

CS671: Deep Learning and Applications  
Programming Assignment 3

RAUNAV GHOSH(M.Tech-CSP)  
T22104

PRASHANT DHANANJAY KULKARNI(M.Tech-CSE)  
T22058

SACHIN BAHULEYAN(M.Tech-CSE)  
T22060

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Backpropagation Algorithm . . . . .	3
1.2	Optimizer . . . . .	3
<b>2</b>	<b>Problem Statement</b>	<b>5</b>
<b>3</b>	<b>Methodology</b>	<b>5</b>
<b>4</b>	<b>Results</b>	<b>7</b>
<b>5</b>	<b>Best architecture</b>	<b>16</b>
<b>6</b>	<b>Inferences</b>	<b>17</b>

# 1 Introduction

## 1.1 Backpropagation Algorithm

Backpropagation adjusts weights between neurons to minimize error between predicted and actual output in artificial neural networks. It has two phases: forward propagation computes output, compares with desired output, and calculates error, while backward propagation propagates error back through layers, updates weights using chain rule of differentiation. The iterative process continues until error is minimized.

Backpropagation is used in deep learning to compute gradients of loss function, which are then used to update parameters to minimize loss. Optimization algorithms are used to update parameters based on the gradients computed by backpropagation.

However, the backpropagation algorithm itself does not specify how to update the parameters of the network. This is where optimization algorithms come in. An optimization algorithm is used to update the parameters of the network based on the gradients computed by the backpropagation algorithm.

## 1.2 Optimizer

Various optimization algorithms have been proposed over the years. Some of them are mentioned below which have been tested in our experimentation in this assignment.

1. **Stochastic gradient descent** - This is done by calculating the gradient of the cost function with respect to each parameter in the model, and then updating the parameters in the opposite direction of the gradient.
2. **Batch Gradient descent** - it minimizes the loss function of a model during training. It updates the parameters of the model after each training batch. The updates are done on a random subset of the training data. This randomness helps escape the local minima hence it would reach a better solution.
3. **Stochastic Gradient Descent with momentum (generalized delta rule)** - SGD updates the weights based on the gradient of the loss function with respect to the weights. This process can be slow and would take a lot of time to converge. To overcome this issue, the SGD with momentum algorithm is often used. The momentum technique accelerates the learning process by accumulating an exponentially decaying moving average of past gradients and using it to update the weights.

$$\Delta w_{jk}^{(0)}(m) = -\eta \frac{\partial E_n(m)}{\partial w_{jk}^{(o)}(m)} + \alpha \Delta w_{jk}^{(0)}(m-1)$$
$$\Delta w_{ij}^{(h)}(m) = -\eta \frac{\partial E_n(m)}{\partial w_{ij}^{(h)}(m)} + \alpha \Delta w_{ij}^{(j)}(m-1)$$

4. **SGD with momentum (NAG)** - This is another variant of SGD to accelerate the process of convergence. Here we use the term "look ahead". A look-ahead is an approximation of where the parameter would be in the next step. The gradient is then calculated at the look-ahead location. This combines the momentum of the previous iteration and leads to faster convergence.

$$\begin{aligned}w_{\text{lookahead}} &= w(m) + \alpha \Delta w(m-1) \\ \Delta w(m) &= \alpha \Delta w(m-1) - \eta \nabla w_{\text{lookahead}} \\ w(m+1) &= w(m) + \Delta w(m) \\ w(m+1) &= w(m) + [\alpha \Delta w(m-1) - \eta \nabla w_{\text{lookahead}}]\end{aligned}$$

5. **AdaGrad** - Adaptive gradient is an optimization algorithm. AdaGrad adjusts the learning rate for each parameter based on the frequency of updates and the magnitude of the gradients, giving a larger learning rate to less frequently updated parameters and a smaller learning rate to frequently updated parameters. It handles sparse features better than momentum based optimizers. This can help avoid vanishing or exploding gradients and can lead to faster convergence. The learning rate decreases over time for each parameter, which can prevent the algorithm from overshooting the optimal solution.

$$\begin{aligned}v_l(m) &= v_l(m-1) + (\nabla w_l(m))^2 \quad \forall l = 1 \dots L \\ w_l(m+1) &= w_l(m) - \frac{\eta}{\sqrt{v_l(m)} + \varepsilon} \nabla w_l(m) \quad \forall l = 1 \dots L\end{aligned}$$

