Assignment 3b- Multilayer Neural Networks

Created Multilayer Neural Network with parameters such as lr, loss function, architecture, mini-batch-size, and learning rate.

Example: hyperparams = {'arch': [100, 100], 'r': 26, 'lr': 0.1, 'mini_batch_size': 100, 'loss': 'mse', 'learning_rate':'adaptive', 'activation':'relu'}

I have used Xavier Initialization in training of our model to converge faster. (Initialize our weights with mean zero and variance of 2/ (number of inputs + number of outputs))

Part b - Varying single hidden layer size

For different layer size, convergence criteria are different because more neurons take higher training time or number of epochs to converge. Thus, I limited number of epochs in this case.

Convergence criteria for arch=[100] =>

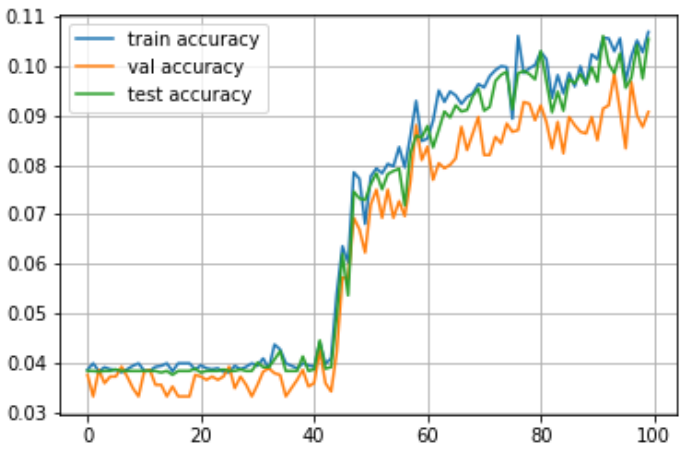
if(epoch>11 and (val_log[-1]-val_log[-2]) < 0.0001 and (np.average(val_log[-10:])-val_log[-11])<0.0001): break

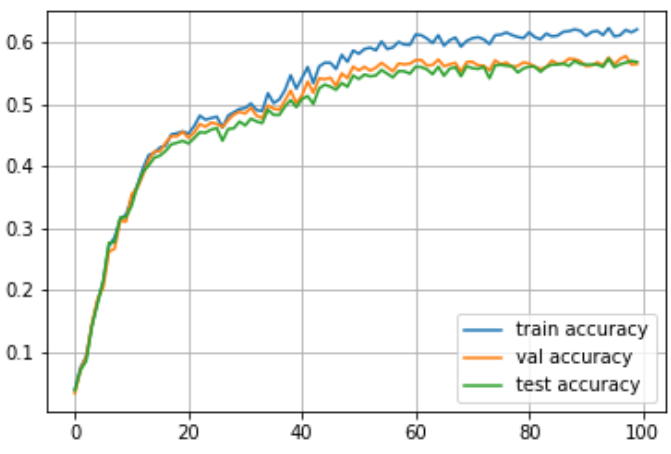
Layer size vs exec. time [100 epochs with Sigmoid]:

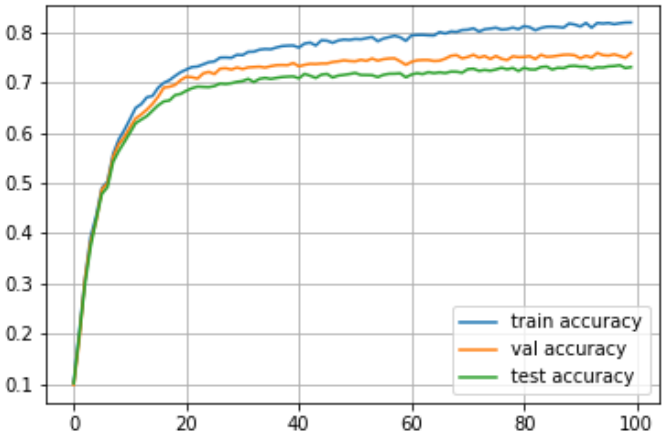
Model Details: Arch: [100], r: 26, lr: 0.1, mini_batch_size: 100, Loss: 'mse','learning_rate':'constant

