EN2550 2020: Assignment 04: 180437U

A function is predefined to pre-process input data: preproc(norm,reshape). “norm” and “reshape” are Boolean inputs. Pixel value normalization and reshaping the input data can be done by setting them to True.

1. Linear Classifier

Loss function and the accuracy function is predefined to reduce the complexity in coding. Mean sum of squared errors with regularization is used as the loss function. Normalized difference between the true label and the predicted label with the highest probability is used to calculate the accuracy.

# Defining Regularized loss function

def regloss(y\_pred,y,w1,w2=0):

    batch\_size=y\_pred.shape[0] #determining the number of input data

    loss=(1/(batch\_size))\*(np.square(y-y\_pred)).sum()+reg\*(np.sum(w1\*w1)+np.sum(w2\*w2))

    return loss

# Defining accuracy function

def accuracy(y\_pred,y):

    batch\_size=y\_pred.shape[0] #determining the number of input data

    K=y\_pred.shape[1] # determining number of classes

    acc=1-(1/(batch\_size\*K))\*(np.abs(np.argmax(y,axis=1)-np.argmax(y\_pred,axis=1))).sum()

    return acc

Defining the linear classifier function.

# Defining linear Classifier function

def linclas(x\_train,y\_train,x\_test,y\_test,K,Din,lr,lr\_decay,reg):

    Ntr = x\_train.shape[0] # Number of training data

    Nte = x\_test.shape[0] # Number of testing data

    loss\_history = []

    loss\_history\_test = []

    train\_acc\_history = []

    val\_acc\_history = []

    seed = 0

    rng = np.random.default\_rng(seed=seed)

    # Initializing weight and bias arrays

    Din=x\_train.shape[1]

    std=1e-5

    w1 = std\*np.random.randn(Din, K)

    b1 = np.zeros(K)

    for t in range(iterations):

        # shuffling the training data set to randomize the training process.To prevent overfitting

        indices = np.arange(Ntr)

        rng.shuffle(indices)

        x=x\_train[indices]

        y=y\_train[indices]

        # forward pass

        y\_pred=x.dot(w1)+b1

        y\_pred\_test=x\_test.dot(w1)+b1

        # calculating loss

        train\_loss=regloss(y\_pred,y,w1)

        test\_loss=regloss(y\_pred\_test,y\_test,w1)

        loss\_history.append(train\_loss)

        loss\_history\_test.append(test\_loss)

        # calculating accuracy

        train\_acc=accuracy(y\_pred,y)

        train\_acc\_history.append(train\_acc)

        test\_acc=accuracy(y\_pred\_test,y\_test)

        val\_acc\_history.append(test\_acc)

        if t%10 == 0:

            print('epoch %d/%d: loss= %f-- ,test loss= %f--,train accracy= %f--, test accracy= %f' % (t,iterations,train\_loss,test\_loss,train\_acc,test\_acc))

        # Backward pass

        dy\_pred=(1./batch\_size)\*2.0\*(y\_pred-y)

        dw1=x.T.dot(dy\_pred)+reg\*w1

        db1=dy\_pred.sum(axis=0)

        # updating parameters

        w1-=lr\*dw1

        b1-=lr\*db1

        lr\*=lr\_decay

    return w1,b1,loss\_history,loss\_history\_test,train\_acc\_history,val\_acc\_history

Running the linear classifier for 300 iterations with an initial learning rate of 0.014.