$\varepsilon$  is small positive value such as  $10^{-8}$

6. **RMSPProp** - RMSProp (Root Mean Square Propagation) is an optimization algorithm for updating the parameters of a neural network during training. The main idea behind RMSProp is to adjust the learning rate for each parameter based on the historical average of the squared gradients. In other words, it scales the learning rate by a moving average of the magnitude of recent gradients. This helps to avoid the oscillations that can occur when using a fixed learning rate in SGD, especially in the presence of sparse gradients. It is improvement on AdaGrad which suffers from vanished learning by adding a decay term into the learning rate.

$$\begin{aligned}v_l(m) &= \beta v_l(m-1) + (1-\beta) (\nabla w_l(m))^2 \quad \forall l = 1 \dots L \\ w_l(m+1) &= w_l(m) - \frac{\eta}{\sqrt{v_l(m)} + \varepsilon} \nabla w_l(m) \quad \forall l = 1 \dots L\end{aligned}$$

$\varepsilon$  is small positive value such as  $10^{-8}$

7. **Adam** - The Adam optimizer maintains an exponentially decaying average of past gradients, which is used to calculate the adaptive learning rates for each parameter. It also keeps an exponentially decaying average of past squared gradients, which is used to scale the learning rate for each parameter. Adam uses both momentum and adaptive gradient features to improve convergence.

$$\begin{aligned}
u_l(m) &= \beta_1 u_l(m-1) + (1 - \beta_1) \nabla w_l(m) \quad \forall l = 1 \dots L \\
v_l(m) &= \beta_2 v_l(m-1) + (1 - \beta_2) (\nabla w_l(m))^2 \quad \forall l = 1 \dots L \\
&\quad \text{Bias correction term} \\
\hat{u}_l(m) &= \frac{u_l(m)}{1 - \beta_1^m} \\
\hat{v}_l(m) &= \frac{v_l(m)}{1 - \beta_2^m} \\
w_l(m+1) &= w_l(m) - \frac{\eta}{\sqrt{\hat{v}_l(m) + \varepsilon}} \hat{u}_l(m) \quad \forall l = 1 \dots L \\
&\quad \varepsilon \text{ is small positive value such as } 10^{-8}
\end{aligned}$$

## 2 Problem Statement

Task is to train the fully connected neural network (FCNN) with different hidden layers (minimum number of hidden layers is 3 and maximum is 5) using different optimizers for the backpropagation algorithm and compare the number of epochs that it takes for convergence along with their classification performance.

1. Stochastic gradient descent (SGD) algorithm (batch size = 1)
2. Batch gradient descent algorithm (vanilla gradient descent) – (batch size = total number of training examples)
3. SGD with momentum (generalized delta rule) – (batch size = 1)
4. SGD with momentum (NAG) – (batch size = 1)
5. AdaGrad – (batch size = 1)
6. RMSProp – (batch size = 1)
7. Adam optimizer – (batch size = 1)

## 3 Methodology

- **Given Data** : A subset of the MNIST digit dataset with 5 classes is given. Given dataset is train-validation-test separated. Every image is of the size 28 x 28. Each image is flattened to represent it as a vector of 784-dimension (28 x 28). The dataset has been summarized in Table 1.

Dataset name	No. of samples
Training data	11385
Testing data	3795
Validation data	3795

Table 1: Dataset summary

- **Reading data** : We used the OpenCV library to read and process image data. Images were read as grayscale images and stored in memory as a matrix, i.e., each pixel value is represented by a single intensity value between 0 and 255 in a matrix.
- **Data Preparation** : We scaled the input values from  $[0, 255]$  to  $[0, 1]$ . If the input features have large values, the optimization algorithm updates can become unstable and cause the optimization process to converge slowly or not at all. Scaling the input features helps to avoid this issue and improve the convergence rate of the optimization process.
- **Base Architecture** : Fully Connected Neural Network.
- **Cross entropy loss**: Categorical cross-entropy is a loss function used for classification. It measures the difference between predicted and true probability distributions to minimize loss and improve accuracy. This utilises *Softmax* activation function at the output layer to generate a PMF for the given sample being present in a given class.
- We set the initial weights of a neural network layer using the random normal distribution with mean 0.0, different standard deviations and seed value. This is done by "RandomNormal" initializer in the Keras Initializer API. Poor initialization can lead to vanishing or exploding gradients, which can cause the network to converge slowly or not at all.
- The seed parameter is used to set a random seed value for the random number generator, which ensures that the same random values are generated each time the code is run.
- By initializing each layer with different weights, the network is able to learn a wider variety of features and representations, which can lead to better performance on the task at hand. It can also help prevent the network from getting stuck in a local minimum during training.
- Each architecture of FCNN we experimented on were initialized using the same initial random values of weights.
- **Activation Function**: For the hidden layers *sigmoidal activation function* is used and for the output layer *softmax activation function* has been used.

