Assignment A04: Neural Networks

Thalagala B.P. 180631J

Data set: *CIFAR10 50,000 32×32×3 training images and 10 classes. Accuracy greater than 0.1 shows learning.*

Part 1

For the linear classifier in `cell 2`, the score function is $f(x) = Wx+b$. But Keeping track of two sets of parameters $\boldsymbol{W}$ and $\boldsymbol{b}$ separately is not really efficient in computation. This cumbersomeness can be eliminated by combining both of them into one single matrix as coded in `cell 1`. Additionally a column of ones must be added in front of training and testing images matrices(`x_train`,`x_test`) to enable matrix multiplication after this rearrangement.

```
[1]: std=1e-5
     w1 = std*np.random.randn(Din, K) # Initializing the weight matrix with random weights
     b1 = np.zeros(K) # Initializing the bias vector
     # Rearranging train and test samples: (ra=rearranged)
     x_train_ra = np.concatenate((np.ones((x_train.shape[0],1)),x_train), axis=1)
     x_test_ra  = np.concatenate((np.ones((x_test.shape[0],1)),x_test), axis=1)
     # Rearranging weight matrix and bias matrix into single matrix
     w1 = np.concatenate((b1.reshape(1,K), w1), axis=0)
```

After above described rearrangements of the matrices, gradient descent can be implemented as follows in `cell 2`. Mean sum of squared errors function is used as the loss function and regularization is also used to eliminate unnecessary growth of parameters. And it was divided by two to make the rest of the coding easy.

$$\therefore\ Loss\ Function = \frac{1}{2m}.\sum\left[(hypothesis - y_{train})^2 + \lambda W^2\right]\ where\ \lambda = regularization\ parameter$$

```
[2]: m = x_train.shape[0]  # Number of training examples
     for t in range(1,iterations+1):
         # Forward Propagation
         hypothesis = x_train_ra.dot(w1)
         loss = (1/(2*m))*np.sum(( hypothesis - y_train)**2) + (1/(2*m))*reg*np.sum(w1**2)
         # Backward Propagation
         dw1 = (1/m)*(x_train_ra.T.dot(hypothesis - y_train))  + (1/m)*reg*w1
         w1 = w1 - lr*dw1
         # Decaying the learning rate
         lr = lr*lr_decay
```

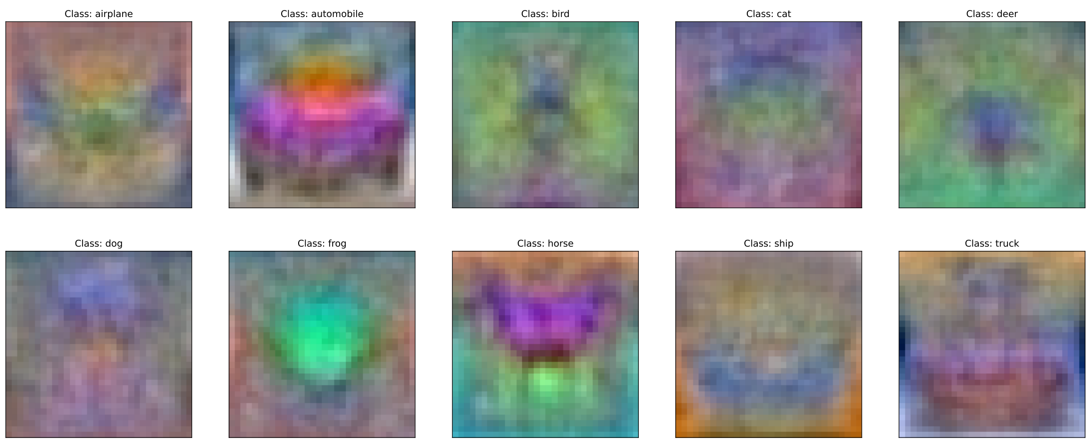

Figure 1: Weights matrix `w1` as 10 images

The learned weights at the end of learning for CIFAR-10 (after running the gradient descent 300 epochs), can be visualized as 10 figures in Fig. 1. Observe that, the weight matrix of the "horse" class slightly looks like a horse with two heads. This property is also visible in the “automobile” class too. Therefore theses templates will give maximum score for images of horses and automobiles respectively when the inner product is considered between the template and the image.

