SMAI Report

ADAM: A Method For Stochastic Optimisation

Team Number: 5

Team Members: Gautam Ghai(2020101020) ,Aditya Malhotra(2020101052), Mayank Bhardwaj(2020101068) ,Shreyash Jain(2020101006)

1. Introduction

Given that computing first-order partial derivatives with respect to all of the parameters has the same computational complexity as just evaluating the function, gradient descent is a moderately effective optimization strategy when the function is differentiable with respect to its parameters.
Adam is given a stochastic optimization method in the study that just needs first-order gradients and is memory-efficient. The method, known as Adam, derives its name from the calculation of the first and second moments of the gradients, which are used to determine the specific adaptive learning rates for different parameters. Our method combines the advantages of two recently popular methods: RMSProp, which performs well in non-stationary and online environments, and AdaGrad, which performs well with sparse gradients.

2. Related Work

Batch Gradient Descent

1. Uses the gradient of the cost function w.r.t. to the parameters/weights θ to updated the parameters by the equations:

$$\theta = \theta - \eta \nabla J(\theta)$$

2. It is easy to implement and interpret. However, in each step it considers the whole batch at once which makes it slow for large datasets.

Advantage

Very easy to interpret and to implement.

Disadvantage

A single step of this consider the whole batch at once, so it is much slower to obtain any improvement when the dataset is large.

Stochastic Gradient Descent

1. It is the same as batch gradient descent, however it considers one training sample per step as given below and as a result it takes less time and memory to run:

$$\theta = \theta - \eta \nabla J(\theta; x_i, y_i)$$

2. However, the objective function fluctuates a lot and is not smooth and it may also overshoot the minima depending upon the learning rate.

Advantage

Requires less time and memory, as smaller but frequent updates. It can also find other local minimas.

Disadvantage

The objective is not smooth and fluctuates a lot(high variance). It may also overpass a minima depending on the learning rate.

Momentum

1. The Batch Gradient descent algorithm is modified by adding a momentum term. This smoothens the curve and helps reach convergence faster.
2. The weightage is increased in the relevant direction by considering momentum of the previous step.

3. We use the hyper-parameter **γ** for defining the weightage given to previous step.

$$v_t = \gamma . v_{t-1} + \eta . \nabla J(\theta)$$
$$\theta = \theta - v_t$$

4. Addition of another hyper-parameter to choose. If momentum is too high, it can overpass the local minima.

Advantage

It smoothens the curve (lower variance) of the objective function and convergence is reached faster.

Disadvantage

Addition of one more hyper-parameter, which has to be chosen properly. Also if momentum is too high it can over pass the local minima.

Nesterov Accelerated Gradient

1. In the momentum method if the momentum is too high, it can overshoot the local minima. To overcome this, we use the momentum of the just previous step while updating the parameters i.e. we are looking one step ahead in the relevant direction:

$$v_t = \gamma . v_{t-1} + \eta . \nabla J(\theta - \gamma . v_{t-1})$$
$$\theta = \theta - v_t$$

2. Due to this one step look ahead, it will converge faster than only momentum approach. It doesn’t overshoot the local minima as updates will slow down when reaching the minima.

3. However, just like the momentum approach we have to be careful in choosing the hyperparameters.

Advantage

It converges even faster than only *momentum approach .* It will not skip a local minima and the updates will slow down when reaching a minima.

Disadvantage

The hyper-parameter has to be chosen properly.

Adagrad

1. For all previous algorithms, learning rate is kept constant. Which is a problem as when we reach closer to the minima, as we need smaller steps, and away from the minima we need bigger steps.

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

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

2. Here, G is a diagonal matrix where the values are sum of the squares of the gradients (g) upto time (t). i denotes the sample number. ϵ is very small positive number.

Advantages

1. It makes larger update for less frequent parameters (like which are further than minima) and smaller updates for frequent parameters.
2. It works well for sparse gradients.

Disadvantage

It is computationally expensive, at every step it has to calculate the gradients and perform operations on all the previous gradients.

RMSprop

1. It is an improvement over Adagrad by taking a moving average of the gradients instead of summing square of all previous gradients.

$$g_t = \nabla J(\theta_t)$$

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

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

2. It gives the advantages of Adagrad and also learning rate does not decay during the later steps, due to which the learning doesn't stop.
3. It is computationally expensive and we have to be careful about the tuning of hyperparameters.

3. Adam

Adam can be viewed as an RMSprop and stochastic gradient descent with momentum combination. Like RMSprop, it scales the learning rate using squared gradients and, like SGD with momentum, it takes advantage of momentum by using the moving average of the gradient rather than the gradient itself. As an adaptive learning rate method, Adam calculates individual learning rates for various parameters. Adam employs estimates of the first and second moments of the gradient to change the learning rate for each weight of the neural network, which is how it gets its name, adaptive moment estimation. Nth moment of a random variable is defined as the expected value of that variable to the power of n.

To estimate the moments, Adam utilizes exponentially moving averages, computed on the gradient evaluated on a current mini-batch:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$m_t = (1 - \beta_1) \sum_{i=0}^{t} \beta_1^{t-i} g_i$$

Now, let’s take a look at the expected value of m, to see how it relates to the true first moment, so we can correct for the discrepancy of the two:

$$\begin{aligned} E[m_t] &= E[(1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i} g_i] \\ &= E[g_i](1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i} + \zeta \\ &= E[g_i](1 - \beta_1^t) + \zeta \end{aligned}$$

Because the approximation is taking place, the error *ζ* emerges in the formula. In the last line, we just use the formula for the sum of a finite geometric series. There are two things we should note from that equation.

- We have *a biased estimator*. This is not just true for Adam only, the same holds for algorithms, using moving averages (SGD with momentum, RMSprop, etc.).

- It won’t have much effect unless it’s the begging of the training, because the value *β* to the power of *t* is quickly going towards zero.

