

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.

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

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

The derivative of the above function used for gradient descent.

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

```
1 def grad_analytical(scores,x_tr,y_tr,reg,w_mat):
2     grad = (1./50000)*(2.0)*((scores-y_tr).dot(x_tr.T)) + 2*reg*w_mat
3     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. Vectorized implementation in the code, scores, class, W are numpy vectors.

1.1.2 Gradient descent

```
1 while iteration < epochs:
2     loss_function(x_train,y_train,reg,w_0)
3     val_scores = w_0.dot(x_test)
4     val_acc_history.append(accuracy(val_scores, y_test.T))
5     w_grad = grad_analytical(scores,x_train,y_train,reg,w_0)
6     w_0 += -lr*w_grad #descent
7     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

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

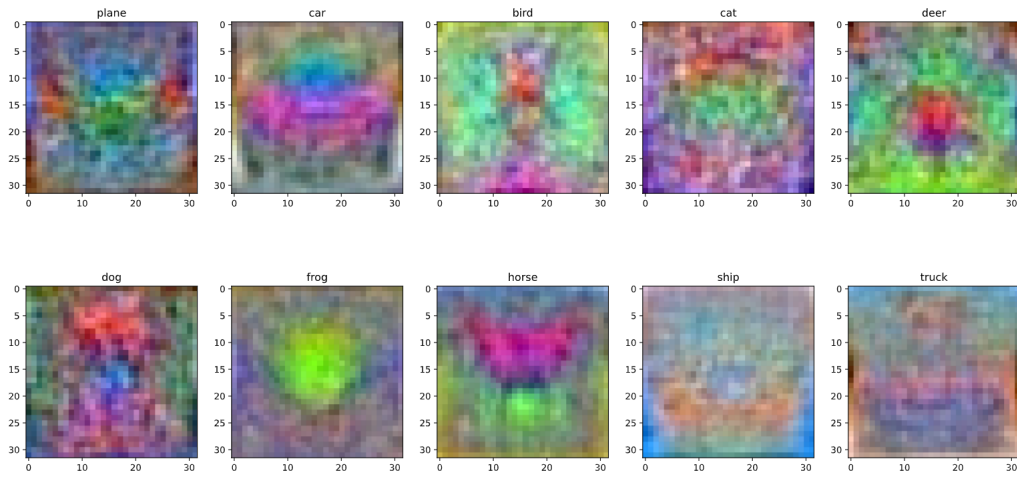


Figure 1: Weights matrix W as 10 images

1.3 Training and testing loss and accuracy

Initial Learning rate = 0.017

Learning rate decay = 0.99995

Regularization parameter $\lambda = 1 \times 10^{-10}$

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

Calculating accuracy

FINAL VALUES: Training : 0.880277 Validation : 0.177282

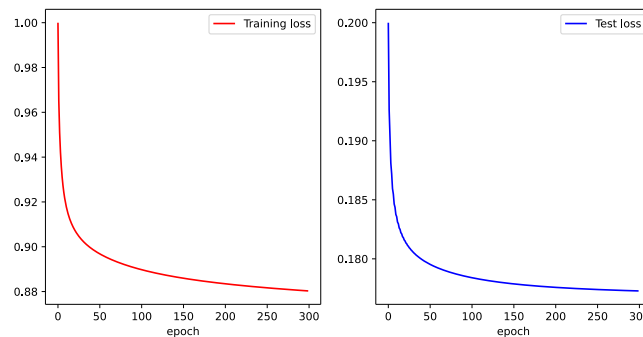


Figure 2: Linear classifier losses

FINAL VALUES: Training : 0.42356 Validation : 0.4074

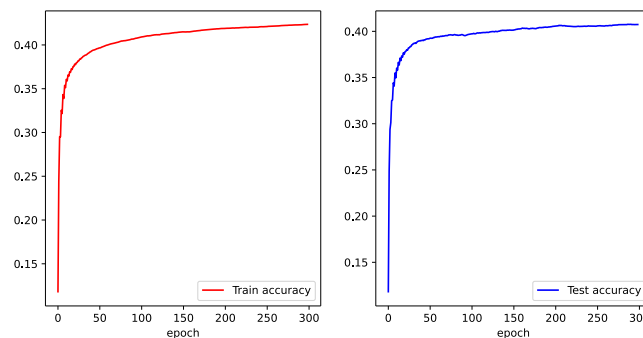