- **Learning Rate** : For all optimizers learning rate is fixed at 0.001.
- **Optimizers for backpropagation algorithm**
  1. Stochastic gradient descent (SGD) algorithm (batch size = 1)
  2. Batch gradient descent algorithm (vanilla gradient descent) – (batch size = total number of training examples)
  3. SGD with momentum (generalized delta rule) – (batch size = 1)
  4. SGD with momentum (NAG) – (batch size = 1)
  5. AdaGrad – (batch size = 1)
  6. RMSProp – (batch size = 1)
  7. Adam optimizer – (batch size = 1)
- **Stopping criterion**: Using absolute difference between average error of successive epochs to fall below a threshold of  $10^{-4}$ .
- We experimented with different architectures and selected the best architecture based in the validation accuracy.
- In this experimental setup, we have a total of 5 classes but we are using 10 nodes in the output layer. This is for the sole purpose of convenience. Our model uses *sparse categorical cross entropy*, which is same as cross-entropy but the input labels are in *integers* instead of *one-hot vectors*. This takes care of the change of labels to one-hot notation. We can use this because it is guaranteed that the network will never give an invalid class output, i.e., a class which is not present in the training dataset.

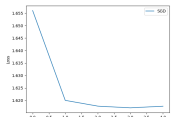
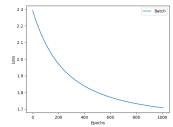
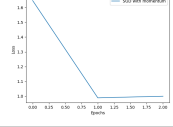
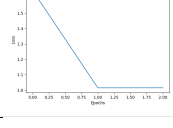
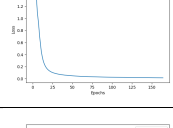
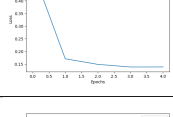
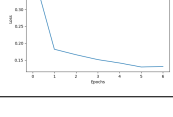
## 4 Results

The various optimisers have been applied on different architectures. The initial value for each of the optimizer have been set using an initialiser such that all the initial parameters remain same.

## 1. Architecture 1

Layer	No. of Nodes	Activation
Input	784	<i>Linear</i>
Hidden 1	400	<i>Sigmoid</i>
Hidden 2	200	<i>Sigmoid</i>
Hidden 3	100	<i>Sigmoid</i>
Output	10	<i>Softmax</i>

Below is the summary of the performance of each optimizer for this architecture:

Optimizer	Epochs	Loss	Train acc.	Val. acc.
SGD	5		0.20000	0.20000
Batch GD	1007		0.20000	0.20000
SGD with momentum	3		0.59657	0.60210
NAG	3		0.54317	0.53570
AdaGrad	192		0.99631	0.97259
RMSProp	5		0.96723	0.96706
Adam	12		0.96873	0.96310