Hyperparameters of the model, training and testing loss and accuracies corresponding to the first and last epochs are given below. Normalized losses and the accuracies are plot in the Fig. 2.

```
Initial Learning Rate = 1.4e-2, Learning Rate Decay = 0.999, Regularization Parameter = 5e-6
Optimizer = Vanilla Gradient Descent(VGD), Epochs = 300

| Epoch 001 | Loss 0.5000 | accuracy: 0.0834 | val_loss: 0.4846 | val_accuracy: 0.2486 |
| Epoch 300 | Loss 0.3946 | accuracy: 0.4104 | val_loss: 0.3958 | val_accuracy: 0.3981 |
```

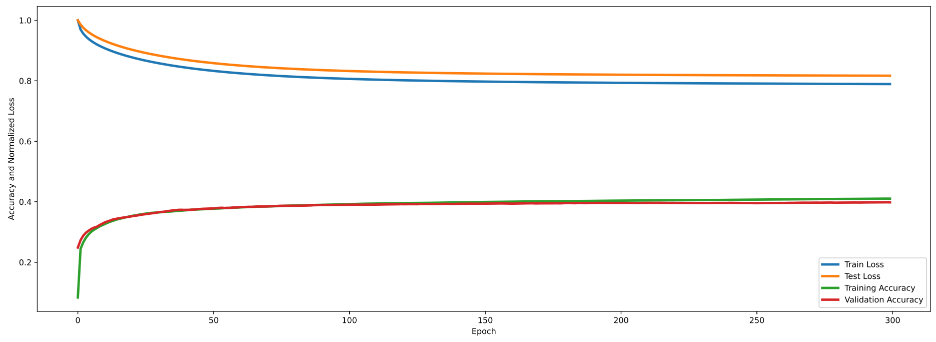

Figure 2: Training Loss, Training Accuracy, Validation Loss and Validation Accuracy of the Linear Classifier with each iteration for 300 epochs

When observing the above figure following conclusions can be made. As the number of epochs increase, training and validation accuracies monotonically increase. But gap between the two curves also increases due to over-fitting of the model. Because parameters become more and more specific to the training set rather than become general to never seen images. The situation is the same for losses.

Part 2

As described in the part 1 weight matrices and bias vectors are concatenated into two matrices named `w1` and `w2`. The two layers are fully connected and the `sigmoid function` is used as the activation function for the hidden layer which consists of 200 hidden nodes. Layer 2, the output layer has no activation function.

[3]:
```python
H = 200 # No of hidden nodes
std=1e-5
# Hidden Layer
w1 = std*np.random.randn(Din, H) # Initializing the weight matrix with random weights
b1 = np.zeros(H) # Initializing the bias vector
# Last Layer
w2 = std*np.random.randn(H, K) # Initializing the weight matrix with random weights
b2 = np.zeros(K) # Initializing the bias vector
# Rearranging train and test samples: (ra=rearranged)
x_train_ra = np.concatenate((np.ones((x_train.shape[0],1)),x_train), axis=1)
x_test_ra  = np.concatenate((np.ones((x_test.shape[0],1)),x_test), axis=1)
# Rearranging weight matrices and bias vectors into single matrices
w1 = np.concatenate((b1.reshape(1,H), w1), axis=0)
w2 = np.concatenate((b2.reshape(1,K), w2), axis=0)
```

Normalization of image pixel values was removed from the data pre-processing stage as it was observed that model stops learning after few iterations. Because extremely small weights in the matrices, keeps the weight matrices almost the same and the change in loss function becomes negligible.

