

1. Structure of the code.

largeMarginSoftmaxLoss :

Function compute the large margin softmax loss for the CNN. We have global weight vectors corresponding to each class and the x_i components. Find the angle θ between the weight vector and x_i . Then using the following formula compute the loss L_i .

$$L_i = -\log \left(\frac{e^{\|W_{y_i}\| \|x_i\| \cos(\theta_{y_i})}}{\sum_j e^{\|W_j\| \|x_i\| \cos(\theta_j)}} \right)$$

CNN layer structure:-

```
model.add(ZeroPadding2D((1,1),input_shape=x_train.shape[1:]))
model.add(Convolution2D(64, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(64, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(128, 3, 3, activation='relu'))
model.add(ZeroPadding2D((1,1)))
model.add(Convolution2D(128, 3, 3, activation='relu'))
model.add(MaxPooling2D((2,2), strides=(2,2)))

model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

We have used the pretrained model. We have applied two 64 filters of 3*3 and two 128 filters of 3*3. For the fully connected neural layers, We have used the two Dense layers of 4096, and the final one of 10(i.e no of classes). We have used 1*1 padding and 2*2 pooling for the CNN.

Stochastic gradient descent

We have used stochastic gradient as an optimizer for our model simulation.

```
sgd = SGD(lr=lr, momentum=0.9, decay=decay, nesterov=False)
```

DATASETS used

We have trained our model on MNIST and Cifar10 datasets.

2. Instructions for setting up and running the code.

Use python2 to run the code.

Install keras and all its dependencies.

3. list of methods

We have written the following LARGE MARGIN SOFTMAX loss function.

```
def largeMarginSoftmaxLoss(y_true, y_pred):  
    wt=wt.reshape(10,4096);  
    totalLoss=0;  
    predictedClass = y_pred  
    normw=normCalculation(w[predictedClass])  
    normx=normCalculation(x)  
    angleXW=angle(x,w[predictedClass])  
    angleXW = m*angleXW  
    xwi=normx*normw*math.cos(angleXW)  
    xwi=math.exp(xwi)  
    totalSum=0.0  
    for each in w:  
        normw=normCalculation(w[each])  
        angleXW=angle(x,w[each])*m  
        xwt=normx*normw*math.cos(angleXW)  
        totalSum=totalSum + math.exp(xwt)  
    totalLoss = totalLoss - math.log(xwi/totalSum)  
  
    return totalLoss
```

4. RESULTS

For CIPHAR10

Value of m	Error rate
1	9.10
2	7.75
3	7.67
4	7.60

```
on your machine and could speed up CPU computations.
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use SSE4.2 instructions, but these are
on your machine and could speed up CPU computations.
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use AVX instructions, but these are
your machine and could speed up CPU computations.
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use AVX2 instructions, but these are
your machine and could speed up CPU computations.
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use FMA instructions, but these are
your machine and could speed up CPU computations.
18720/35000 [=====>.....] - ETA: 993s - loss: 1.9765 - acc: 0.2676^[[C23;5~
35000/35000 [=====] - 2303s - loss: 1.7915 - acc: 0.3377 - val_loss: 1.4329 - val_acc: 0.4855
Epoch 2/10
35000/35000 [=====] - 2287s - loss: 1.3224 - acc: 0.5210 - val_loss: 1.1595 - val_acc: 0.5880
Epoch 3/10
35000/35000 [=====] - 2138s - loss: 1.1099 - acc: 0.6017 - val_loss: 1.0342 - val_acc: 0.6350
Epoch 4/10
35000/35000 [=====] - 2101s - loss: 0.9428 - acc: 0.6657 - val_loss: 1.0009 - val_acc: 0.6457
Epoch 5/10
35000/35000 [=====] - 2105s - loss: 0.8148 - acc: 0.7108 - val_loss: 0.8908 - val_acc: 0.6926
Epoch 6/10
35000/35000 [=====] - 2118s - loss: 0.6950 - acc: 0.7546 - val_loss: 0.8749 - val_acc: 0.6986
Epoch 7/10
35000/35000 [=====] - 2118s - loss: 0.5854 - acc: 0.7939 - val_loss: 0.8780 - val_acc: 0.7034
Epoch 8/10
35000/35000 [=====] - 2110s - loss: 0.4857 - acc: 0.8297 - val_loss: 0.8591 - val_acc: 0.7221
Epoch 9/10
35000/35000 [=====] - 2097s - loss: 0.3929 - acc: 0.8627 - val_loss: 0.8889 - val_acc: 0.7219
```