1 Linear Classifier

1.1 Implementing gradient descent

1.1.1 Score function and its derivative

The mean sum of squared errors function is taken as the loss function.

$$L = \frac{1}{N} \sum_{images} [scores - class]^2 + \lambda \sum_{elements} w^2 \tag{1}$$

```
def loss_function(x_tr,y_tr,reg,w1,type):
    '''type =0:training, type =1: validation'''
    scores = w1.dot(x_tr)            # scores is a 10x50000 array
    loss = (1./50000)*np.square(scores-y_tr).sum()  + reg*np.sum([np.sum(w*w)for w in
    w1])
    #CONDITIONAL APPENDING OF SCORES AND LOSSES TO RELEVENT LISTS
    return scores
```

The derivative of the above function used for gradient descent.

$$\nabla L = \frac{1}{N} \times 2[scores - class] + 2 \times W \tag{2}$$

```
def grad_analytical(scores,x_tr,y_tr,reg,w_mat):
    grad = (1./50000)*(2.0)*((scores-y_tr).dot(x_tr.T)) + 2*reg*w_mat
    return grad
```

The analytical closed form of the derivative is used, the numerical method took a large amount of time to compute. Can be seen in the history. Vectored implementation in the code, scores, class, W are numpy vectors.

1.1.2 Gradient descent

```
while iteration < epochs:
    loss_function(x_train,y_train,reg,w_0)
    val_scores = w_0.dot(x_test)
    val_acc_history.append(accuracy(val_scores, y_test.T))
    w_grad = grad_analytical(scores,x_train,y_train,reg,w_0)
    w_0 += -lr*w_grad    #descent
    lr *= lr_decay
```

Each iteration the scores are calculated and sent to the `grad_analytical()` function and the returned values are used to update the W(lines 6-7).

1.2 Weights matrix W

1.2.1 Code snippet

```
w_0_copy = w_0
w_0_copy = np.delete(w_0_copy,3072,axis=1)
img=[[],[],[],[],[],[],[],[],[],[]]
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', '
    truck']
for i in range(w_0.shape[0]):
    img_i = w_0_copy[i,:]
    img[i]= (255.0*(img_i-min(img_i))/(max(img_i)-min(img_i))).reshape(32,32,3)     #
    remap to image values so can be dispalyed
    img[i] = img[i].astype("uint8")         #data type conversion
```

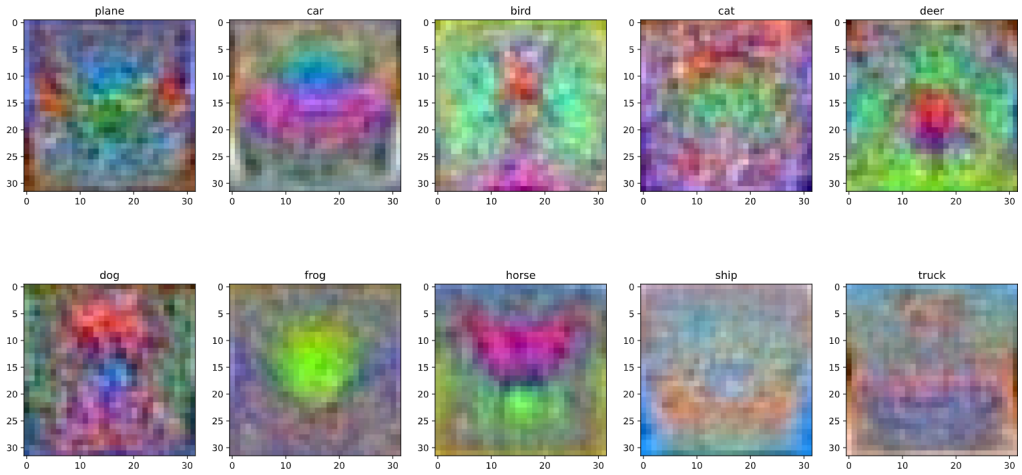

Figure 1: Weights matrix W as 10 images

1.3 Training and testing loss and accuracy

$$\text{Initial Learning rate} = 0.017$$
$$\text{Learning rate decay} = 0.99995$$
$$\text{Regularization parameter} \lambda = 1 \times 10^{-10}$$

```
def accuracy(scores,ytrain):
    predclass = np.argmax(scores,axis=0)
    trueclass = np.argmax(ytrain,axis=0)
    return ( np.sum(predclass==trueclass)/trueclass.size )
```