The loss function used in the part 1 is used here as well, with the additional term related to the `w2` matrix.

$$\therefore\ Loss\ Function = \frac{1}{2m}.\sum\left[(hypothesis - y_{train})^2 + \lambda W_1^2 + \lambda W_2^2\right]\ where\ \lambda = regularization\ parameter$$

[4]:
```python
for t in range(1,iterations+1):
    # Forward Propagation
    hypo = sigmoid(x_train_ra.dot(w1)) # Layer 1 with sigmoid activation
    hypothesis = np.concatenate((np.ones((hypo.shape[0],1)),hypo), axis=1) # Rearranging for layer 2
    predict = hypothesis.dot(w2) # Layer 2
    loss = (1/(2*m))*np.sum(( predict - y_train)**2)\
        + (1/(2*m))*reg*np.sum(w1**2) + (1/(2*m))*reg*np.sum(w2**2)
    # Back Propagation: partial dertivatives of Loss function
    dpredict =  (1/m)*(predict - y_train)
    dw2 = hypothesis.T.dot(dpredict) + (1/m)*reg*w2
    dh = dpredict.dot(w2[1:,].T) # Removing bias vector w2(201 X 10)--> 200 X 10
    dhdxw1 = hypo*(1 - hypo) #using the hypothesis(50000X200), the one before rearranging.
    dw1 = x_train_ra.T.dot(dh*dhdxw1) + (1/m)*reg*w1
    # Gradient Descent
    w1 = w1 - lr*dw1
    w2 = w2 - lr*dw2
     # Decaying learning rate
    lr = lr*lr_decay
```

```
Initial Learning Rate = 1.4e-2, Learning Rate Decay = 0.999, Regularization Parameter = 5e-6
Optimizer = Vanilla Gradient Descent(VGD), Epochs = 300

| Epoch 001 | Loss 0.5000 | accuracy: 0.1000 | val_loss: 0.4541 | val_accuracy: 0.1339 |
| Epoch 300 | Loss 0.3765 | accuracy: 0.4436 | val_loss: 0.3813 | val_accuracy: 0.4275 |
```

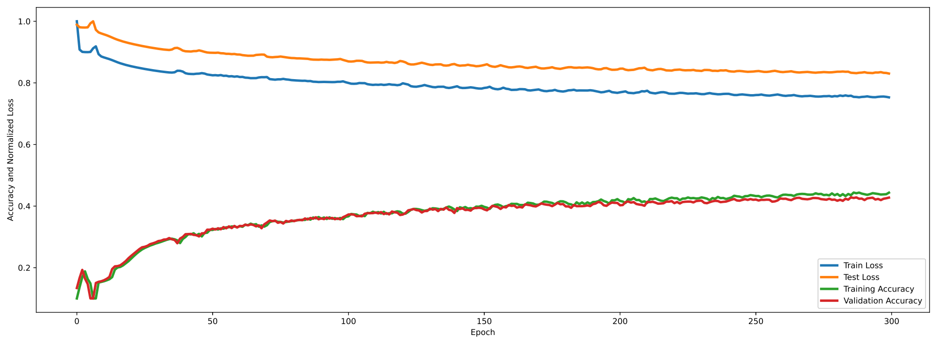

Figure 3: Training Loss, Training Accuracy, Validation Loss and Validation Accuracy of the 2 Layer Classifier with each iteration: for 300 epochs

As described in the part 1 the same reason applies for the behavior of the gaps between normalized losses and accuracies curves. In addition to that due to the non linearity introduced by the `sigmoid function` at the hidden nodes, curves are not smooth as they were in the part 1 linear classifier.