#defining parameters and running the linear classifier

iterations = 300

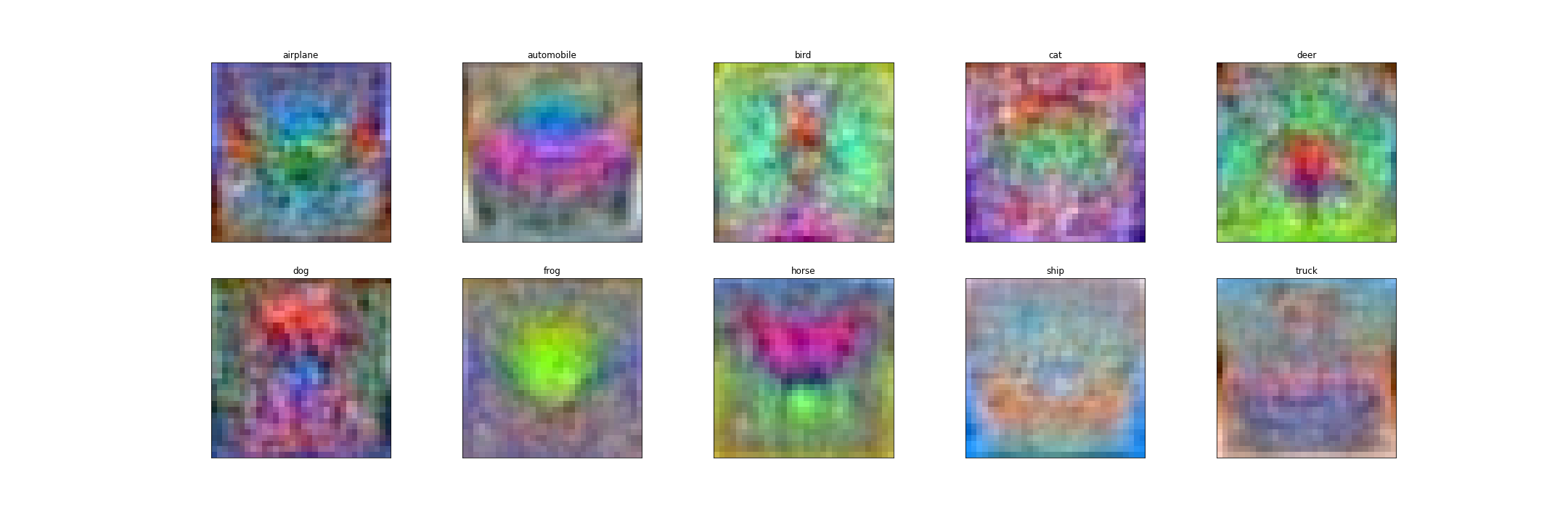
lr = 1.4e-2

lr\_decay= 0.999

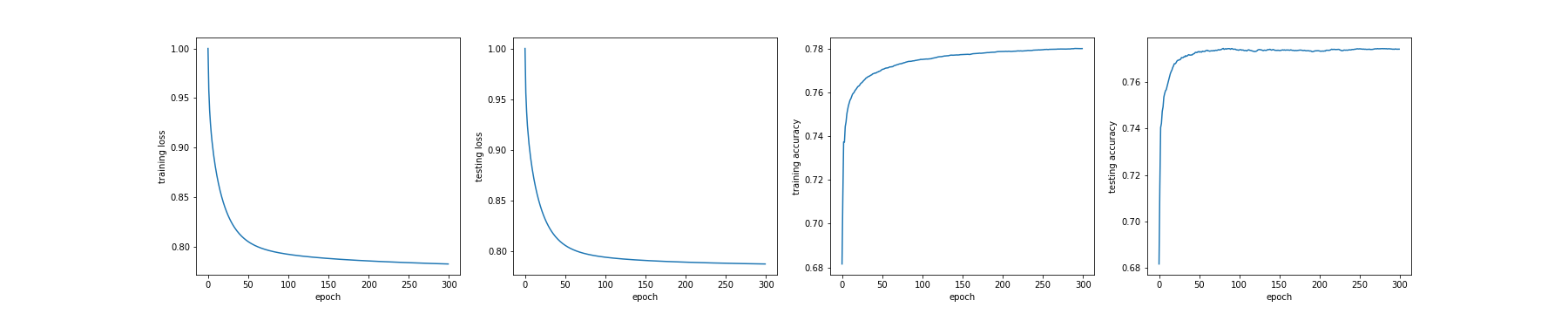
reg = 5e-6 #lamda(regularization constant for the loss function)

x\_train,y\_train,x\_test,y\_test=preproc(norm=True,reshape=True)

w1,b1,loss\_history,loss\_history\_test,train\_acc\_history,val\_acc\_history=linclas(x\_train,y\_train,x\_test,y\_test,K,Din,lr,lr\_decay,reg)

