Assignment A04: Neural Networks

Thalagala B.P. 180631J

1 Part 1

For our linear classifier, the score function is $f(x) = Wx + b$. But Keep track of two sets of parameters $\boldsymbol{w}$ and $\boldsymbol{b}$ separately is not really efficient. This cumbersomeness can be eliminated by combining both of them into one single matrix as coded in `cell 1`. Additionally column of ones must be added in front of train images matrix to enable matrix multiplication.

[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)
```

[4]:
```
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
    # Training Accuracy and Validation Accuracy
    train_acc = getAccuracy(hypothesis, y_train)
    valid_acc = getAccuracy(x_test_ra.dot(w1), y_test)
    # Decaying learning rate
    lr_hitory.append(lr)
    lr = lr*lr_decay
```

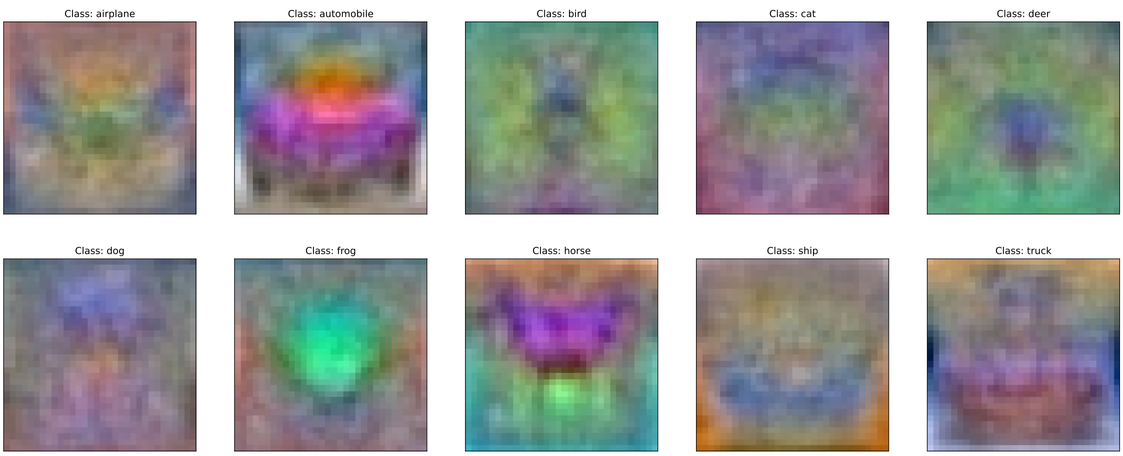

Figure 1: Weights matrix W1 as 10 images

[5]:
```
H = 200 # No of hidden nodes
std=1e-5
# Hidden Layer
```

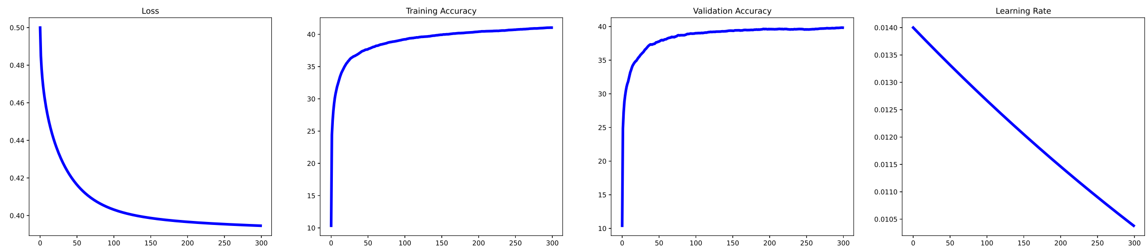

Figure 2: Loss, Training Accuracy, Validation Accuracy and Learning Rate of the Linear Classifier with each iteration: for 300 epochs

```
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)
```

[6]:
```
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(201x10)--> 200x10
    dhdxw1 = hypo*(1 - hypo) #using hypothesis 50000*200 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_hitory.append(lr)
    lr = lr*lr_decay
```

[7]:
```
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) # Rearranging␣
↪for layer 2
        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
        # Gradient Descent
        w1 = w1 - lr*dw1
        w2 = w2 - lr*dw2
    # Decaying learning rate
    lr_hitory.append(lr)
    lr = lr*lr_decay
```

[8]:
```
(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)
# Declaring the CNN
model = models.Sequential()
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))
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
```