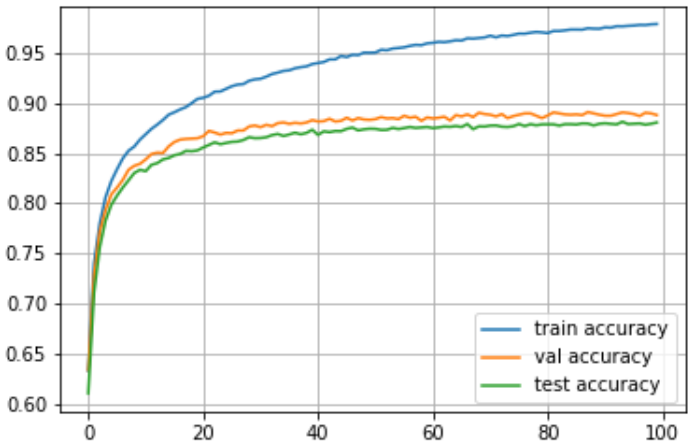
Layer Size	Exec Time (sec)	Train Accuracy	Test Accuracy
1	48.07287645339966	0.1068	0.10538461538461538
5	55.723833322525024	0.6203	0.5678461538461539
10	56.60548114776611	0.8197	0.7309230769230769
50	79.56502532958984	0.978	0.8803076923076923
100	112.03223538398743	0.9926	0.9018461538461539

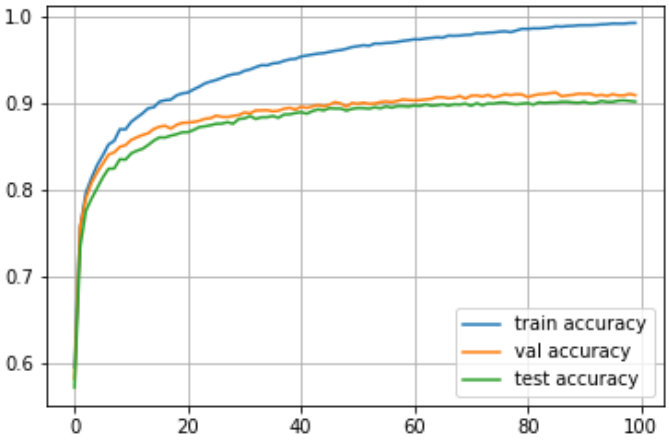
Below are the plots of accuracy v/s numbers of epochs for various architectures [1, 5, 10, 50, 100]

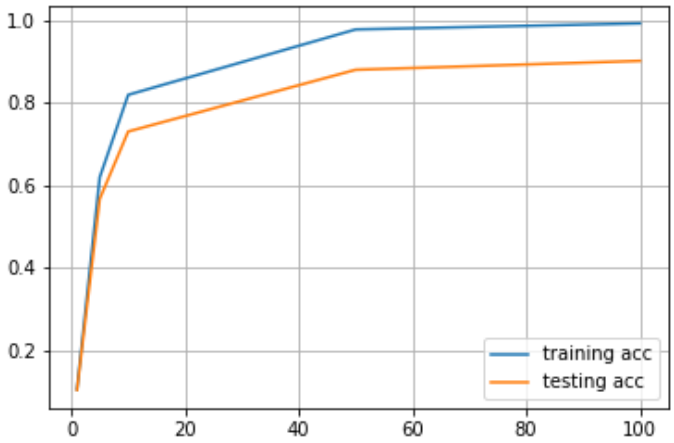












Last plot shows the graph of accuracy vs time taken

Observation:

Increasing the number of neurons increased the accuracy on the training and test data. This is evident because more neurons introduce more modularity and non-linearity in the trained model.

Also, increasing the number of neurons further increased the accuracy. First hidden layer should have same number of neurons as that of size of input feature vector.