Calculating accuracy

FINAL VALUES: Training : 0.880277 Validation : 0.177282

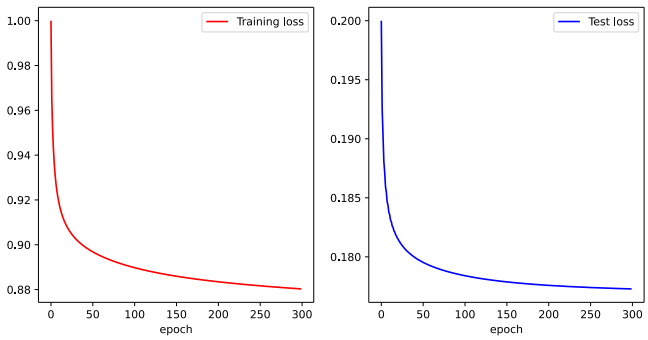

Figure 2: Linear classifier losses

FINAL VALUES: Training : 0.42356 Validation : 0.4074

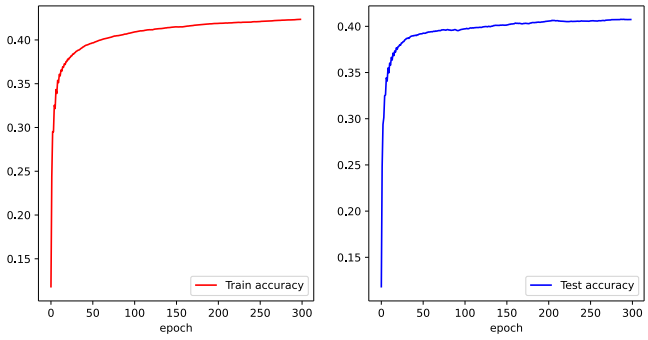

Figure 3: Linear classifier accuracy

2 Two Layer fully connected network with H=200 hidden nodes

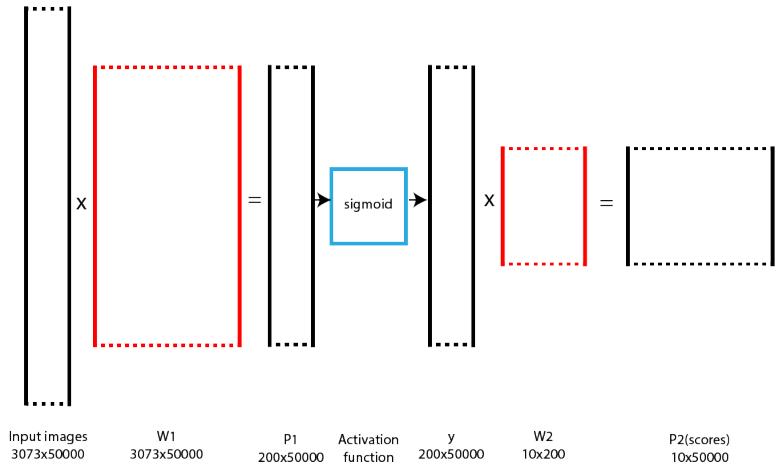

Figure 4: Structure of the 2 layer neural net with 200 hidden nodes

2.1 Implementing gradient descent

The loss function gets a slight modification to include the effect of the 2^{nd} layer.

$$L = \frac{1}{N}\sum_{images}[scores - labels]^2 + \lambda\left(\sum_{elements} w_1^2 + \sum_{elements} w_1^2\right) \tag{3}$$

The gradient is found back-propagating with chain rule.