Now we need to correct the estimator so that the expected value is the one we want. This step is usually referred to as **bias correction**. The final formulas for our estimator will be as follows:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \quad , \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

To perform weight update we do the following:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Where *w* is model weights, *η* is the step size (it can depend on iteration). That’s the update rule for Adam.

Adam Implementation

We created a class for Adam optimizer inheriting `optimizer_v2.OptimizerV2` from Tensorflow, and made relevant functions following Tensorflow API.

The default values of the parameters are :

1. Learning Rate = 0.001
2. Beta 1 = 0.9
3. Beta 2 = 0.999

We have set the value of alpha which is step size upper bound parameter equal to square root of ((1 - beta_2_power) / (1 - beta_1_power))

```
def _prepare_local(self, var_device, var_dtype, apply_state):
    super()._prepare_local(var_device, var_dtype, apply_state)

    local_step = tf.cast(self.iterations + 1, var_dtype)
    beta_1_t = tf.identity(self._get_hyper("beta_1", var_dtype))
    beta_2_t = tf.identity(self._get_hyper("beta_2", var_dtype))
    beta_1_power = tf.pow(beta_1_t, local_step)
    beta_2_power = tf.pow(beta_2_t, local_step)
    lr = apply_state[(var_device, var_dtype)]["lr_t"] * (
        tf.sqrt(1 - beta_2_power) / (1 - beta_1_power)
    )
    apply_state[(var_device, var_dtype)].update(
        dict(
            lr=lr,
            epsilon=tf.convert_to_tensor(self.epsilon, var_dtype),
            beta_1_t=beta_1_t,
            beta_1_power=beta_1_power,
            one_minus_beta_1_t=1 - beta_1_t,
            beta_2_t=beta_2_t,
            beta_2_power=beta_2_power,
            one_minus_beta_2_t=1 - beta_2_t,
        )
    )
```

The above function creates a dictionary with the updated values. These variables are updated using `tf.raw_ops.ResourceApplyAdam()` function.

Implementation

We have 3 models

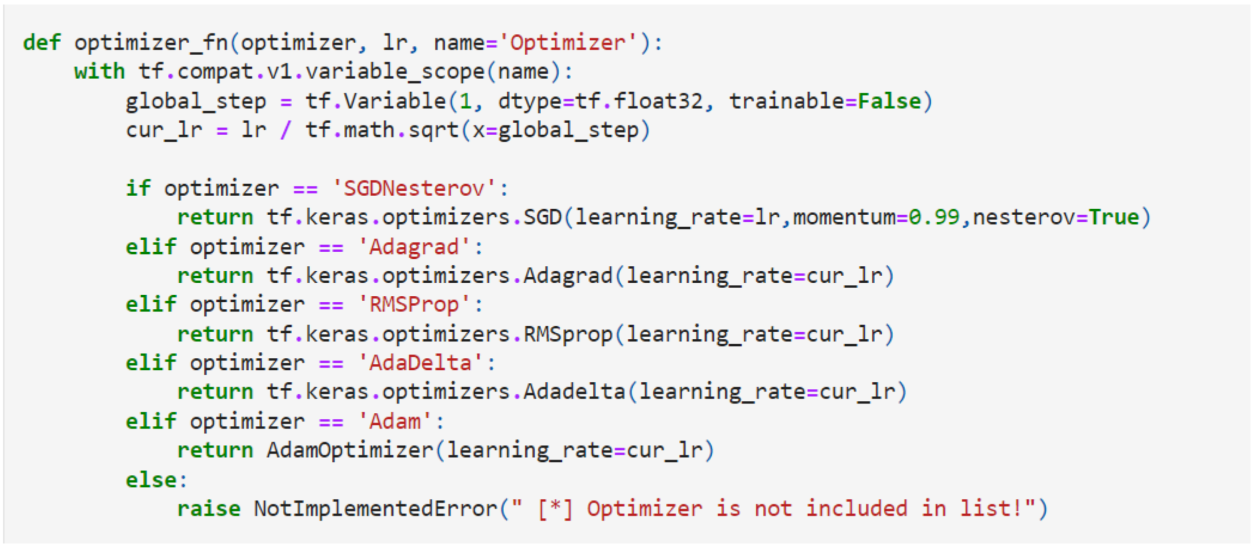
1. Logistic Regression
2. Neural Network
3. Convolutional Neural Network

For each model there is different file. The logistic regression uses version 1 of tensorflow and the other 2 uses the version 2.

Code Flow

1. A function was written to create a basic model, the optimizer name was passed to that function.

2. Using a loop each optimizer was passed to that function one by one.
3. A function was created to return the optimizer.



```
def optimizer_fn(optimizer, lr, name='Optimizer'):
    with tf.compat.v1.variable_scope(name):
        global_step = tf.Variable(1, dtype=tf.float32, trainable=False)
        cur_lr = lr / tf.math.sqrt(x=global_step)

        if optimizer == 'SGDNesterov':
            return tf.keras.optimizers.SGD(learning_rate=lr,momentum=0.99,nesterov=True)
        elif optimizer == 'Adagrad':
            return tf.keras.optimizers.Adagrad(learning_rate=cur_lr)
        elif optimizer == 'RMSProp':
            return tf.keras.optimizers.RMSprop(learning_rate=cur_lr)
        elif optimizer == 'AdaDelta':
            return tf.keras.optimizers.Adadelta(learning_rate=cur_lr)
        elif optimizer == 'Adam':
            return AdamOptimizer(learning_rate=cur_lr)
        else:
            raise NotImplementedError(" [*] Optimizer is not included in list!")
```

The function to return the optimizer based upon the arguments that are name of the optimizer and learning rate