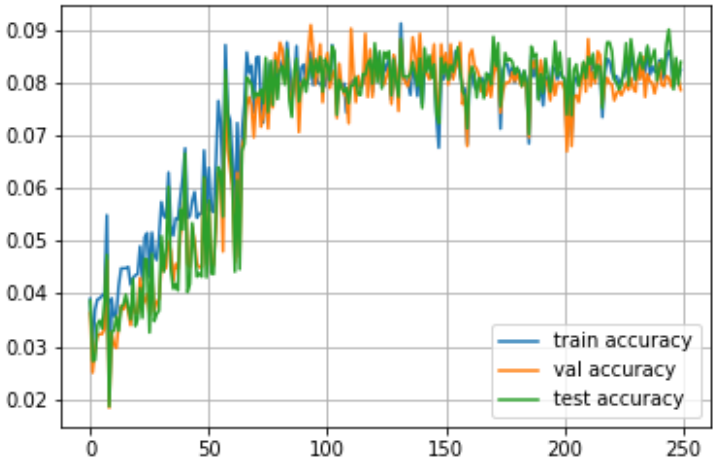
250 epochs used for every size of layer. Although, smaller size architectures require less epochs. The convergence condition will take care of that.

Part C – Adaptive Learning

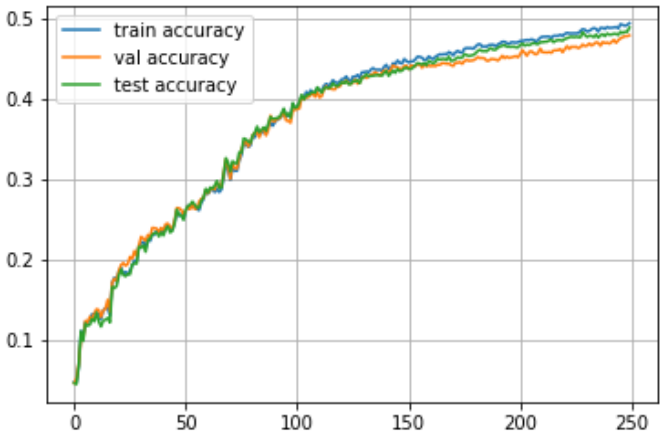
Again, stopping criteria for every different size of architecture will be different. Thus, I ran the model for the number of epochs, on which largest architecture converged i.e. 250.

I came to 250 epochs due to the convergence criteria:
While (epoch>11 and (val_log[-1]-val_log[-2]) < 0.0001 and (np.average(val_log[-10:])-val_log[-11])<0.0001) is FALSE.

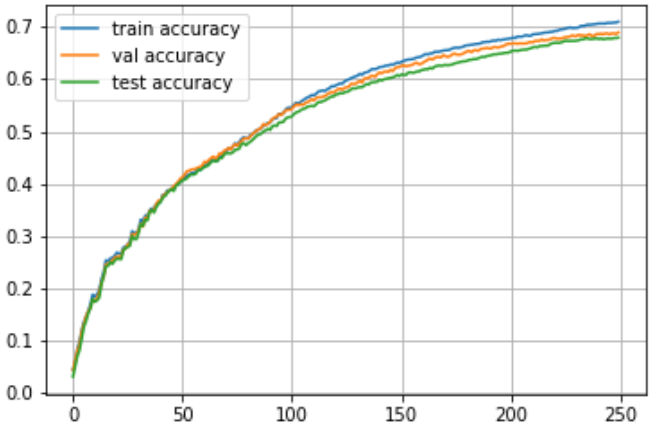
Below are the plots of accuracy v/s number of epochs



