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.

preproc(norm,reshape)

Parameters,

- norm: Boolean value. Set to True, to normalize training and testing data sets.
- Reshape: Boolean value. Set to True, to reshape training and testing data sets.

1. Linear Classifier

Loss function and the accuracy function is predefined to reduce the complexity in coding.

- Loss function: Mean sum of squared errors with regularization.
- Accuracy: Normalized difference between the true label and the label with the highest score is used to calculate the accuracy.

I tested code with different initial learning rates. Normally 0.01 is used as the initial learning as a rule of thumb. I tested for higher learning rates and the loss got exploded at 0.016. So I’m using 0.014 as initial learning rate as safety margin. I’m using 0.999 as the learning rate decay for a faster learning as we are only training for 300 epochs.

I have defined a function to run linear classifier for a given data set.

linclas(x_train,y_train,x_test,y_test,lr,lr_decay,reg):

Parameters,

- x_train: Training data set
- y_training: labels for training data
- x_test: validation data set
- y_test: labels for validation data
- lr: learning rate
- lr_decay: learning rate decay
- reg: regularization parameter

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

```
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,lr,lr_decay,reg)
```

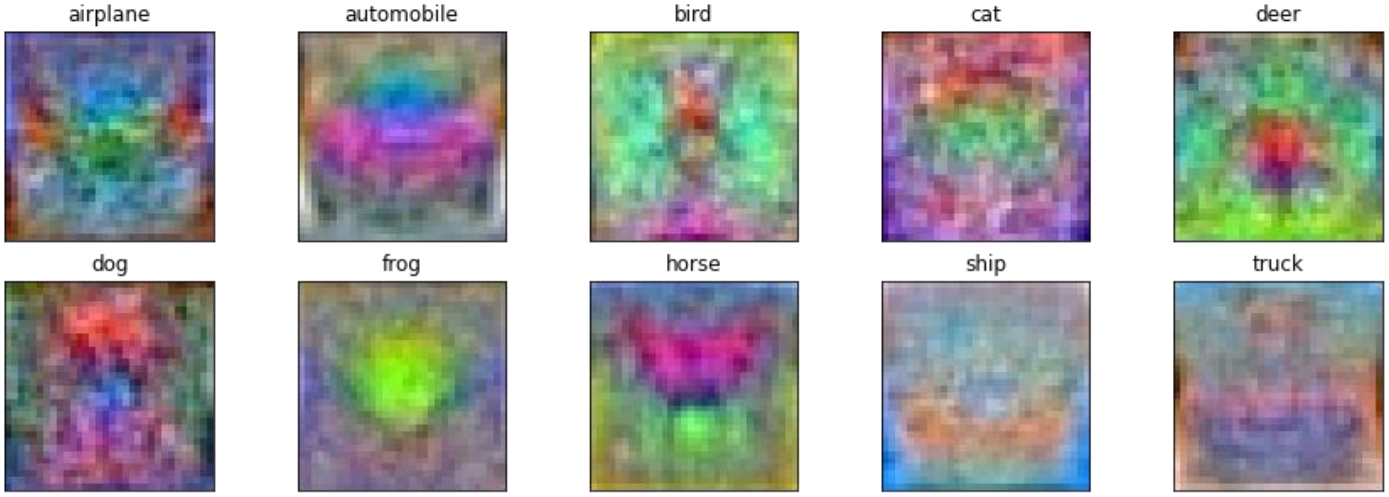
```
loss= 0.783117-- ,test loss= 0.787732--,train accracy= 0.779952--, test accracy= 0.774150
```

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(each column of W1) 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.