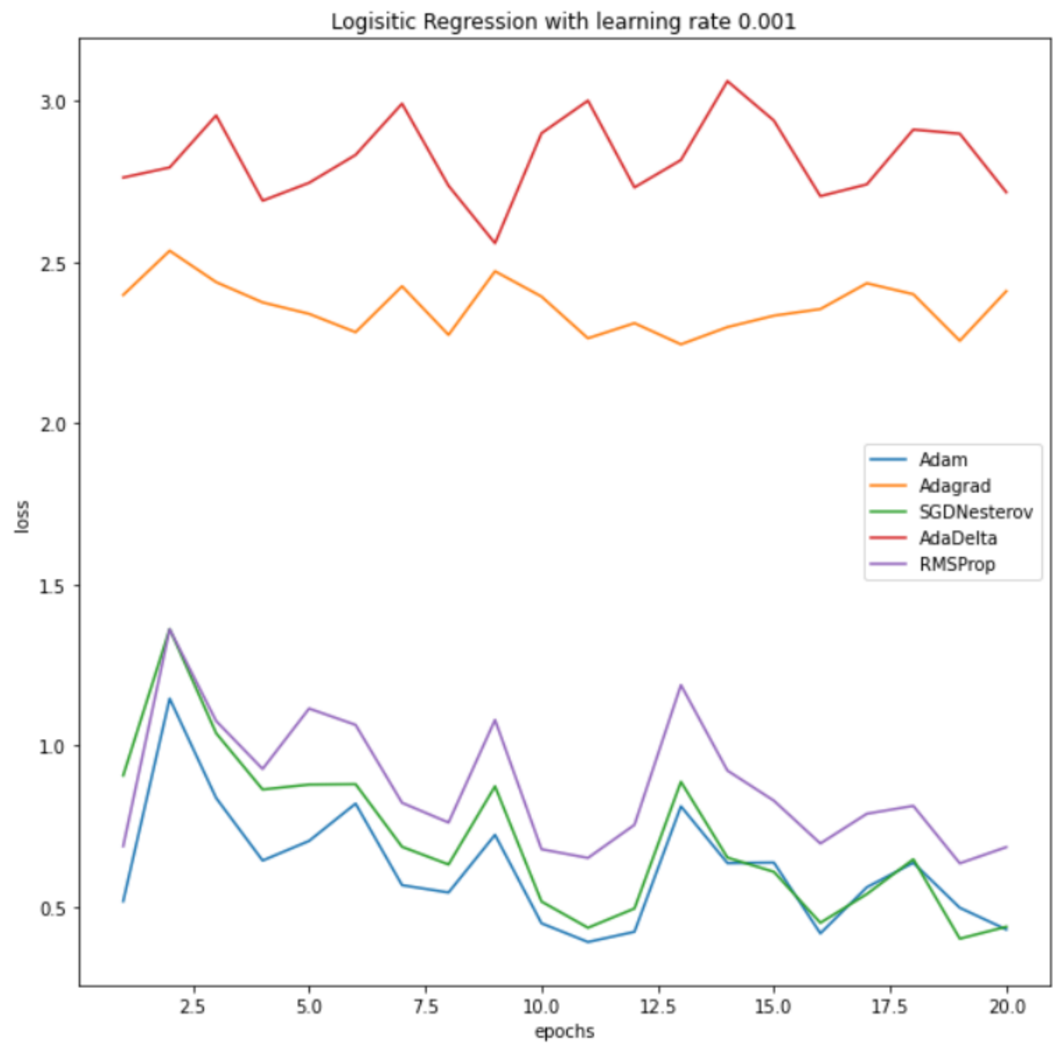
For Adam Optimizer a class was written, for the other optimizers the builtin functions are used.

Logistic Regression

The dataset used is MNIST

```
X = tf.placeholder(tf.float32, [None, 784])
Y = tf.placeholder(tf.float32, [None, 10])
W = tf.Variable(0.1 * np.random.randn(784, 10).astype(np.float32))
B = tf.Variable(0.1 * np.random.randn(10).astype(np.float32))
Y_predicted = tf.nn.softmax(tf.add(tf.matmul(X, W), B))
loss = tf.reduce_mean(-tf.reduce_sum(Y * tf.log(Y_predicted), axis=1))
optimizer = optimizer_fn(optimizer,learning_rate,loss)
with tf.Session () as sess:
  sess.run ( tf.global_variables_initializer ( ) )
  for epoch in range(epochs):
    for i in range(batches):
      offset = i * epoch
      x = x_train[offset: offset + batch_size]
      y = y_train[offset: offset + batch_size]
      sess.run(optimizer, feed_dict={X: x, Y:y})
      c = sess.run(loss, feed_dict={X:x, Y:y})
```

The code for logisitic regression



These are results obtained for 5 different optimizer. Adam and SGDNesterov performed the best whereas Adagrad and AdaDelta had poor loss.

Neural Network:

The dataset used is Fashion MNIST.