$$\frac{\partial L}{\partial W2} = \frac{\partial L}{\partial P2}\frac{\partial P2}{\partial W2}$$

$$\frac{\partial L}{\partial P2} = \frac{1}{50000} \times 2 \times (P2 - labels) \qquad \frac{\partial P2}{\partial W2} = y$$

$$\frac{\partial L}{\partial W2} = \frac{1}{50000} \times 2 \times (P2 - labels) \times y \tag{4}$$

$$\frac{\partial L}{\partial W1} = \frac{\partial L}{\partial P2}\frac{\partial P2}{\partial y}\frac{\partial y}{\partial P1}\frac{\partial P1}{\partial W1}$$

$$\frac{\partial P2}{\partial y} = W2 \qquad \frac{\partial y}{\partial P1} = y(1-y) \qquad \frac{\partial P1}{\partial W1} = X(Input)$$

$$\frac{\partial L}{\partial W1} = \frac{1}{50000} \times 2 \times (P2 - labels) \times W2 \times y(1-y) \times X \tag{5}$$

The derivative of regularization is added separately.

```
def sigmoid(alpha):
    return 1/(1+np.exp(-alpha))

while epoch < epochs:
    indices = np.arange(Ntr)    #shuffling images each epoch
    rng.shuffle(indices)
    X = x_train[:,indices]
    Y_tr = y_train[:,indices]
    #FORWARD PASS##
    P1 = w_1.dot(X)    #P1 200X50000
    y = sigmoid(P1)    #ys
    P2 = w_2.dot(y)    #P2, in this case outputs,no activation 10x50000
    #explicit implementation of loss
    L = (1./50000)*((np.square(P2-Y_tr)).sum() + reg*(np.sum(w_1**2)+np.sum(w_2**2)))
    #explicit calculation of the two gradients
    #PARTIAL DERIVATIVE WRT W2
```

```
    dL_dP2 = (1./50000)*2.0*(P2-Y_tr)
    dP2_dw2 = y.T
    dL_dw2 = dL_dP2.dot(dP2_dw2) + reg*w_2  #pfffttt
    #PARTIAL DERIVATIVE WRT W1
    dP2_dy = w_2
    dy_dP1 = y*(1-y)
    dP1_dw1= X
    dL_dw1 = np.multiply(((dP2_dy.T).dot(dL_dP2)),dy_dP1).dot(dP1_dw1.T)  +reg*w_1
    #BACKWARD PASS##
    w_1 += -lr*dL_dw1
    w_2 += -lr*dL_dw2
    lr *= lr_decay
    epoch+=1
```

2.2 Training and testing loss and accuracy

Initial Learning rate = **0.025**, Learning rate decay = **0.975**, Regularization parameter $\lambda = 5 \times 10^{-5}$
FINAL VALUES LOSSES: Training : 0.871400 Test : 0.871593
FINAL VALUES ACCURACY: Training : 0.2521 Test : 0.26

