Neural Network

Assignment 4: COL341

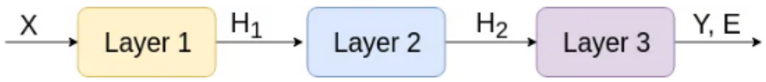
Monu(2020CS50432)

File Structure :

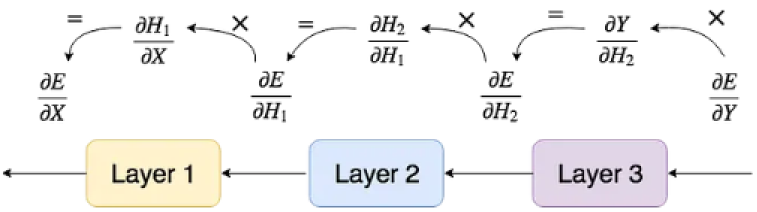
1. Part 1 - -> A4_3.ipynb
2. Part 2 -
   - 4.1 - **Hyper-parameter Tuning**
     1. Learning Rate (LR) -> A4_4_1_1.ipynb
     2. Variation in LR -> A4_4_1_2.ipynb
     3. Number of Training Epochs -> A4_4_1_3.ipynb **&** A4_4_1_1.ipynb
     4. Batch Size -> A4_4_1_4.ipynb
   - 4.2 - **Effect of Loss Function -> A4_4_2.ipynb**
   - 4.3 - **Effect of Data Augmentation -> A4_4_3.ipynb**
3. Part 3 - -> A4_3.ipynb

Part 1 : Implement a Neural Network

Idea : We can design each layer of neural network separately and then at last connect this layer with each other and pass output of one layer to next layer.



Similarly we can also pass gradient backward and increase weight of each layer by backpropagation concept which is mainly based on chain rule.



Implementation Steps : First of all I implemented all required layers with basic structure as shown in fig1 . For any layer during forward pass we will get Y as output. Functionality of forward and backward will be more clear by fig 2 and fig 3. I implemented following layers:

1. Convolutional

2. maxPool
3. ReLU
4. Reshape
5. FullyConnected

```python
class Layer:
    def __init__(self):
        self.input = None
        self.output = None

    def forward(self, input):
        pass

    def backward(self, output_gradient, learning_rate):
        pass
```

Fig 1 : Basic Layer structure

$$X \rightarrow \boxed{\text{layer}} \rightarrow Y$$

Fig 2 : layer.forward()

$$\frac{\partial E}{\partial X} \leftarrow \boxed{\text{layer}} \leftarrow \frac{\partial E}{\partial Y}$$

Fig 3 : layer.backward()

After layer implementation, I designed model for the given neural network.(fig 4)

Results :

I get Value Error for CONV1 layer during backpropagation. I searched for it but didn't get any solution.

```
ValueError: For 'valid' mode, one must be at least as large as the other in every dimension
```

```
class Net:
  def __init__(self):
    self.conv1 = Convolutional((3,32,32),3,32)  # Kernel size =
    self.pool1 = maxPool((32,30,30),2,1)      # Kernel size = 2x2
    self.conv2 = Convolutional((32,29,29),5,64) # Kernel size =
    self.pool2 = maxPool((64,25,25),2,1)      # Kernel size = 2x2
    self.conv3 = Convolutional((64,24,24),3,64)
    self.reshape = Reshape((64,22,22),(64*22*22,1))
    self.fc1 = FullyConnected(64*22*22,64)
    self.fc2 = FullyConnected(64,10)
    self.relu1 = ReLU()
    self.relu2 = ReLU()
    self.relu3 = ReLU()
    self.relu4 = ReLU()

  def forward(self,input):
    x = self.conv1.forward(input)
    x = self.relu1.forward(x)
    x = self.pool1.forward(x)
    x = self.conv2.forward(x)
    x = self.relu2.forward(x)
    x = self.pool2.forward(x)
    x = self.conv3.forward(x)
    x = self.relu3.forward(x)
    x = self.reshape.forward(x)
    x = self.fc1.forward(x)
    x = self.relu4.forward(x)
    x = self.fc2.forward(x)
    return x

  def backward(self, out_grad , learning_rate):
      x = self.fc2.backward(out_grad , learning_rate)
      x = self.relu4.backward(x , learning_rate)
      x = self.relu4.backward(x , learning_rate)
      x = self.fc1.backward(x , learning_rate)
      x = self.reshape.backward(x , learning_rate)
      x = self.relu3.backward(x , learning_rate)
      x = self.conv3.backward(x , learning_rate)
      x = self.pool2.backward(x , learning_rate)
      x = self.relu2.backward(x , learning_rate)
      x = self.conv2.backward(x , learning_rate)
      x = self.pool1.backward(x , learning_rate)
```