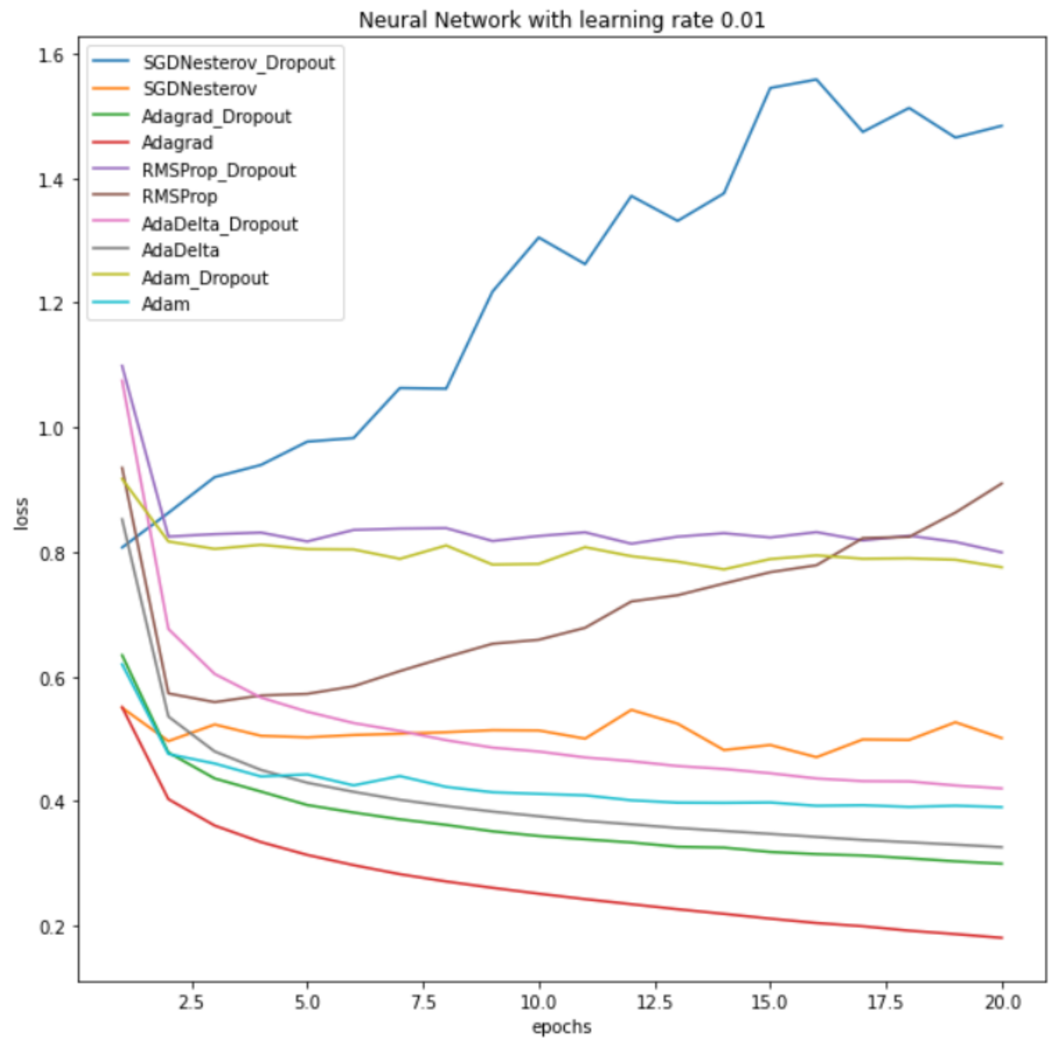
There are 2 hidden layers with 1000 nodes each. The activation function used for these hidden layers is relu and activation function for output layer is softmax.

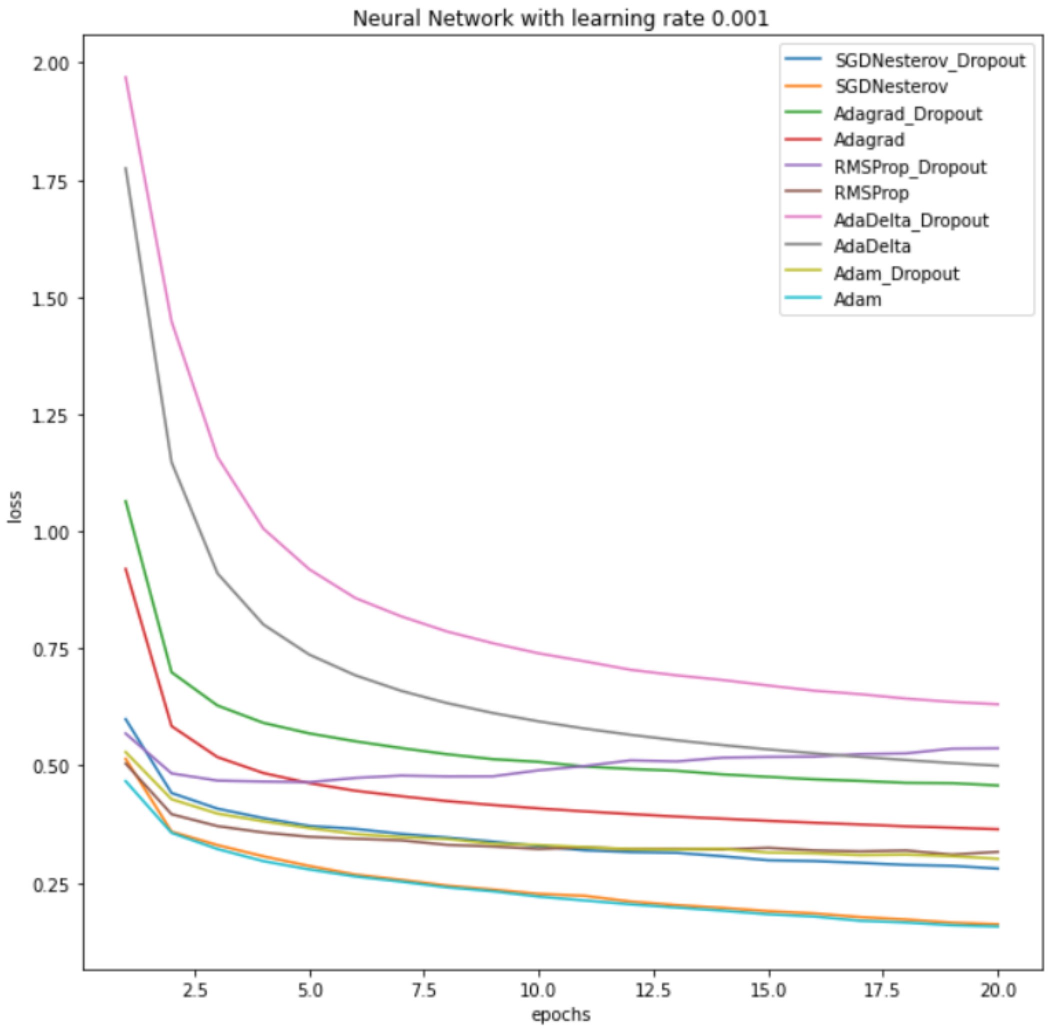
For each optimizer we have tried 2 methods with dropout and without dropout.

