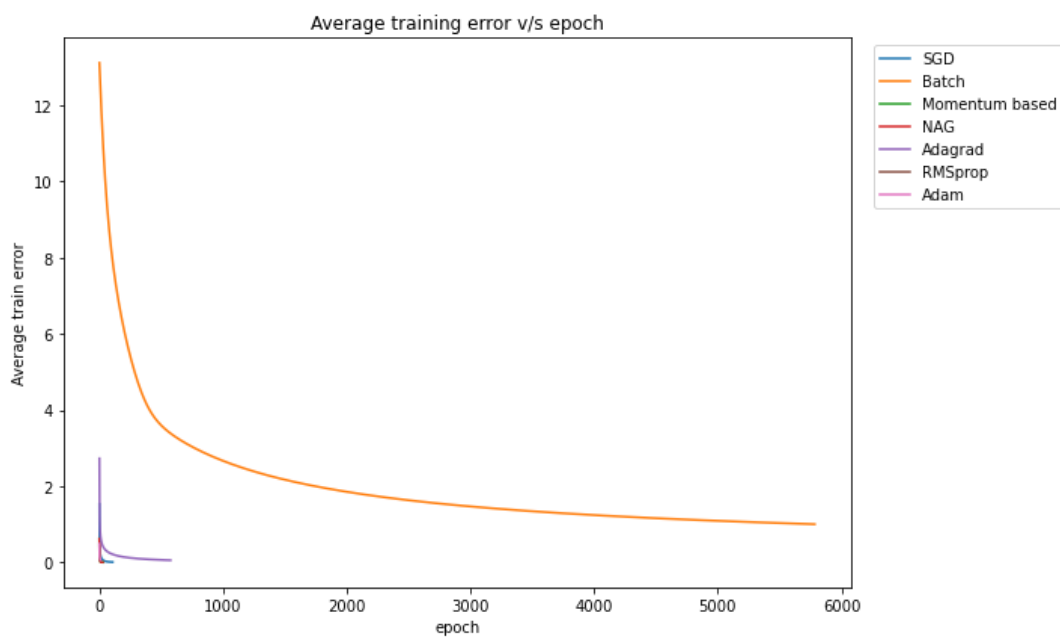
Hyperparameters	Optimiser	Values
Hidden layer activation function	All	Logistic
Output layer activation function	All	Softmax
Loss function	All	Cross Entropy
Learning rate (η)	All	0.001
Alpha (α)	Momentum based GD, NAG	0.9
Beta (β)	RMSprop	0.99
Beta 1 (β_1)	Adam	0.9
Beta 2 (β_2)	Adam	0.999
Epsilon (ϵ)	Adagrad, RMSprop, Adam	10^{-8}

Results and Observations:

Comparison of various optimiser performance				
Optimiser	No. of epoch	Error vs epoch	Training accuracy	Validation Accuracy
SGD (batch_size=1)	119		99.96%	93.47%
Batch GD (batch_size=N)	5567		65.36%	63.45%
Momentum based GD (batch_size=1)	31		100%	94.76%
NAG (batch_size=1)	30		100%	95.10%
Adagrad (batch_size=1)	440		99.34%	91.36%

RMSprop (batch_size=1)	17		99.52%	97.94%
Adam (batch_size=1)	13		98.81%	97.94%

Plot of training error for all optimisers



1.5.4. Architecture 4

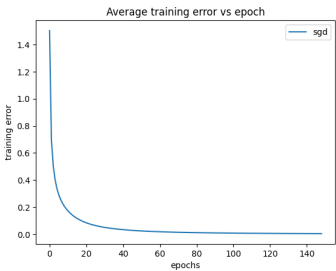
Model: "sequential_7"