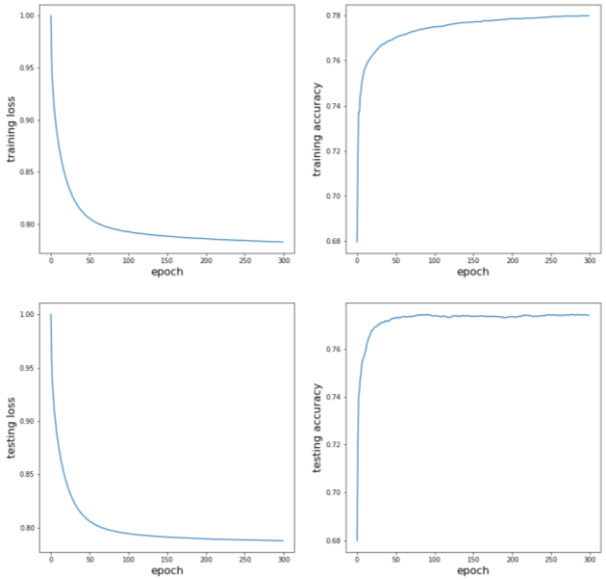
Our linear score function is,

$$f(x) = Wx + b$$

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 and validation processes.



As we can observe the loss and accuracies have a large gradient at the beginning but after a while gradient has 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

2. Two layer dense Network

I have defined a function to run the two layer dense neural network,

layer2(x_train,y_train,x_test,y_test,batch_size,H,lr,lr_decay,reg,sgd=False)

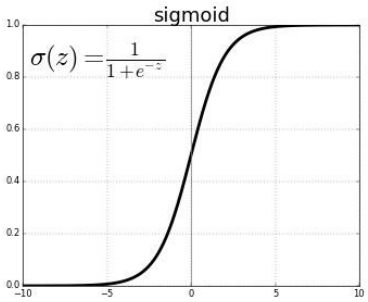
Parameters,

- x_train: Training data set
- y_training: labels for training data
- x_test: validation data set
- y_test: labels for validation data
- batch_size: set a value for training data mini batching. Input the whole training data set size if mini batching is not required.
- H: number of hidden nodes

- lr: learning rate
- lr_decay: learning rate decay
- reg: regularization parameter
- sgd: set value to True, to activate stochastic gradient descent with random data.

In the linear classifier we normalized the pixel values. Otherwise the score values won't be in the range 0-1. Because we didn't use any activation function to map the output of the linear layer.If we don't normalize the data for the linear classifier it will take a longer time to train.

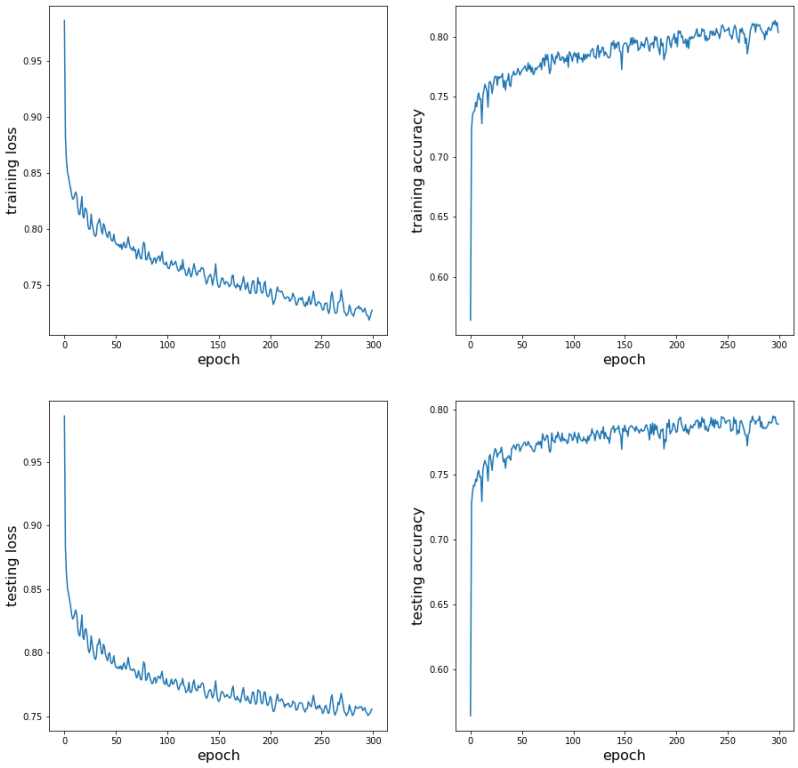
But in this two layer model we are using hidden layer followed by a sigmoid activation function. So all the outputs from the hidden layer get mapped to a value between 0 and 1.