W1 weight array is of the shape 3072 x 10. Where 3072 is the length of the flattened input images and 10 is the number of classes. So, each node of the 10-node linear layer will correspond to each class in CIFAR10 data set. So, at the end of the training each node must be able to calculate a score to an image considering its’ similarity to the corresponding class. Since we are taking the vector product between the node and the input image, the node must be a similar image to label of its’ class. If we reshape the weight arrays back to an image, we can see this similarity. We can achieve this by reshaping each column of W1.

Let us observe the loss and accuracy of training data set. As we can observe the loss and accuracies have a large slope at the beginning but after a while slope become smaller. This is due to the low number of layers and nodes. Testing data also shows similar characteristics. So we can conclude that the linear classifier is not either underfitting or overfitting.



1. 2 Layer dense Network

Defining the two layer dense network

# defining two layer dense network

def layer2(x\_train,y\_train,x\_test,y\_test,batch\_size,H,K,Din,lr,lr\_decay,reg):

    Ntr = x\_train.shape[0]

    Nte = x\_test.shape[0]

    loss\_history = []

    loss\_history\_test = []

    train\_acc\_history = []

    val\_acc\_history = []

    seed = 0

    rng = np.random.default\_rng(seed=seed)

    #initializing weight and bias arrays

    std=1e-5

    w1 = (2/(Ntr\*Din))\*\*0.5\*np.random.randn(Din, H)

    w2 = (2/(H\*Din))\*\*0.5\*np.random.randn(H, K)

    b1 = np.zeros(H)

    b2 = np.zeros(K)

    for t in range(iterations):

        # mini batching the training data set

        indices = np.random.choice(Ntr,batch\_size)

        # shuffling the training data set to avoid overfitting

        rng.shuffle(indices)

        x=x\_train[indices]

        y=y\_train[indices]

        #forward pass

        h=1/(1+np.exp(-(x.dot(w1)+b1)))

        h\_test=1/(1+np.exp(-((x\_test).dot(w1)+b1)))

        y\_pred=h.dot(w2)+b2

        y\_pred\_test=h\_test.dot(w2)+b2

        # calculating the loss

        train\_loss=regloss(y\_pred,y,w1,w2)

        test\_loss=regloss(y\_pred\_test,y\_test,w1,w2)

        loss\_history.append(train\_loss)

        loss\_history\_test.append(test\_loss)

        # calculating accuracy

        train\_acc=accuracy(y\_pred,y)

        train\_acc\_history.append(train\_acc)

        test\_acc=accuracy(y\_pred\_test,y\_test)

        val\_acc\_history.append(test\_acc)

        if t%10 == 0:

            print('epoch %d/%d: loss= %f-- ,test loss= %f--,train accracy= %f--, test accracy= %f' % (t,iterations,train\_loss,test\_loss,train\_acc,test\_acc))

        # Backward pass

        dy\_pred=(1./batch\_size)\*2.0\*(y\_pred-y)

        dw2=h.T.dot(dy\_pred)+reg\*w2

        db2=dy\_pred.sum(axis=0)

        dh=dy\_pred.dot(w2.T)

        dw1=x.T.dot(dh\*h\*(1-h))+reg\*w1

        db1=(dh\*h\*(1-h)).sum(axis=0)

        # updating parameters

        w1-=lr\*dw1

        b1-=lr\*db1

        w2-=lr\*dw2

        b2-=lr\*db2

        lr\*=lr\_decay

    return w1,b1,w2,b2,loss\_history,loss\_history\_test,train\_acc\_history,val\_acc\_history

Preprocssing the input data set without pixel normalization to avoid underfitting. Because iwe don’t need normalization of data as we use sigmoid function to normalize the score of each node of the hidden layer.