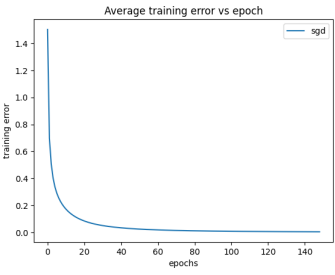
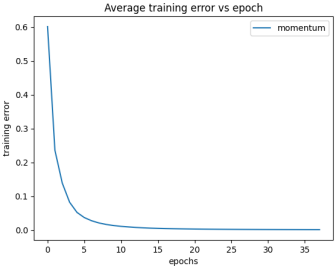
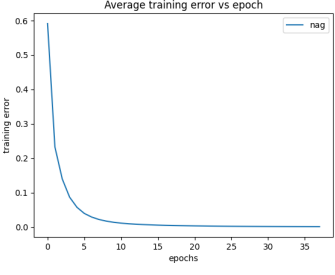
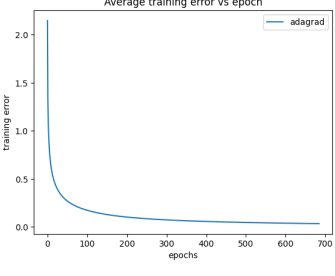
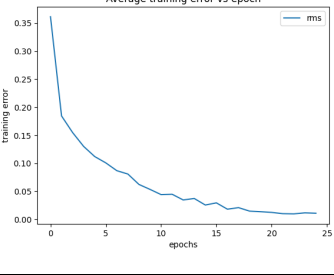
Layer (type)	Output Shape	Param #
rescaling_7 (Rescaling)	(None, 28, 28, 1)	0
flatten_7 (Flatten)	(None, 784)	0
dense_35 (Dense)	(None, 512)	401920
activation_35 (Activation)	(None, 512)	0
dense_36 (Dense)	(None, 256)	131328
activation_36 (Activation)	(None, 256)	0
dense_37 (Dense)	(None, 128)	32896
activation_37 (Activation)	(None, 128)	0
dense_38 (Dense)	(None, 64)	8256
activation_38 (Activation)	(None, 64)	0
dense_39 (Dense)	(None, 5)	325
activation_39 (Activation)	(None, 5)	0

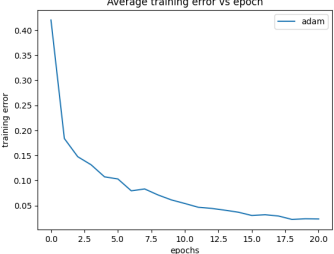
=====
Total params: 574,725
Trainable params: 574,725
Non-trainable params: 0
=====

Fig: The figure shows the architecture summary including number of layers, number of units in each layer and total number of parameters.

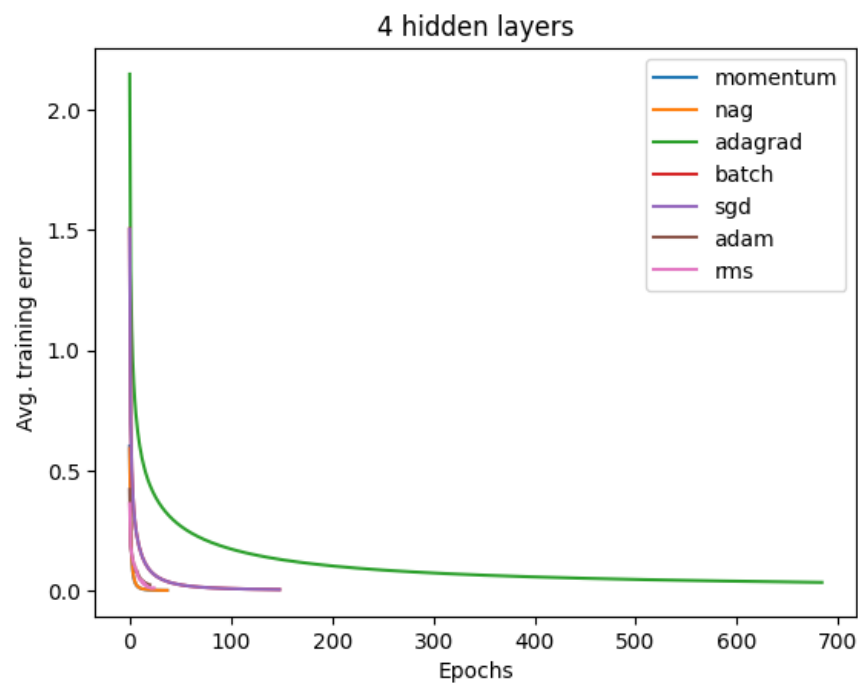
Results and Observations:

Comparison of various optimiser performance				
Optimiser	No. of epoch	Error vs epoch	Training accuracy	Validation Accuracy
SGD (batch_size=1)	149	<div><div>Average training error vs epoch</div></div>	99.63%	92.62%

Batch GD (batch_size=N)	153		99.99%	92.77%
Momentum based GD (batch_size=1)	38		100%	94.62
NAG (batch_size=1)	38		100%	94.99%
Adagrad (batch_size=1)	685		99.64%	91.09%
RMSprop (batch_size=1)	25		99.77%	98.36%

Adam (batch_size=1)	21		99.39%	97.20%
------------------------	----	---	--------	--------

Plot of training error for all optimisers

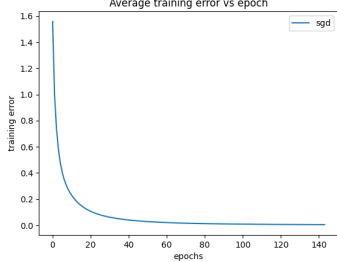
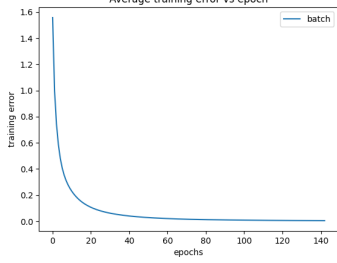


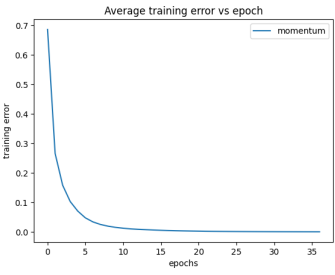
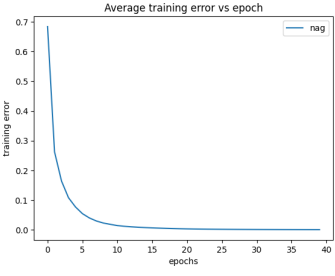
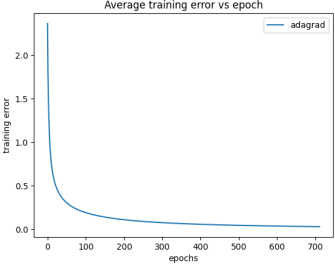
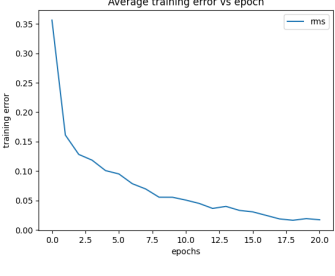
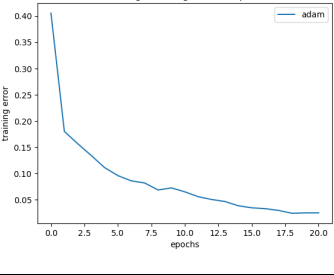
1.5.5. Architecture 5

Model: "sequential_18"