Also unnecessary normalizations can lead to an underfitting model.

Running two layer dense network for **0.014 initial learning rate.**

```
x_trainn,y_trainn,x_testn,y_testn=preproc(norm=False,reshape=True)
batch_size = x_trainn.shape[0]
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=la
yer2(x_trainn,y_trainn,x_testn,y_testn,batch_size,H,lr,lr_decay,reg)
>>>loss= 0.725935-- ,test loss= 0.754461--,train accracy= 0.808128--, test accracy=
0.790950
```



Even though the training process show fluctuations, the accuracy of the Neural Network has increased compared to the linear classifier. And the validation loss has drastically decreased compared to the linear classifier.

3. Stochastic gradient descent

In stochastic gradient descent, instead of using the whole training data set for the forward and backward propagation, we use mini batches of data. In each epoch we run forward and propagation for each mini batch, until all data training data is used.

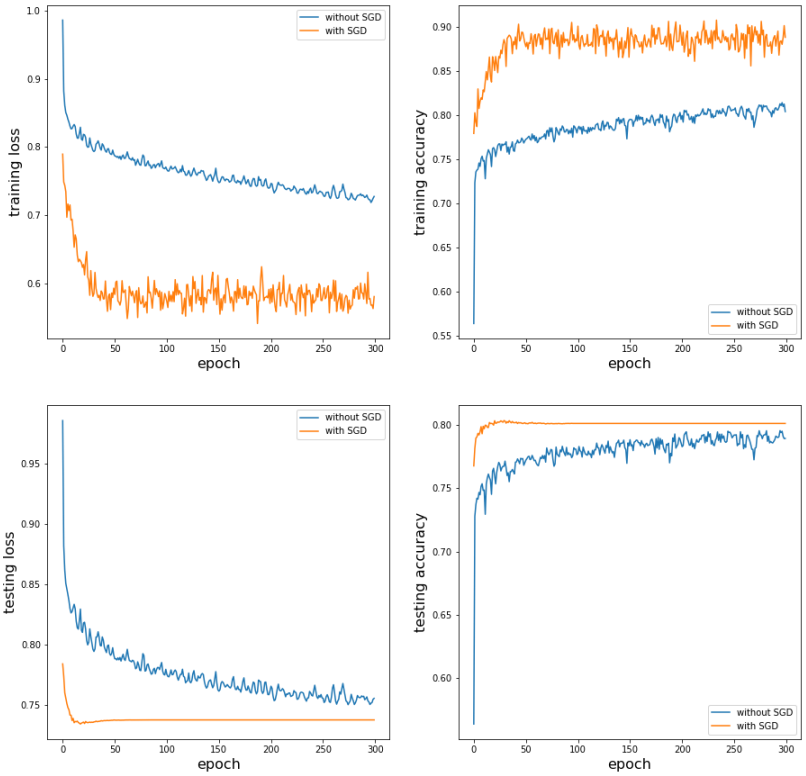
Process,

- In each epoch training data is randomly shuffled.
- Then all training data is divided to mini batches of the given batch size.
- We run forward and backward propagation for all mini batches.
- Then in the next epoch we repeat above process again.