Fig 4: Model for given Neural Network

Part 2 : PyTorch Implementation

4.1 Hyper-parameter Tuning

1. **Learning Rate (LR) :**

   Parameters: Epoch : 10 , Batch Size : 4

   I had taken 4 values for LR = {0.001,0.005,0.01,0.05} .

As we are increasing Learning rate Accuracy decreases with other parameters constant. And loss is increasing with increase in learning rate.

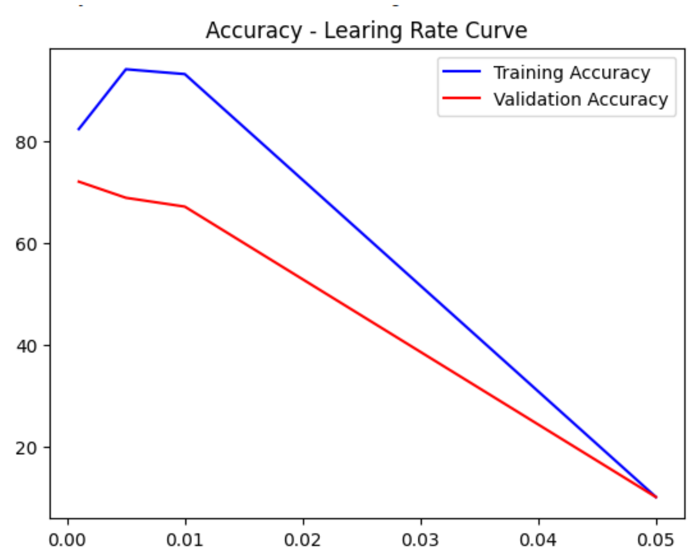


Fig : Accuracy with Learning Rate Curve

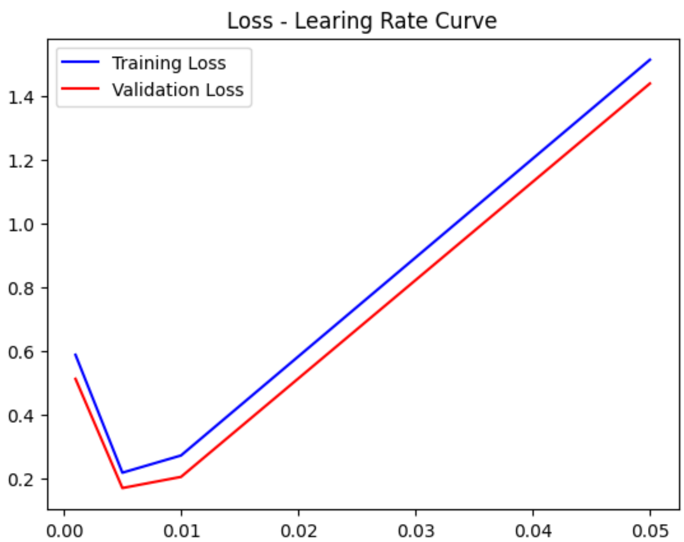


Fig : Loss with Learning Rate Curve

We can observe following points from Class Wise Accuracy Data for LR = 0.005 and 0.05 :

i) When we increase learning rate very much then model will not be able learn complete data and loops over some local minima. Here for LR =0.05, our model give 100% accuracy for plane but didn't learn other data. This is because of small step size.

ii) For LR = 0.005 or 0.001, Loss for both validation and train data decrease with the number of Epoch. But it is not true for LR = 0.05.