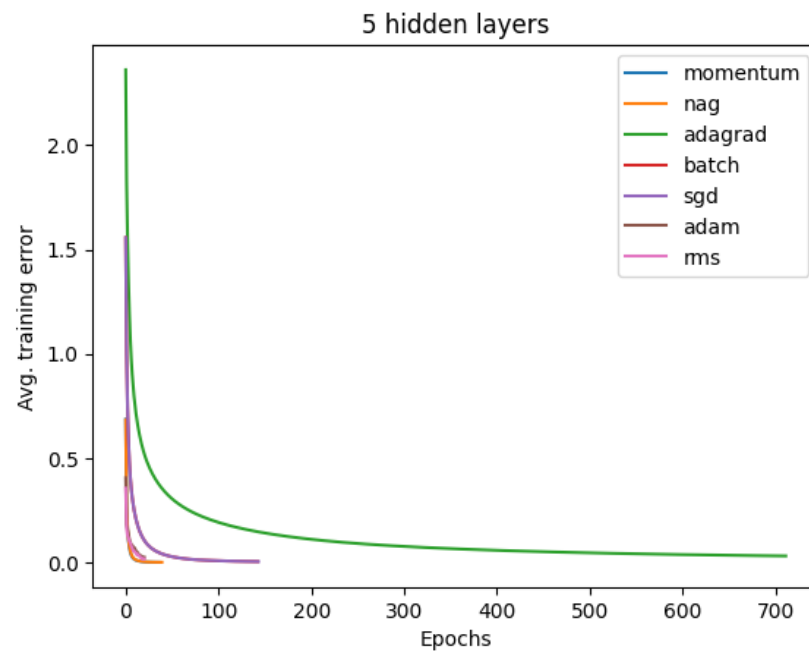
Layer (type)	Output Shape	Param #
rescaling_18 (Rescaling)	(None, 28, 28, 1)	0
flatten_18 (Flatten)	(None, 784)	0
dense_96 (Dense)	(None, 512)	401920
activation_96 (Activation)	(None, 512)	0
dense_97 (Dense)	(None, 256)	131328
activation_97 (Activation)	(None, 256)	0
dense_98 (Dense)	(None, 128)	32896
activation_98 (Activation)	(None, 128)	0
dense_99 (Dense)	(None, 64)	8256
activation_99 (Activation)	(None, 64)	0
dense_100 (Dense)	(None, 32)	2080
activation_100 (Activation)	(None, 32)	0
dense_101 (Dense)	(None, 5)	165
activation_101 (Activation)	(None, 5)	0

=====
Total params: 576,645
Trainable params: 576,645
Non-trainable params: 0

Optimiser	No. of epoch	Error vs epoch	Training accuracy	Validation Accuracy
SGD (batch_size=1)	144		99.97%	92.56%
Batch GD (batch_size=N)	143		99.97%	92.51%

Momentum based GD (batch_size=1)	37		100%	94.28%
NAG (batch_size=1)	40		100%	94.44%
Adagrad (batch_size=1)	712		99.66%	92.64%
RMSprop (batch_size=1)	21		99.63%	98.05%
Adam (batch_size=1)	21		99.29%	97.89%