I have defined the function **layer2sgd()** to run stochastic gradient descent.

```
batch_size = 500
H=200
iterations = 300
lr = 1.4e-2
lr_decay= 0.999
reg = 5e-6
x_trainn,y_trainn,x_testn,y_testn=preproc(norm=False,reshape=True)
w1o,b1o,w2o,b2o,loss_historyo,loss_history_testo,train_acc_historyo,val_acc_historyo=la
yer2sgd(x_trainn,y_trainn,x_testn,y_testn,batch_size,H,lr,lr_decay,reg)
```

```
loss= 0.599417-- ,test loss= 0.737728--,train accracy= 0.877800--, test accracy= 0.800910
```



As we can observe with Stochastic gradient descent training process shows huge oscilations. But it has achived a low loss and a high accuracy in few epochs. That loss is even lower than the loss achieved by Normal gradient descent after 300 epochs.

Explanation:

When we are using the whole batch at once for the gradient descent, the effect of bad data get normalized. So the Neuaral Network can have a lower loss. But at the same time the effect of good data is also normalized. So it takes lot of iterations to reach the global minima.

But when we are using stochastic gradient descent we run forward and backward propagation for small batches of data. So the good data have a huge impact in gradient descent. So the neural network can reach a global minima in a less number of epochs. But at the same time, the effect of bad data is also magnified. This causes high fluctuation in loss and accuracy.

4. CNN

Data is preprocessed with normalization of pixels and without reshaping the images.

```
x_trainc,y_trainc,x_testc,y_testc=preproc(norm=True,reshape=False)
```

CNN is coded using Keras.models.Sequential. We can get the summery of the CNN using model.summary().

CNN has 73,418 learnable parameters and 0 non learnable parameters

We can get the summary of the CNN by using model.summary(). According to the output CNN has,

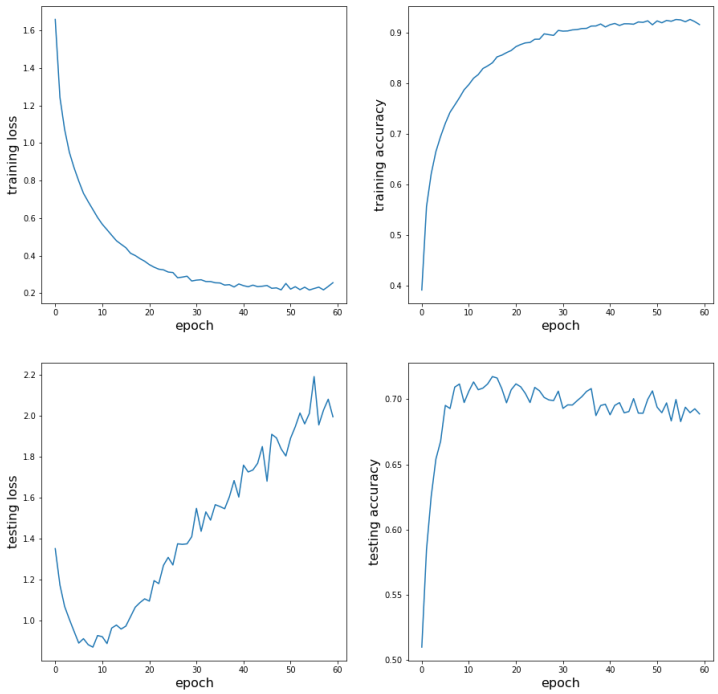
- Total parameters: 73418
- Learnable parameters: 73418
- Non Learnable parameters: 0

Here we are using SGD with momentum as the optimizer. It is a mix of mini batching process and using the past loss values regulate the gradient descent.

- Initial learning rate: 0.01
- Momentum: 0.9

Momentum decides how much past losses are affecting the current gradient descent. After running the CNN for 60 epochs, I got the result below,

```
loss: 0.2560 - accuracy: 0.9169 - val_loss: 1.9944 - val_accuracy: 0.6888
```



Eventhough our model has lot of convolutional and dense layers, we can observe that model is overfitting after 20 epochs. This can happen due to many reasons. Most probably gradients are at a local minima instead of the global minima. We can overcome this by training for large number of epochs and increasing learning rate.

Link to the original code: [Image_processing/A04_180437U.ipynb at main · OshanJayawardana/Image_processing (github.com)]

Link to the repository: https://github.com/OshanJayawardana/Image_processing.git