The following code handles creates the model

```
def getModel(dropout,rate=0.2):
    if dropout:
        model = keras.Sequential([
                keras.layers.Flatten(input_shape=(28,28)),
                keras.layers.Dropout(rate),
                keras.layers.Dense(1000, activation="relu"),
                keras.layers.Dropout(rate),
                keras.layers.Dense(1000, activation="relu"),
                keras.layers.Dropout(rate),
                keras.layers.Dense(10, activation="softmax")
                ])
        return model
    else:
        model = keras.Sequential([
                keras.layers.Flatten(input_shape=(28,28)),
                keras.layers.Dense(1000, activation="relu"),
                keras.layers.Dense(1000, activation="relu"),
                keras.layers.Dense(10, activation="softmax")
                ])
        return model
```

Comparison after changing the learning rate

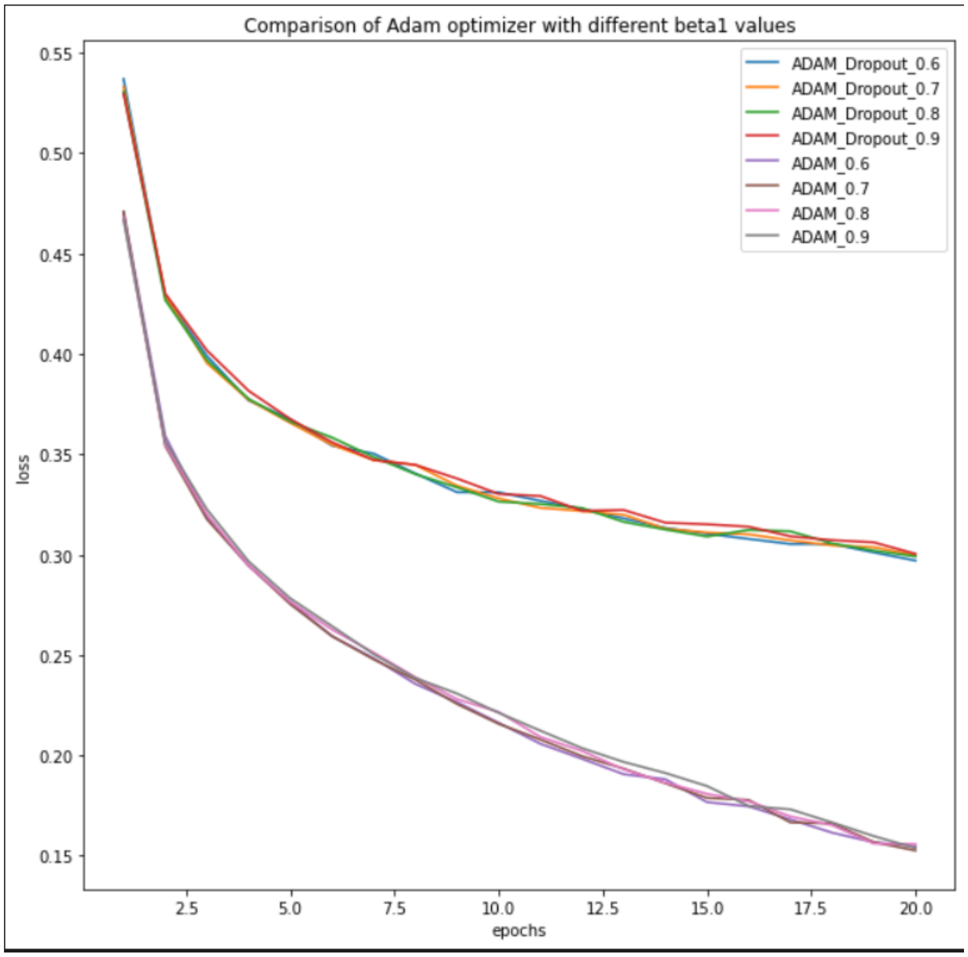

Neural Network with learning rate 0.01
SGDNesterov_Dropout
SGDNesterov
Adagrad_Dropout
Adagrad
RMSProp_Dropout
RMSProp
AdaDelta_Dropout
AdaDelta
Adam_Dropout
Adam
loss
epochs
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0