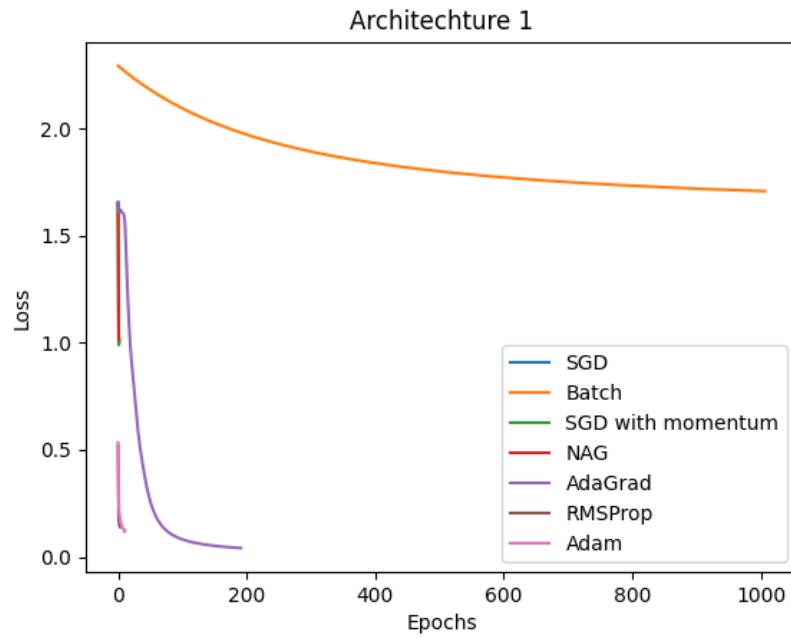


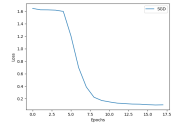
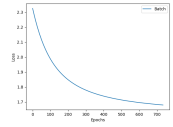
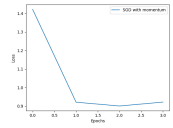
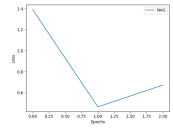
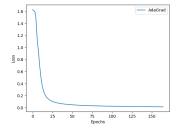
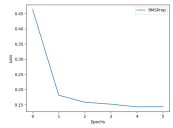
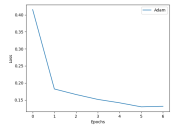
Figure 1: Performance of all optimizers on architecture 1

Figure 1 shows the comparative performance of all optimizers on architecture 1.

## 2. Architecture 2

Layer	No. of Nodes	Activation
Input	784	<i>Linear</i>
Hidden 1	600	<i>Sigmoid</i>
Hidden 2	300	<i>Sigmoid</i>
Hidden 3	200	<i>Sigmoid</i>
Output	10	<i>Softmax</i>

Below is the summary of the performance of each optimizer for this architecture:

Optimizer	Epochs	Loss	Train acc.	Val acc.
SGD	18		0.97435	0.96679
Batch GD	735		0.20316	0.20184
SGD with momentum	4		0.69925	0.70303
NAG	3		0.72534	0.72384
AdaGrad	165		0.99841	0.97681
RMSProp	6		0.96987	0.96337
Adam	7		0.95335	0.95177

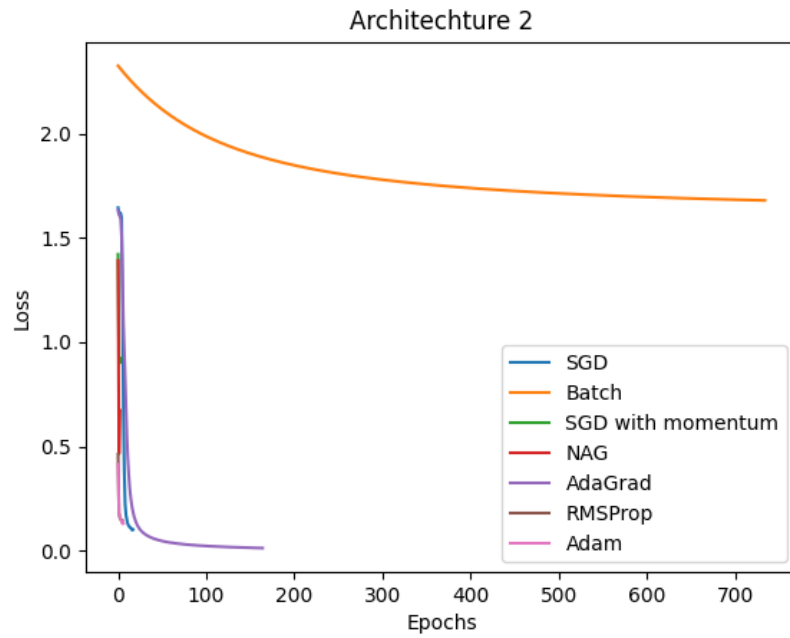


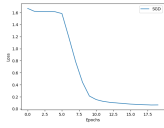
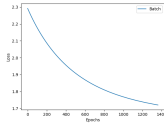
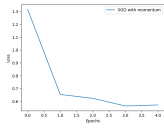
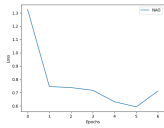
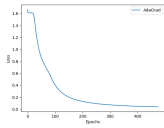
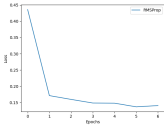
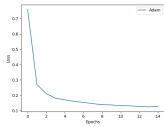
Figure 2: Performance of all optimizers on architecture 2

Figure 2 shows the comparative performance of all optimizers on architecture 2.