Plot of training error for all optimisers



1.6. Inferences

- The training accuracy and the validation accuracy of each of the above models show that they perform well to classify MNIST data. However, the model that uses batch gradient descent with three hidden layers shows poor accuracy even after being trained for a long time. It's likely that batch gradient descent is getting stuck in a suboptimal region of the weight space and not improving the accuracy of the model. It is also possible that the learning rate (η) is too high and therefore, the parameters oscillate back and forth across the optimum and fail to converge.
- Each optimizer has a different convergence time (number of epochs), which is due to their unique approach to updating the model's weights. Batch gradient descent updates the weights after processing all input patterns, making it slow to converge. On the other hand, momentum-based gradient descent and Nesterov accelerated gradient incorporate a momentum term to speed up the process. In case of Adagrad, RMSprop and Adam, we are using adaptive learning rates (η), which makes the model take large steps for sparse parameters and small steps for dense parameters. It essentially helps with avoiding oscillations and overshooting minimum.
- Adagrad, in particular, is taking too long to converge, even though it adds gradient history. It may be because it is making the learning rate too small as training progresses. Thus, making weight updates very small. This may be avoided if we use batch or mini-batch gradient descent.
- It can be observed that RMSprop and Adam converge faster in all cases because they do not decrease the learning rate aggressively.
- The average error v/s epoch curve for RMSprop and Adam have dips and are not smooth exponentially decreasing curves due to the adaptive learning rate used by these optimizers. During training, the model may encounter steep and flat regions in the error surface. In the steep regions, the optimizer reduces the learning rate to avoid overshooting the minimum, while in the flat regions, the optimizer increases the learning rate to make more progress towards the minimum. This leads to fluctuations in the curve.
- In all cases, both RMSprop and Adam perform well and are very competitive.
- It should be noted that, in a few cases, the training accuracy is reaching 100% while validation accuracy is 95%. This suggests that our model could be memorizing the data which calls for the need of regularization techniques.

1.7. Results of best selected architecture

The parameters we have taken into consideration while comparing the performance of various models are:

1. Model complexity
2. Training time
3. Validation loss and accuracy

The results obtained on the best selected architecture with 4 hidden layer are as follows:

In the confusion matrix, the rows represent the predicted labels and the columns represent the actual label.

train : 356/356 [=====] - 1s 2ms/step - loss: 0.0227 - accuracy: 0.9933

356/356 [=====] - 1s 2ms/step

```
tf.Tensor(  
[[ 0  0  0  0  0  0  0  0  0  0]  
[ 0 2273  2  1  0  0  0  0  0  1]  
[ 0  4 2267  1  0  0  5  0  0  0]  
[ 0  6  25 2242  0  0  1  0  0  3]  
[ 0  0  0  0  0  0  0  0  0  0]  
[ 0  0  0  0  0  0  0  0  0  0]  
[ 0  2  1  2  0  0 2272  0  0  0]  
[ 0  0  0  0  0  0  0  0  0  0]  
[ 0  0  0  0  0  0  0  0  0  0]  
[ 0  3  7 11  0  0  1  0  0 2255]], shape=(10, 10), dtype=int32)
```

Accuracy 99.33%, loss 0.0227 and obtained confusion matrix

validation:

119/119 [=====] - 0s 2ms/step - loss: 0.1187 - accuracy: 0.9752

119/119 [=====] - 0s 2ms/step

```
tf.Tensor(  
[[ 0  0  0  0  0  0  0  0  0  0]  
[ 0 752  3  0  0  0  3  0  0  1]  
[ 0  1 744  4  0  0  7  0  0  3]  
[ 0  7  20 719  0  0  2  0  0 11]  
[ 0  0  0  0  0  0  0  0  0  0]  
[ 0  0  0  0  0  0  0  0  0  0]  
[ 0  3  6  4  0  0 746  0  0  0]  
[ 0  0  0  0  0  0  0  0  0  0]  
[ 0  0  0  0  0  0  0  0  0  0]  
[ 0  3  4 11  0  0  1  0  0 740]], shape=(10, 10), dtype=int32)
```

Accuracy 97.52%, loss 0.1187 and obtained confusion matrix

test:

119/119 [=====] - 0s 2ms/step - loss: 0.1388 - accuracy: 0.9744

119/119 [=====] - 0s 2ms/step

tf.Tensor(
[[0 0 0 0 0 0 0 0 0 0]
[0 753 5 1 0 0 0 0 0 0]
[0 10 726 10 0 0 8 0 0 5]
[0 9 12 728 0 0 4 0 0 6]
[0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0]
[0 3 4 0 0 0 751 0 0 1]
[0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0]
[0 8 2 8 0 0 1 0 0 740]], shape=(10, 10), dtype=int32)

Accuracy 97.44%, loss 0.1388 and obtained confusion matrix