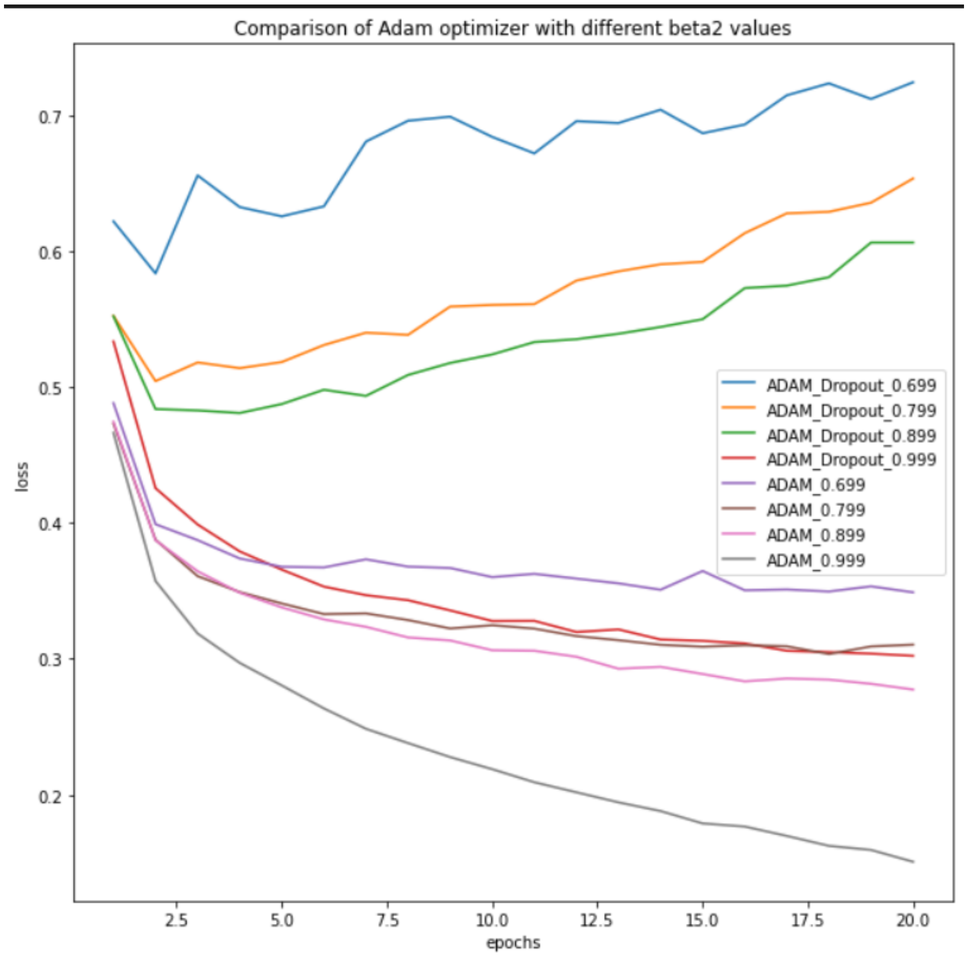
Better results are observed when learning rate is 0.001. Adam outperformed every other optimizer when the learning rate is 0.001 whereas in 0.01 it is observed that Adam optimizer is not the best. Due to a higher learning rate it is possible that the values converge to a suboptimal value/local minima.

Even though it is observed that optimizer work better without dropout but using dropout we can prevent overfitting.