x\_trainn,y\_trainn,x\_testn,y\_testn=preproc(norm=False,reshape=True)

Defining the parameters and running the 2 Layer dense neural network.

batch\_size = x\_trainn.shape[0] #initializing batch size as the size of whole data set

H=200

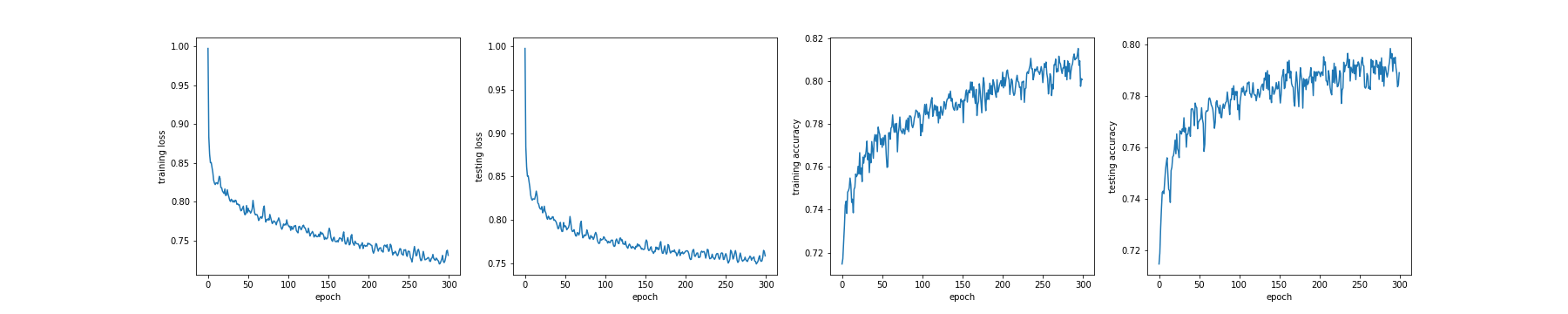
iterations = 300

lr = 1.4e-2

lr\_decay= 0.999

reg = 5e-6

w1n,b1n,w2n,b2n,loss\_historyn,loss\_history\_testn,train\_acc\_historyn,val\_acc\_historyn=layer2(x\_trainn,y\_trainn,x\_testn,y\_testn,batch\_size,H,K,Din,lr,lr\_decay,reg)

Training process shows oscillations be as we increase the hidden layers and nodes. But the overal accuracy has increased.

1. Stochastic gradient descent

Running the 2 Layer Dense Neural Network with mini batching of batch size 500.