Figure 3: Linear classifier accuracy

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

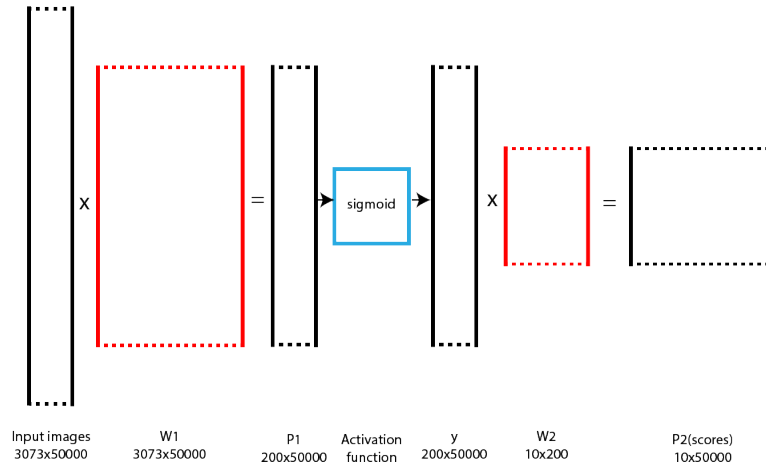


Figure 4: Structure of the 2 layer neural net with 200 hidden nodes

2.1 Implementing gradient descent

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

$$L = \frac{1}{N} \sum_{images} [scores - labels]^2 + \lambda \left(\sum_{elements} w_1^2 + \sum_{elements} w_2^2 \right) \quad (3)$$

The gradient is found back-propagating with chain rule.

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

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

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

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

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

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

The derivative of regularization is added separately.

```

1 def sigmoid(alpha):
2     return 1/(1+np.exp(-alpha))
3
4 while epoch < epochs:
5     indices = np.arange(Ntr)      #shuffling images each epoch
6     rng.shuffle(indices)
7     X = x_train[:,indices]
8     Y_tr = y_train[:,indices]
9     #FORWARD PASS##
10    P1 = w_1.dot(X)                #P1 200X50000
11    y = sigmoid(P1)                #ys
12    P2 = w_2.dot(y)                #P2, in this case outputs,no activation 10x50000
13    #explicit implementation of loss
14    L = (1./50000)*((np.square(P2-Y_tr)).sum() + reg*(np.sum(w_1**2)+np.sum(w_2**2)))
15    #explicit calculation of the two gradients
16    #PARTIAL DERIVATIVE WRT W2