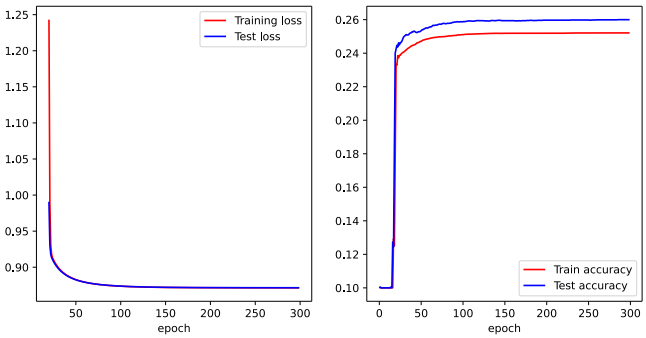

Figure 5: Neural net loss and accuracy

3 Stochastic Gradient descent

Batch size is 500. $lr = 0.02$, lr_decay $= 0.999$, $\lambda = 5 \times 10^{-5}$

3.1 Training and testing loss and accuracy

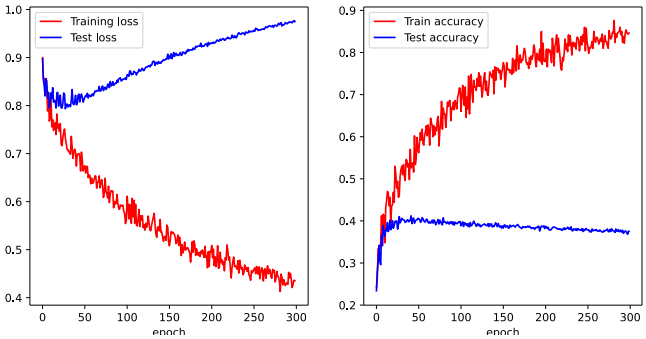

Figure 6: Stochastic gradient descent losses accuracy

FINAL VALUES LOSSES: Training : 0.434285 Test : 0.976056
FINAL VALUES ACCURACY: Training : 0.848 Test : 0.3732

```
while epoch < epochs:
    while iteration <(Ntr/batchsize):
        indices = np.random.choice(Ntr,batchsize,replace=False) # random indices
    batchsize
        X = x_train[:,indices]
        Y_tr = y_train[:,indices]
        #FORWARD PASS
```

```
        #BACKWARD PASS
        iteration +=1
```

3.2 Comparison with item 2

Stochastic gradient descent shows rough curves with spikes, while the vanilla gradient descent shows smooth curves in both the loss and accuracy plots. The initial spikes in the accuracy of the vanilla gradient descent is due to the loss function overcoming a local extrema. The spikes are caused by the stochastic behaviour and it is evident that it has reached a higher accuracy faster. The vanilla gradient descent has only been able to achieve around 0.26 test accuracy at 300 epochs, stochastic gradient descent has been able to achieve that at around 20 epochs while achieving a 0.848 training accuracy and 0.3732 test accuracy. The increase in test set accuracy is more significant despite the amount.

4 Convolutional Neural Network using Tensorflow

```
(x_train, y_train),(x_test,y_test) = cifar10.load_data()
x_train = x_train.astype("float32")/255.0   # normalize
x_test = x_test.astype("float32")/255.0     # normalize
model = keras.Sequential([    keras.Input(shape=(32, 32, 3)),
        layers.Conv2D(32, 3, padding="valid", activation="relu"),
        layers.MaxPooling2D((2,2)),
        layers.Conv2D(64, 3, activation="relu"),
        layers.MaxPooling2D((2,2)),
        layers.Conv2D(64, 3, activation="relu"),
        layers.MaxPooling2D((2,2)),
        layers.Flatten(),
        layers.Dense(64, activation="relu"),
        layers.Dense(10),])
print(model.summary())
model.compile(
    loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    optimizer=keras.optimizers.SGD(lr=0.02,momentum=0.001, name='SGD'),
    metrics=["accuracy"],)
history = model.fit(x_train, y_train, batch_size=50, epochs=100, verbose=2,
    validation_data=(x_test,y_test))
```

[language = Python]

4.1 Parameters

Trainable params: 73,418
Learning rate = 0.02
Momentum =-0.001
Results after 100 epochs.
FINAL VALUES LOSS: Training : 0.8579 Validation : 1.0124
FINAL VALUES ACCURACY: Training : 0.7027 Validation : 0.6548

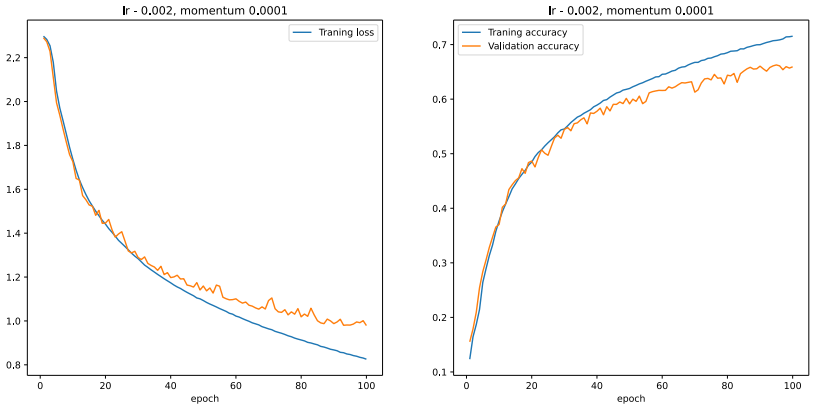

Figure 7: CNN loss and accuracy

REFERENCES
MIT OCW 6.034, Stanford University School of Engineering C231n.