### 3. Architecture 3

Layer	No. of Nodes	Activation
Input	784	<i>Linear</i>
Hidden 1	1000	<i>Sigmoid</i>
Hidden 2	500	<i>Sigmoid</i>
Hidden 3	50	<i>Sigmoid</i>
Output	10	<i>Softmax</i>

Below is the summary of the performance of each optimizer for this architecture:

Optimizer	Epochs	Loss	Train acc.	Val acc.
SGD	20		0.98726	0.977075
Batch GD	1363		0.20000	0.20000
SGD with momentum	5		0.77821	0.77127
NAG	7		0.66974	0.67246
AdaGrad	475		0.99991	0.98181
RMSProp	7		0.96846	0.96495
Adam	15		0.950285	0.94598

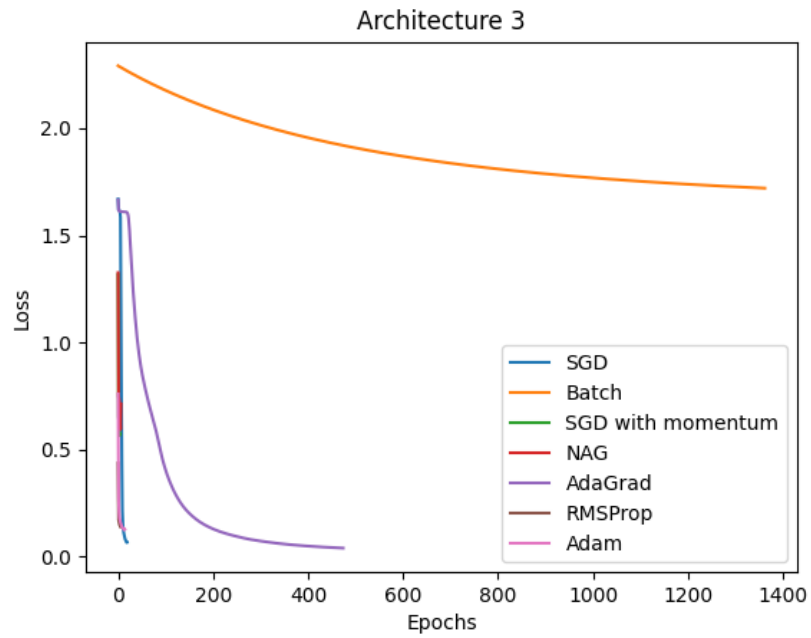


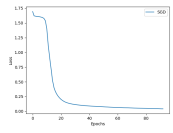
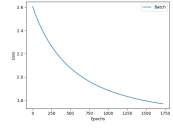
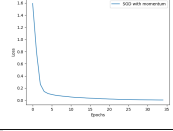
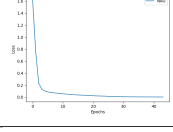
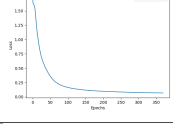
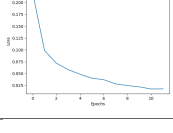
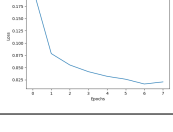
Figure 3: Performance of all optimizers on architecture 3

Figure 3 shows the comparative performance of all optimizers on architecture 3.

#### 4. Architecture 4

Layer	No. of Nodes	Activation
Input	784	<i>Linear</i>
Hidden 1	256	<i>Sigmoid</i>
Hidden 2	128	<i>Sigmoid</i>
Hidden 3	64	<i>Sigmoid</i>
Hidden 4	32	<i>Sigmoid</i>
Output	10	<i>Softmax</i>

Below is the summary of the performance of each optimizer for this architecture:

Optimizer	Epochs	Loss	Train acc.	Val acc.
SGD	93		0.98893	0.97549
Batch GD	1724		0.20000	0.20000
SGD with momentum	35		0.99991	0.98418
NAG	44		0.99991	0.98366
AdaGrad	370		0.98594	0.97733
RMSPProp	12		0.99754	0.98498
Adam	8		0.99771	0.98735