Part 3

For the purpose of comparison hyperparameters were kept the same as in part 2. But the optimizer changes form Vanilla Gradient Descent to Mini-batch Gradient Descent(mentioned as stochastic GD-SGD).

```
Initial Learning Rate = 1.4e-2, Learning Rate Decay = 0.999, Regularization Parameter = 5e-6
Optimizer = Mini Batch Gradient Descent, Batch Size = 500, Epochs = 300
```

```
| Epoch 001 | Loss 0.4768 | accuracy: 0.1249 | val_loss: 0.4619 | val_accuracy: 0.1252 |
| Epoch 300 | Loss 0.3686 | accuracy: 0.4674 | val_loss: 0.3768 | val_accuracy: 0.4432 |
```

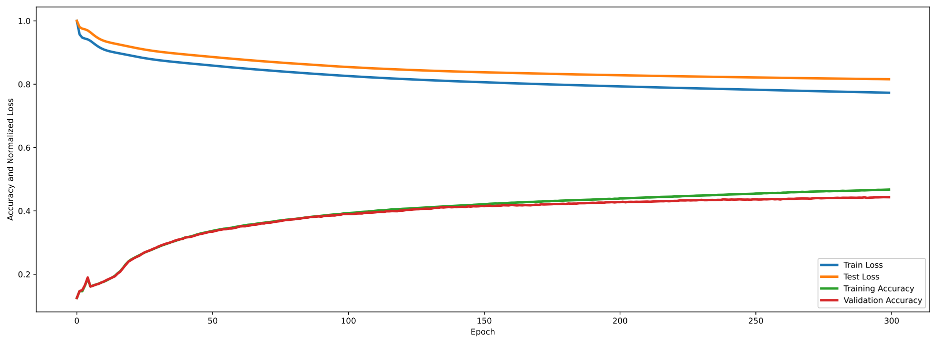

Figure 4: Training Loss, Training Accuracy, Validation Loss and Validation Accuracy of the 2 Layer Classifier with each iteration for 300 epochs: Using Mini Batch Gradient Descent as the optimizer

consider the accuracies and losses related to the last epochs of the Part 2 and Part 3.

```
| VGD | Epoch 300 | Loss 0.3765 | accuracy: 0.4436 | val_loss: 0.3813 | val_accuracy: 0.4275 |
| SGD | Epoch 300 | Loss 0.3686 | accuracy: 0.4674 | val_loss: 0.3768 | val_accuracy: 0.4432 |
```

In Part 2 **the same batch**(50,000) training samples were used ***for every step of Gradient Descent*** and therefore the parameters become more specific to the training set as there is no any randomness in the data processed in each iteration. But with the Mini batch GD optimizer it is not the case. At each epoch the training set(50,000) is ***shuffled*** and ***divided into*** number($n = \frac{Number\ of\ Training\ samples}{Batch\ size}$) of ***mini batches***. After that gradient descent is carried out on each such mini batch and weights are updated accordingly. The advantage of this process is, since the batch is shuffled and only a small set of the batch is seen by the model at one gradient descent step, it performs well on the never seen test samples better than the algorithm in part 2.

[5]:
```python
batch_size = 500 # define the batch size
seed = 0; rng = np.random.default_rng(seed=seed)
for t in range(1,iterations+1):
    indices = np.arange(Ntr) #Number of training samples
    rng.shuffle(indices)
    x_train_3 = x_train_ra[indices]
    y_train_3 = y_train[indices]
    batch_loss = 0 # Loss for each batch
    for start in range(0,Ntr,batch_size):
        stop = start + batch_size
        # Forward Propagation
        hypo = sigmoid(x_train_3[start:stop].dot(w1)) # Layer 1 with sigmoid activation
        hypothesis = np.concatenate((np.ones((hypo.shape[0],1)),hypo), axis=1)
        predict = hypothesis.dot(w2) # Layer 2
        minibatch_loss = (1/(2*m))*np.sum(( predict - y_train_3[start:stop])**2)\
            + (1/(2*m))*reg*np.sum(w1**2) + (1/(2*m))*reg*np.sum(w2**2)
        batch_loss+= minibatch_loss
        # Back Propagation partial dertivatives of Loss function
        dpredict =  (1/m)*(predict - y_train_3[start:stop])
        dw2 = hypothesis.T.dot(dpredict) + (1/m)*reg*w2
        dh = dpredict.dot(w2[1:,].T) # Removing bias vector w2(201x10)--> 200x10
        dhdxw1 = hypo*(1 - hypo) #using hypothesis 50000*200, the one before rearranging.
        dw1 = x_train_3[start:stop].T.dot(dh*dhdxw1) + (1/m)*reg*w1
        w1 = w1 - lr*dw1  # Gradient Descent
        w2 = w2 - lr*dw2  # Gradient Descent
    lr = lr*lr_decay      # Decaying the learning rate
```