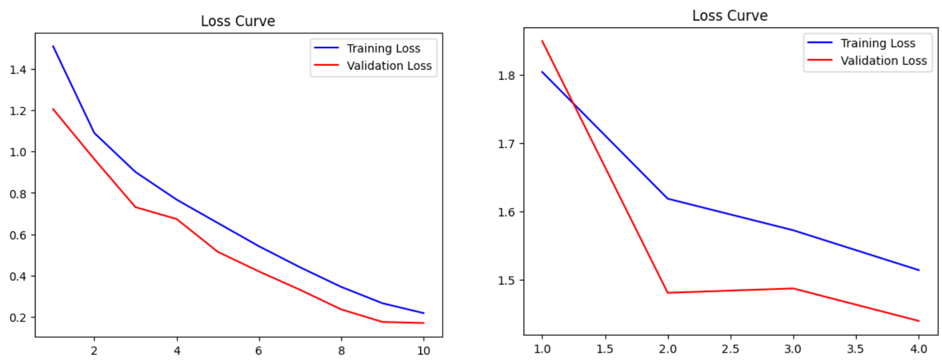
```
Class waise Accuracy on Val Data
Accuracy for class: plane is 75.4 %
Accuracy for class: car   is 75.4 %
Accuracy for class: bird  is 60.0 %
Accuracy for class: cat   is 46.4 %
Accuracy for class: deer  is 64.5 %
Accuracy for class: dog   is 69.8 %
Accuracy for class: frog  is 74.4 %
Accuracy for class: horse is 75.2 %
Accuracy for class: ship  is 71.0 %
Accuracy for class: truck is 76.8 %
```

(a) LR = 0 .005

```
Class waise Accuracy on Val Data
Accuracy for class: plane is 100.0 %
Accuracy for class: car   is 0.0 %
Accuracy for class: bird  is 0.0 %
Accuracy for class: cat   is 0.0 %
Accuracy for class: deer  is 0.0 %
Accuracy for class: dog   is 0.0 %
Accuracy for class: frog  is 0.0 %
Accuracy for class: horse is 0.0 %
Accuracy for class: ship  is 0.0 %
Accuracy for class: truck is 0.0 %
```

(b) LR = 0 .05

Fig : Classwise Accuracy



(a) LR = 0 .005 (b) LR = 0 .05

Fig : Loss with Epoch Curve

2. Variation in LR:
   Parameters: Epoch : 10 , Batch Size : 4
   For this part I tried ReduceLROnPlateau() which is similar to step decrease.

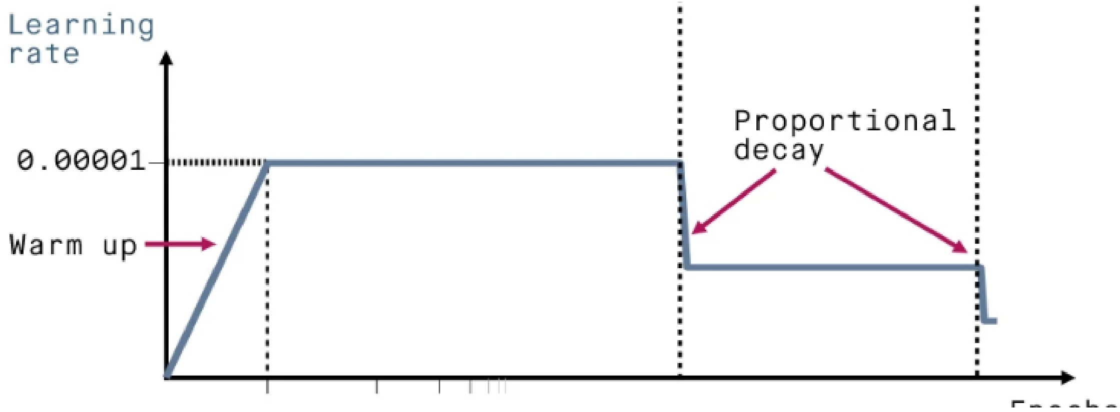
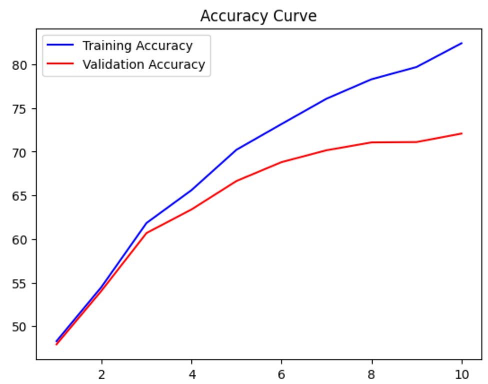
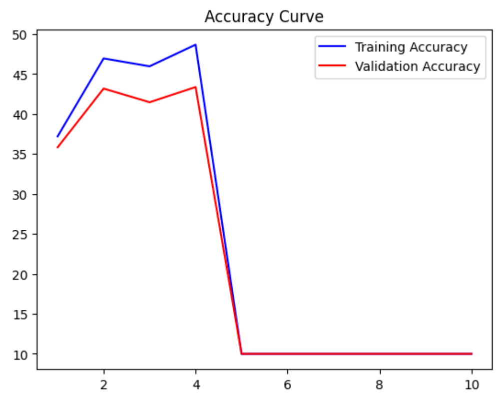