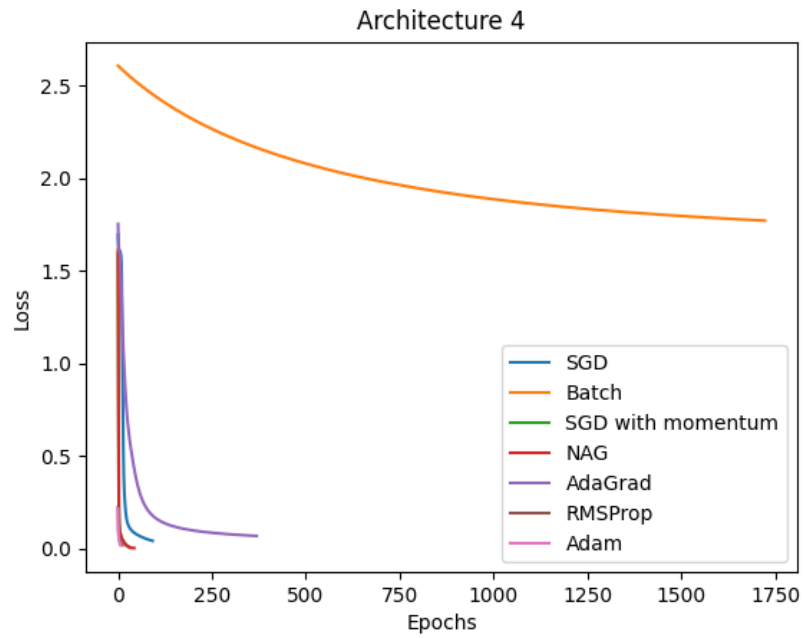


Figure 4: Performance of all optimizers on architecture 4

Figure 4 shows the comparative performance of all optimizers on architecture 4.

## 5 Best architecture

Taking validation accuracy into account, we found that the architecture

Layer	No. of Nodes	Activation
Input	784	<i>Linear</i>
Hidden 1	256	<i>Sigmoid</i>
Hidden 2	128	<i>Sigmoid</i>
Hidden 3	64	<i>Sigmoid</i>
Hidden 4	32	<i>Sigmoid</i>
Output	10	<i>Softmax</i>

is optimal for the classification of the provided MNIST dataset. The measured accuracy on the *validation dataset* is **0.983785** using *Adam* optimizer.

The performance of the architecture on the test data is summarised as below

**Accuracy on test data:** 0.984453

	Pred 0	Pred 1	Pred 2	Pred 6	Pred 7
<b>Actual 0</b>	749	1	6	1	2
<b>Actual 1</b>	1	749	4	2	3
<b>Actual 2</b>	7	3	741	0	8
<b>Actual 6</b>	9	1	4	744	1
<b>Actual 7</b>	0	1	5	0	753

Table 2: Confusion matrix of architecture 4 on Test data

The performance of the architecture on the training data is summarised as below

**Accuracy on train data:** 0.997716

	Pred 0	Pred 1	Pred 2	Pred 6	Pred 7
<b>Actual 0</b>	2273	0	4	1	0
<b>Actual 1</b>	0	2273	3	0	1
<b>Actual 2</b>	0	0	2276	0	1
<b>Actual 6</b>	15	0	0	2262	0
<b>Actual 7</b>	0	1	1	0	2275

Table 3: Confusion matrix of architecture 4 on Test data



## 6 Inferences

- Adam was the best optimizer with fastest convergence at a deeper minima. This is expected due to the presence of both adaptive learning rate and momentum term.
- Close to the performance of Adam was RMSprop who was falling just short of the accuracy by few margins and even performed better in some cases. But Adam was consistent over all the architectures tested.
- AdaGrad started strong in all the architectures but suffered from premature decay of the learning rate which caused the optimizer to slow down by a big factor. Even though the resulting loss curve was much smoother than the non-adaptive learning rate methods, i.e., it didn't suffer from oscillation before convergence, the convergence was rather much slow.
- Both SGD with momentum and NAG overshot a minima and converged into another. This was expected due to the presence of the momentum in the algorithm.
- The optimizers in case of the simple models tend to settle in a shallow minima. This might be due to the low definition of minima in the loss function compared to a more complex model.
- The Batch optimizer is known to have a slower optimization speed compared to other optimizers. As a result, when training various architectures, it did settle at a shallower minimum even after a larger number of epochs.
- Between RMSProp and AdaGrad, there was clear distinction in the performance because of the decay factor( $\beta$ ) in RMSProp that allows it to forget the history of older updates. This also prevents the premature decay of the learning rate.
- SGD, though simple and effective, all the other algorithms were better in finding a minima for the loss function, either in terms of speed(no. of epochs) or the final convergence achieved.
- The use of optimizers with momentum can result in overshooting shallow minima to reach deeper ones, while optimizers with adaptive learning rates tend to exhibit less oscillation and generally settle in a stable minimum before reaching convergence.