batch\_size = 500

H=200

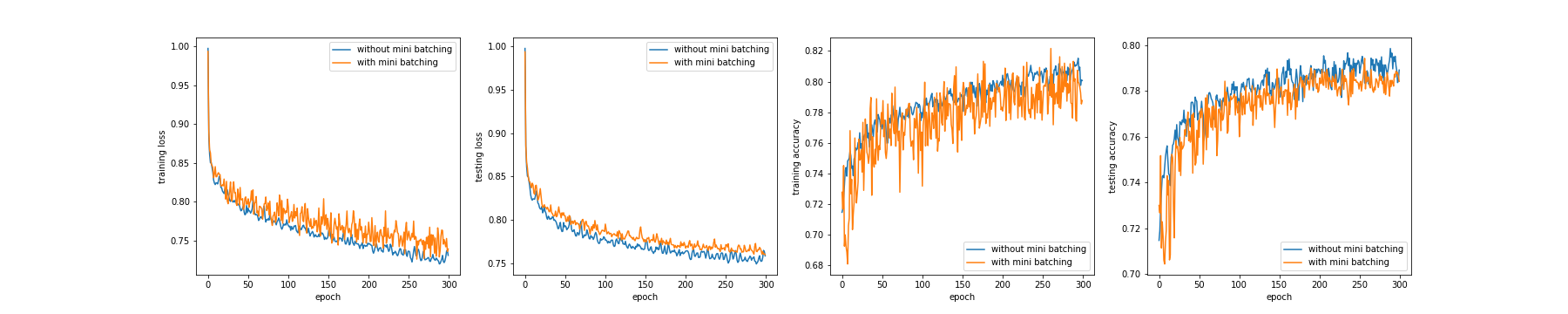
iterations = 300

lr = 1.4e-2

lr\_decay= 0.999

reg = 5e-6

w1m,b1m,w2m,b2m,loss\_historym,loss\_history\_testm,train\_acc\_historym,val\_acc\_historym=layer2(x\_trainn,y\_trainn,x\_testn,y\_testn,batch\_size,H,K,Din,lr,lr\_decay,reg)

Comparison of gradient descent with and without mini batching.

As we can observe with mini stochastic gradient descent training process shows huge oscilations in loss and accuracy functions. This is due to random selection of data from mini batches. But the algorithm becomes fast. Eventhough the training process is oscillating, the overall is sufficiently similar to the gradient discent without minibatching. So it is a good tradeoff.

1. CNN

Preprocessing data without normalization of pixels and without reshaping the images.

x\_trainc,y\_trainc,x\_testc,y\_testc=preproc(norm=False,reshape=False)

CNN coded using Keras.models.Sequential.

model = models.Sequential()

model.add(layers.Conv2D(32,(3,3),activation='relu',input\_shape=(32,32,3)))

model.add(layers.MaxPool2D((2,2)))

model.add(layers.Conv2D(64,(3,3),activation='relu'))

model.add(layers.MaxPool2D((2,2)))

model.add(layers.Conv2D(64,(3,3),activation='relu'))

model.add(layers.MaxPool2D((2,2)))

model.add(layers.Flatten())

model.add(layers.Dense(64,activation='relu'))

model.add(layers.Dense(10))

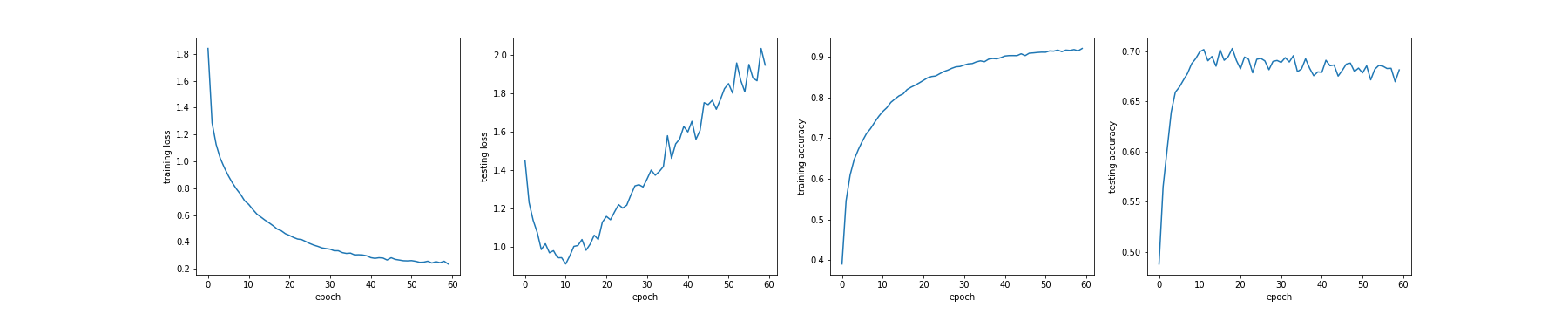
model.summary()

model.compile(optimizer='adam',loss=tf.keras.losses.CategoricalCrossentropy(from\_logits=True),metrics=["accuracy"])

history=model.fit(x\_trainc,y\_trainc,epochs=60,batch\_size=50,validation\_data=(x\_testc,y\_testc),)

weights=model.get\_weights()

para=model.optimizer.get\_config()



We can clearly see that the CNN is overfitting. Even though the