Fig : Accuracy with Epochs for varying LR

3. Number of Epochs :
   For appropriate learning rate Accuracy increases with increase in Epochs. But for very large learning rates, increasing the number of epochs will not help in increase in accuracy.



(a)LR = 0 .001



(b) LR = 0 .05

Fig : Accuracy with number of epochs(from A4_4_1_1.ipynb)

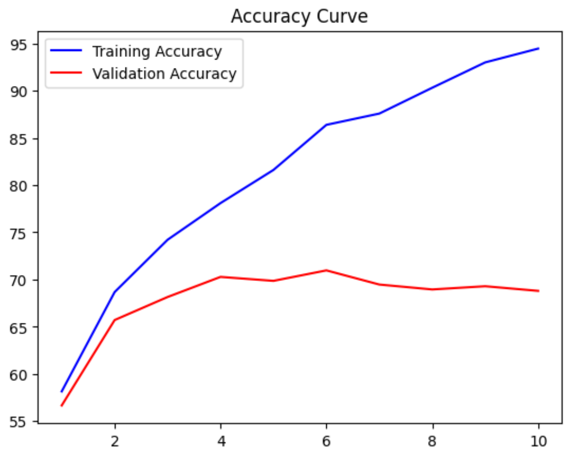


Fig : Increase in epoch will not help always

4. Batch Size :

Both training as well as test accuracy decrease with increase in batch size. This is because in most implementations the loss and hence the gradient is averaged over the batch. This means for a fixed number of training epochs, larger batch sizes take fewer steps. and learning steps are low for higher batch size. That's why Accuracy decreases with increase in batch size.

However, Training time decrease with increase in batch size.

Conclusion : Larger batch sizes will train faster and consume more memory but might show lower accuracy

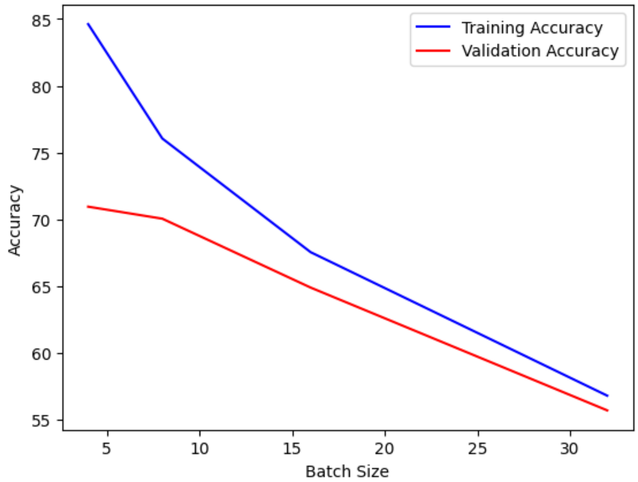


Fig : Accuracy with batch size

4.2 Effect of Loss Function

I varied the loss function and analysed its impacts.