Changing values of Beta 1



Changing values of Beta 2

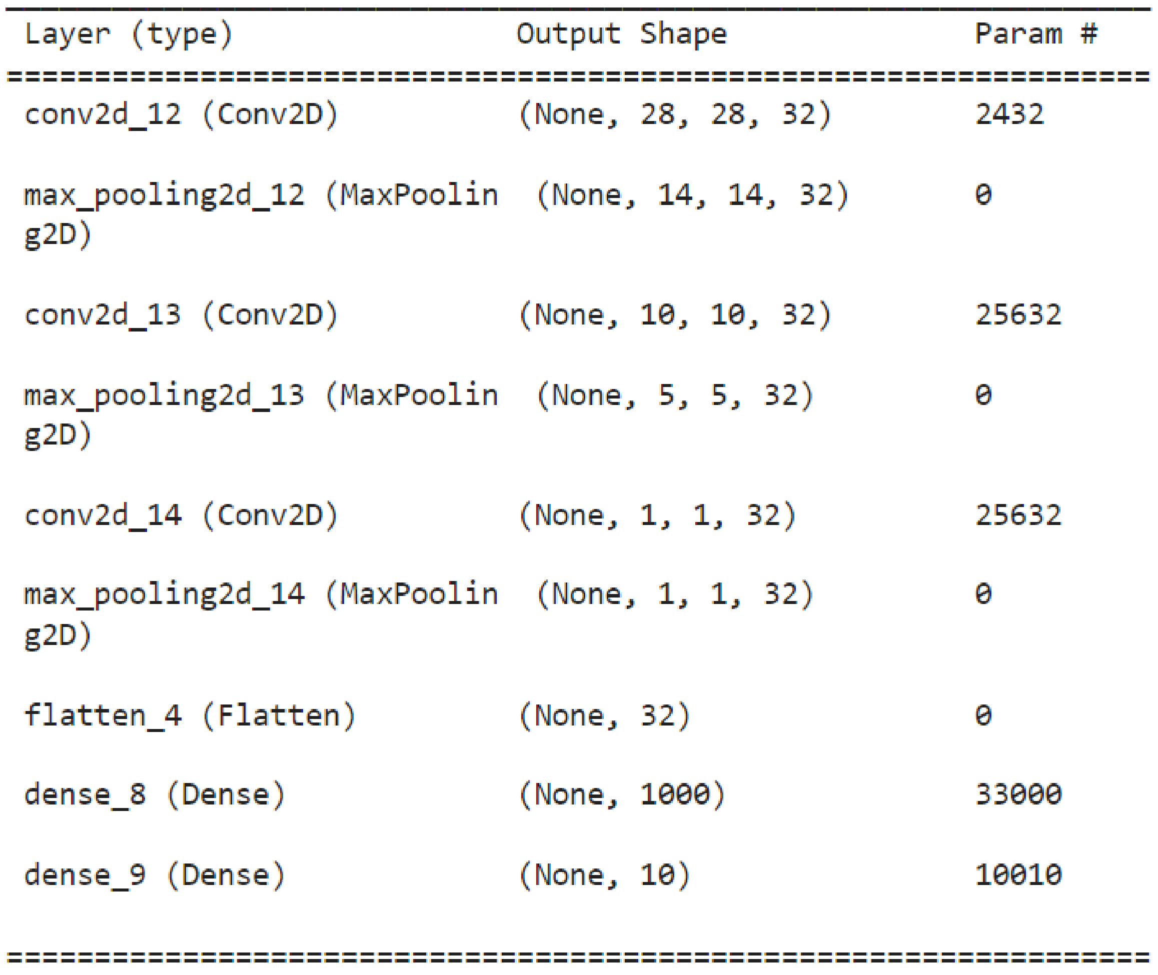


Convolutional Neural Network

The dataset used is CIFAR10. The model contains 3 layers with alternating convolutional and MaxPooling layer. The activation function used is relu. After this the resulting images are flattened and 1 hidden layer of 1000 nodes is used with activation function as relu. The output layer has 10 nodes.

```
model = keras.models.Sequential()
for i in range(3):
  model.add(layers.Conv2D(32, (5,5), strides=(1,1), padding="valid", activation='relu', input_shape=(32,32,3)))
  model.add(layers.MaxPool2D(pool_size=(3, 3),strides=(2,2), padding="same"))
model.add(layers.Flatten())
if(dropout):
  model.add(layers.Dropout(0.2))
model.add(layers.Dense(1000, activation='relu'))
model.add(layers.Dense(10))
print(model.summary())
```

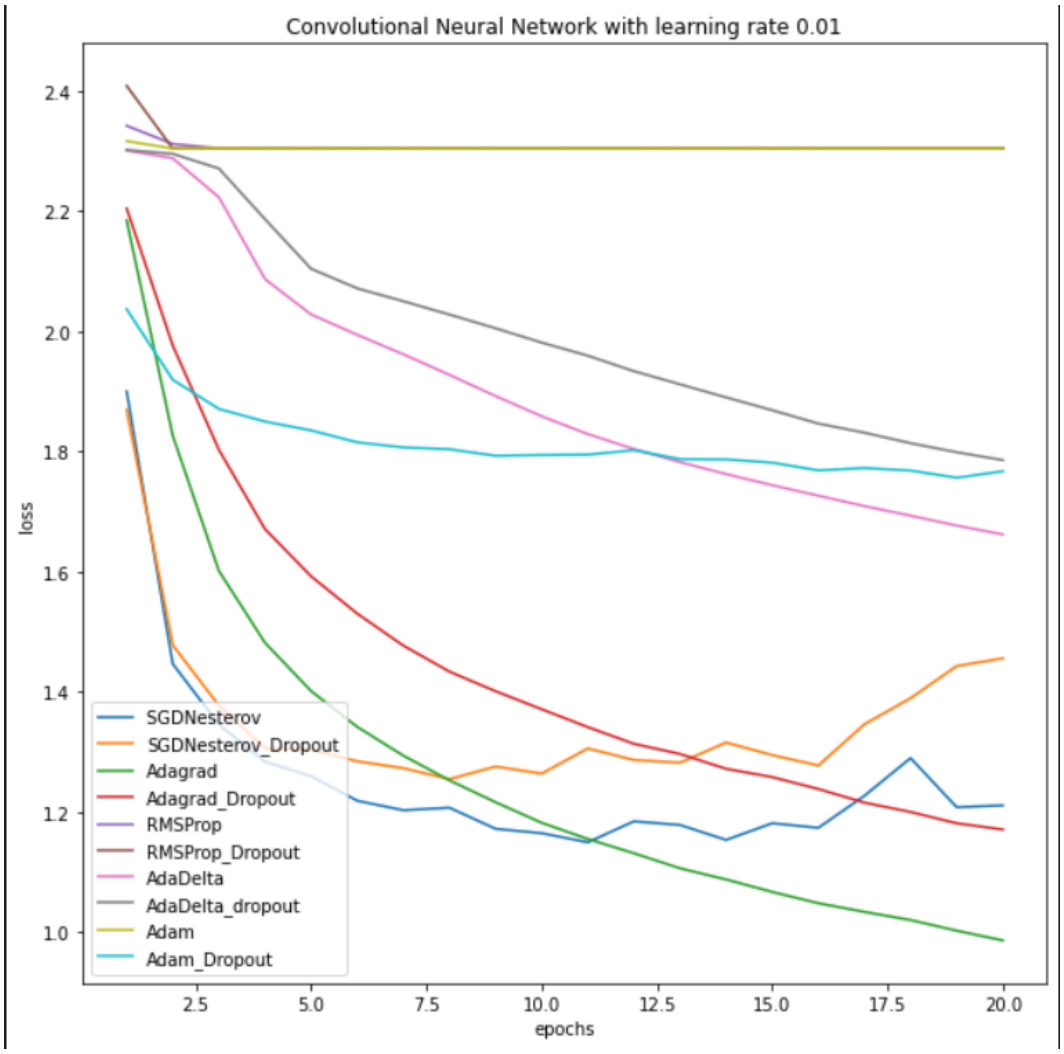
The above code is used to train the model. The batch size is kept as 128.