Part 4

The CNN model declared below starts over-fitting around the 8th epoch. Therefore the number of epochs was limited to 10. This fact is clearly visible, because at the 10th epoch even though the training loss and accuracy are better than that of the 8th epoch, validation loss and validation accuracy are worse. However, the model has performed absolutely better than any of the previous models in 8 epochs with a validation accuracy of 69.79%. Previously we were only able to reach up to 44.32% of validation accuracy with the defined hyperparameters.

```
Initial Learning Rate = 1.4e-2,  Learnable Parameters: 73,418
Optimizer = SGD with Momentum = 0.9, Batch Size = 50, Epochs = 10

| Epoch 001 | Loss: 1.8674 | accuracy: 0.3070 | val_loss: 1.2984 | val_accuracy: 0.5298 |
| Epoch 008 | Loss: 0.6375 | accuracy: 0.7746 | val_loss: 0.8601 | val_accuracy: 0.7088 |
| Epoch 010 | Loss: 0.5651 | accuracy: 0.8021 | val_loss: 0.9394 | val_accuracy: 0.6979 |
```

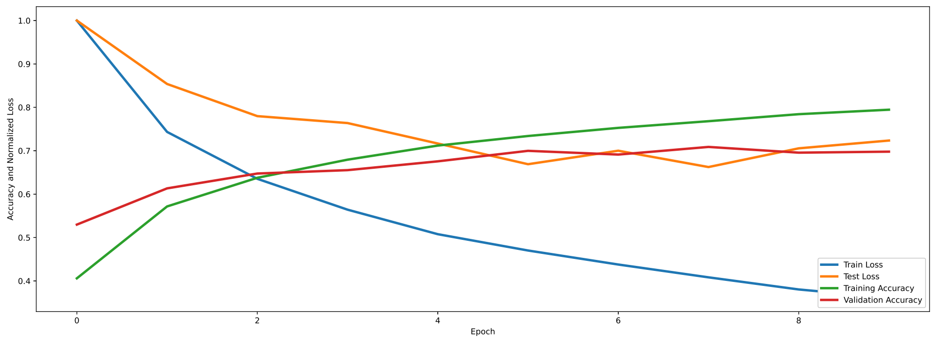

Figure 5: Training Loss, Training Accuracy, Validation Loss and Validation Accuracy of the CNN model

[6]:
```python
(x_train, y_train), (x_test, y_test) = datasets.cifar10.load_data()
K = len(np.unique(y_train)) # Number of Classes
# Normalize pixel values: Image data preprocessing
x_train, x_test = x_train / 255.0, x_test / 255.0
mean_image = np.mean(x_train, axis=0) # axis=0: mean of a column; Mean of each pixel
x_train = x_train - mean_image
x_test = x_test - mean_image
# Convert class vectors to binary class matrices.
y_train = tf.keras.utils.to_categorical(y_train, num_classes=K)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=K)
model = models.Sequential() # Declaring the CNN
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3), name='C32'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu', name='C64_1')) # 64, 3x3 convolutions
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu', name='C64_2')) # 64, 3x3 convolutions
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten()) # Make the (None, 2, 2, 64) tensor flat
model.add(layers.Dense(64, activation='relu', name='F64')) # Dense Layer 1
model.add(layers.Dense(10, name='F10')) # Because CIFAR has 10 output classes
model.summary() # Complete architecture of the model
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=1.4e-2, momentum=0.9),
              loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
history = model.fit(x_train, y_train,batch_size=50, epochs=10,
                    validation_data=(x_test, y_test))
```

– ***Executable code for this assignment can be found on*** –