I experiment with following loss functions:

1. Cross-entropy :

$$L(y, \hat{y}) = -\sum y_i \, log(y_i)$$

Accuracy = 69.28 %

2. Hinge Loss

$$L(y, \hat{y}) = \sum max(0, 1 - (y_i * \hat{y}_i))$$

Accuracy = 67.52 %

3. Squared Hinge Loss

$$L(y, \hat{y}) = L(y, \hat{y}) = \sum max(0, 1 - (y_i * \hat{y}_i))^2$$

Accuracy = 68.7 %

4. Kullback-Leibler Divergence

$$L(y, \hat{y}) = -\sum y_i \, log(y_i / \hat{y})$$

Accuracy = 70.72 %

```
Kullback-Leibler Divergence has higher accuracy as compared to
Cross-entropy
```

4.3 Effect of Data Augmentation

As we can see from the accuracy data that not using transformation has very low accuracy as compared with normalise transformation and flipped transformation. Transformations used are :

```
no_transform = transforms.Compose(
    [transforms.ToTensor()
    ])
normal_transform = transforms.Compose(
    [transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

flip_transform = transforms.Compose(
    [transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ,transforms.RandomHorizontalFlip(), transforms.RandomCrop(size=32)
    ])
```

```
==========================================
Using no transformation
Files already downloaded and verified
Files already downloaded and verified
Data Loaded
 Epoch: 1 loss: 1.669
 Epoch: 2 loss: 1.243
 Epoch: 3 loss: 1.039
 Epoch: 4 loss: 0.892
 Epoch: 5 loss: 0.767
Accuracy for F1 is  47.83
==========================================
Using normalize transformation
Files already downloaded and verified
Files already downloaded and verified
Data Loaded
 Epoch: 1 loss: 1.492
 Epoch: 2 loss: 1.077
 Epoch: 3 loss: 0.882
 Epoch: 4 loss: 0.749
 Epoch: 5 loss: 0.634
Accuracy for F2 is  69.99
==========================================
Using flip transformation
Files already downloaded and verified
Files already downloaded and verified
Data Loaded
 Epoch: 1 loss: 1.509
 Epoch: 2 loss: 1.085
 Epoch: 3 loss: 0.907
 Epoch: 4 loss: 0.794
 Epoch: 5 loss: 0.715
Accuracy for F3 is  73.95
```

Fig : Accuracy data

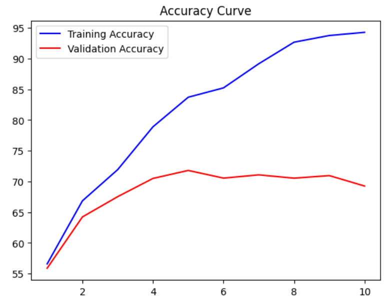
Part 3 : Improved the CNN Model

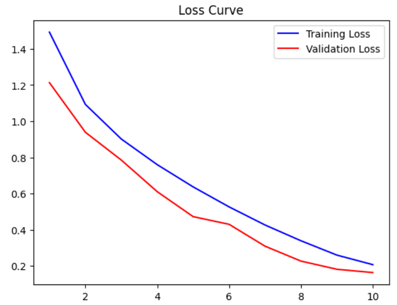
Improved model is

- CONV1: Kernel size(3x3), In channels 3, Out channels 32
- BN1 : In channel 32
- CONV2: Kernel size(3x3), In channels 32, Out channels 64
- BN2 : In channel 64
- CONV3: Kernel size(3x3), In channels 64, Out channels 128
- BN3 : In channel 128
- CONV4: Kernel size(3x3), In channels 128, Out channels 256

- BN4 : In channel 256
- POOL1 : Kernel Size (2x2) and stride = 2
- FC1 : fully connected layer with 512 output neurons.
- BN5 : In channel 512
- FC2 : fully connected layer with 10 output neurons.

Reasons :





This model includes several improvements over a simple CNN model. It uses batch normalisation after each convolutional layer, which helps to reduce overfitting and improve the speed of convergence during training.
Accuracy : 71.8 %