```

```

17 dL_dP2 = (1./50000)*2.0*(P2-Y_tr)
18 dP2_dw2 = y.T
19 dL_dw2 = dL_dP2.dot(dP2_dw2) + reg*w_2 #pfffttt
20 #PARTIAL DERIVATIVE WRT W1
21 dP2_dy = w_2
22 dy_dP1 = y*(1-y)
23 dP1_dw1= X
24 dL_dw1 = np.multiply(((dP2_dy.T).dot(dL_dP2)),dy_dP1).dot(dP1_dw1.T) +reg*w_1
25 #BACKWARD PASS##
26 w_1 += -lr*dL_dw1
27 w_2 += -lr*dL_dw2
28 lr *= lr_decay
29 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

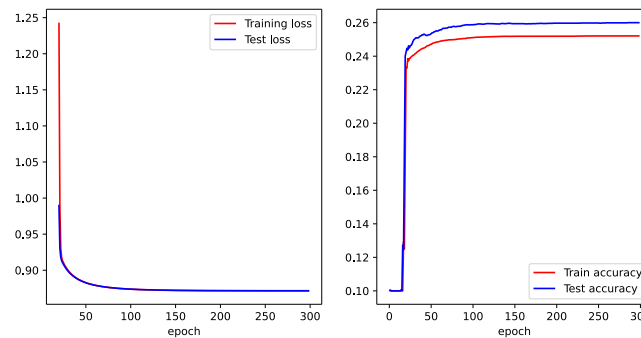


Figure 5: Neural net loss and accuracy

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

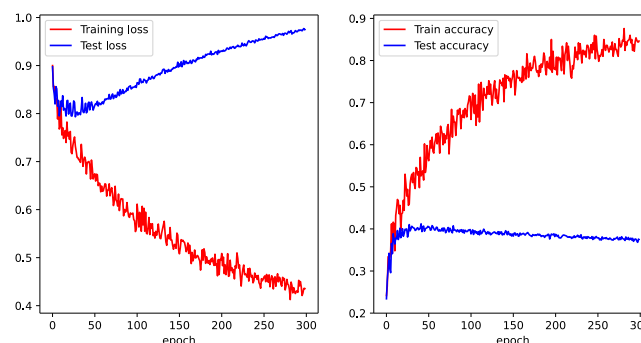


Figure 6: Stochastic gradient descent losses accuracy

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

```

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

```

```

7      #BACKWARD PASS
8      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

```

1 (x_train, y_train), (x_test, y_test) = cifar10.load_data()
2 x_train = x_train.astype("float32")/255.0 # normalize
3 x_test = x_test.astype("float32")/255.0 # normalize
4 model = keras.Sequential([
5     keras.Input(shape=(32, 32, 3)),
6     layers.Conv2D(32, 3, padding="valid", activation="relu"),
7     layers.MaxPooling2D((2,2)),
8     layers.Conv2D(64, 3, activation="relu"),
9     layers.MaxPooling2D((2,2)),
10    layers.Conv2D(64, 3, activation="relu"),
11    layers.MaxPooling2D((2,2)),
12    layers.Flatten(),
13    layers.Dense(64, activation="relu"),
14    layers.Dense(10),])
15 print(model.summary())
16 model.compile(
17     loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
18     optimizer=keras.optimizers.SGD(lr=0.02, momentum=0.001, name='SGD'),
19     metrics=["accuracy"],)
20 history = model.fit(x_train, y_train, batch_size=50, epochs=100, verbose=2,
21                     validation_data=(x_test, y_test))

```

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

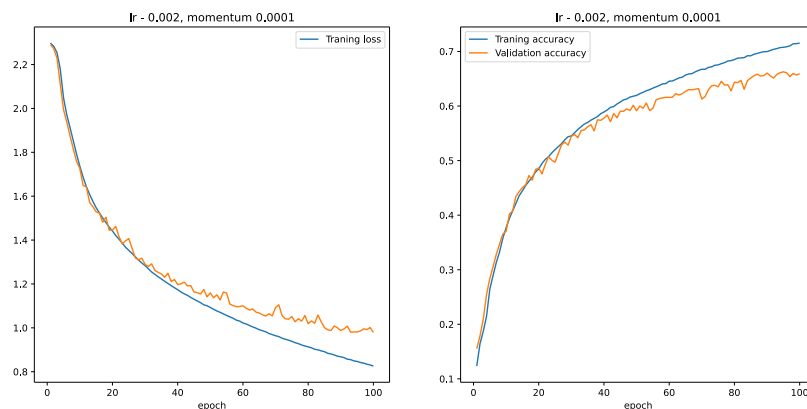


Figure 7: CNN loss and accuracy

REFERENCES

MIT OCW 6.034, Stanford University School of Engineering C231n.