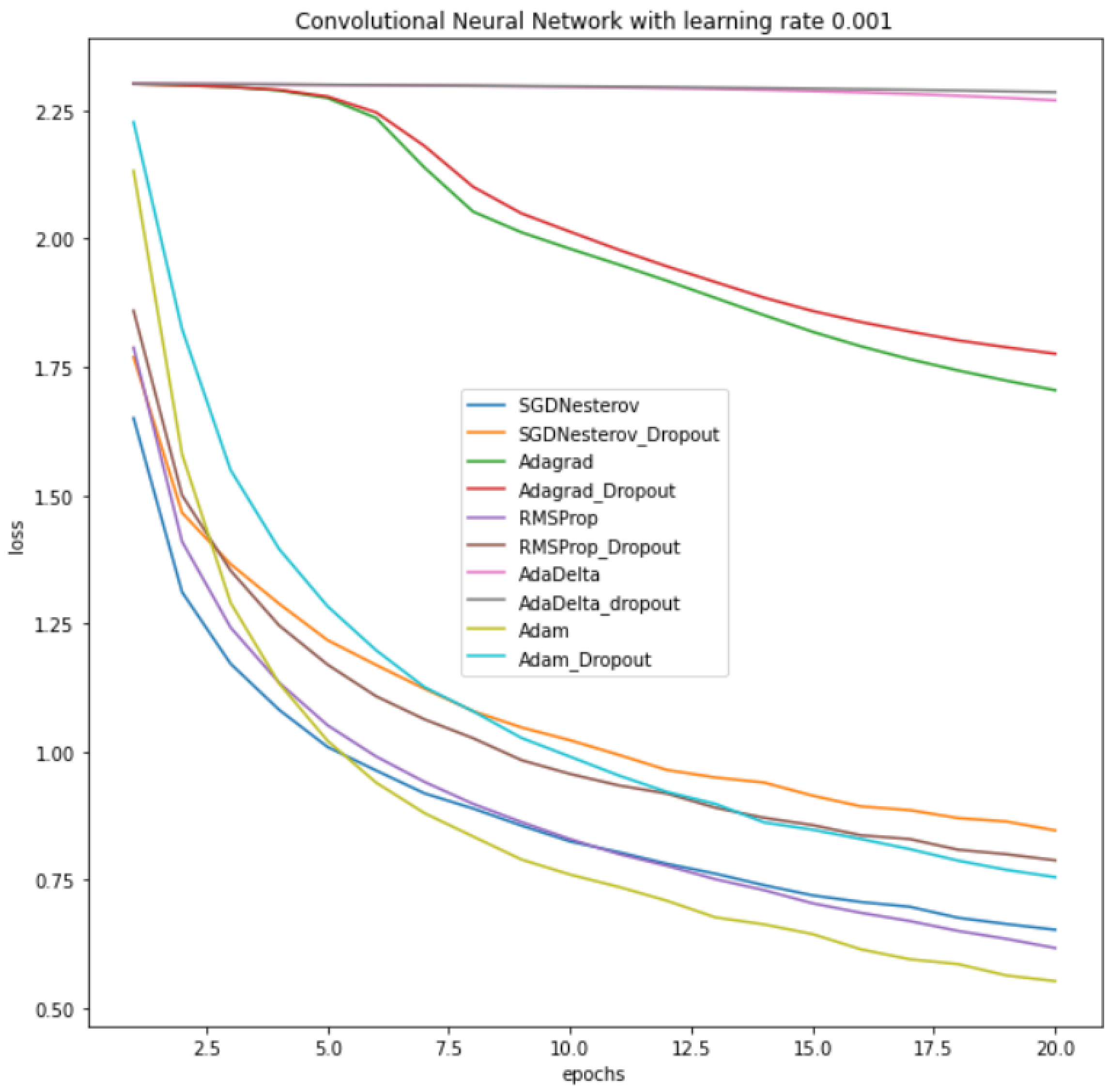
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_12 (Conv2D)          (None, 28, 28, 32)        2432

 max_pooling2d_12 (MaxPoolin  (None, 14, 14, 32)       0
 g2D)

 conv2d_13 (Conv2D)          (None, 10, 10, 32)        25632

 max_pooling2d_13 (MaxPoolin  (None, 5, 5, 32)         0
 g2D)

 conv2d_14 (Conv2D)          (None, 1, 1, 32)          25632

 max_pooling2d_14 (MaxPoolin  (None, 1, 1, 32)         0
 g2D)

 flatten_4 (Flatten)         (None, 32)                0

 dense_8 (Dense)             (None, 1000)              33000

 dense_9 (Dense)             (None, 10)                10010

=================================================================
```

The detailed summary of the model along with the number of parameters to be trained.

The results are obtained on 2 different learning rates 0.01 and 0.001



Convolutional Neural Network with learning rate 0.01
loss
epochs
2.4
2.2
2.0
1.8
1.6
1.4
1.2
1.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
SGDNesterov
SGDNesterov_Dropout
Adagrad
Adagrad_Dropout
RMSProp
RMSProp_Dropout
AdaDelta
AdaDelta_dropout
Adam
Adam_Dropout



Adam performed the best when the learning rate was 0.001.

Future Prospects

Since its release, a lot of work has been done on Adam and various variants of it have been created. Two such variants are AMSGrad and NADAM.

NADAM stands for Nesterov-accelerated Adaptive Moment Estimation which incorporates Nesterov Momentum in Adam and leads to an even better performing optimizer than Adam.

AMSGrad keeps the maximum of all *vt* until the present time step and uses this maximum value for normalizing the running average of the gradient unlike ADAM which uses *vt*.

Summary and Conclusion

Adam proves to be better at finding the minimum and it reaches the state much faster than other optimizers.

- **Changing Learning rate :** It is better to maintain a smaller learning rate since it requires fewer learning steps but is more robust to locate minima than maintaining a bigger learning rate. Although learning can occasionally go more quickly with a higher learning rate.
- **Changing *β*1:** A Lower *β*1 value means using lesser knowledge of history, so keeping a higher value gives a better result as expected.
- **Changing *β*2:** This is similar to *β*1 case, and in every result we found best result at 0.999.

Work Distribution

Shreyash Jain - Basic pipeline of Neural Network , worked on dropouts , ran the scripts on ADA and collected the results , Adam Optimiser Class , Did Logistic Regression

Aditya Malhotra - Convolution Neural Network , made graphs , Adam Optimiser Class , Did Logistic Regression

Gautam Ghai - Did Logistic Regression , Adam Optimiser Class , Literature Review , Neural Network .

Mayank Bhardwaj - Convolution Neural Network , Adam Optimiser Class , Did Logistic Regression , Literature Review .

References

https://towardsdatascience.com/custom-optimizer-in-tensorflow-d5b41f75644a
https://www.tensorflow.org/api_docs/python/tf
https://www.techwithtim.net/tutorials/python-neural-networks/creating-a-model/

https://youtu.be/M2xkmc2oHUc
https://youtu.be/eMMZpas-zX0