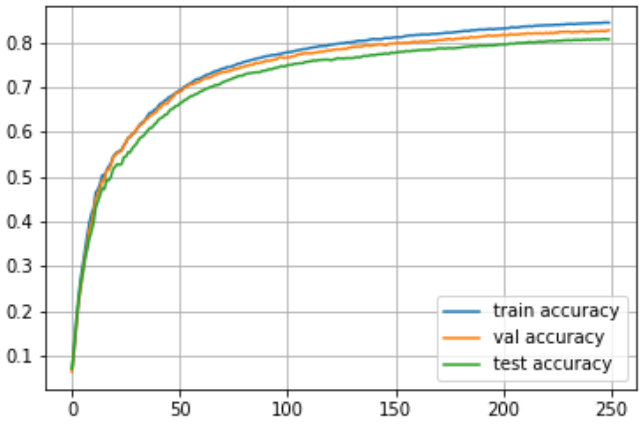
[1]_layer_size.png



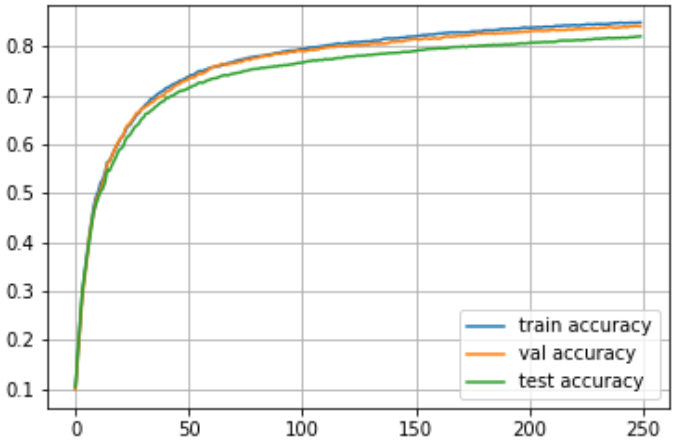
[5]_layer_size.png

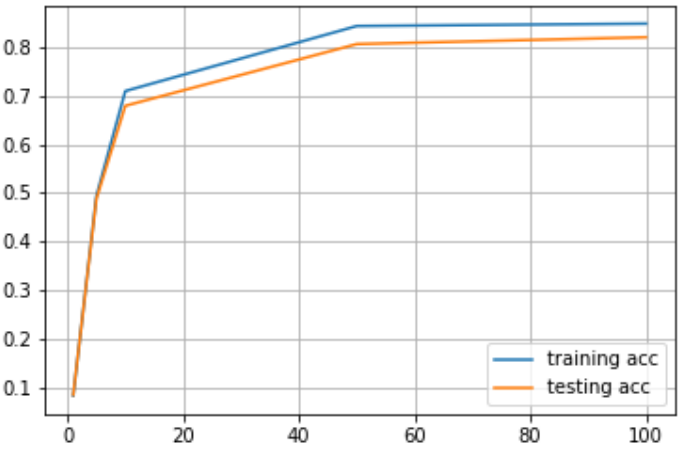


[10]_layer_size.png



[50]_layer_size.png



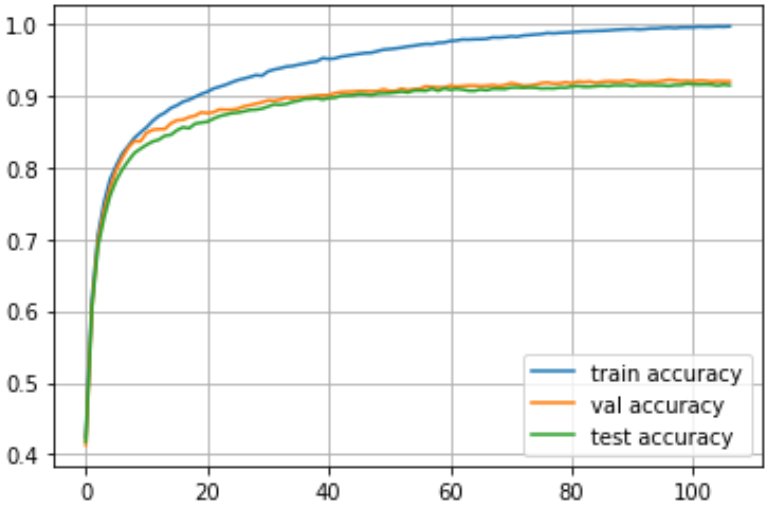


[100]_layer_size.png	Accuracy v/s Exec. Time

Part D - Introduction of 2 hidden layers and ReLU

Model:
hyperparams = {'arch': [100, 100], 'r': 26, 'lr': 0.1, 'mini_batch_size': 100, 'loss': 'mse', 'learning_rate':'adaptive', 'activation':'relu'}

1. Train accuracy: 0.9965
2. Val accuracy: 0.92
3. Test accuracy: 0.9144615384615384



Observation:

ReLU turns out to be better than Sigmoid because gradient vanishes at backpropagation in case of Sigmoid. In case of ReLU, there is no issue of vanishing gradient.

Thus, activation of ReLU results in more accuracy over validation and test data as compared to Sigmoid.

As compared to part 2b, there is **more modularity** in the model. Test accuracy is very much improved with these hyperparameters as compared to part 2b.

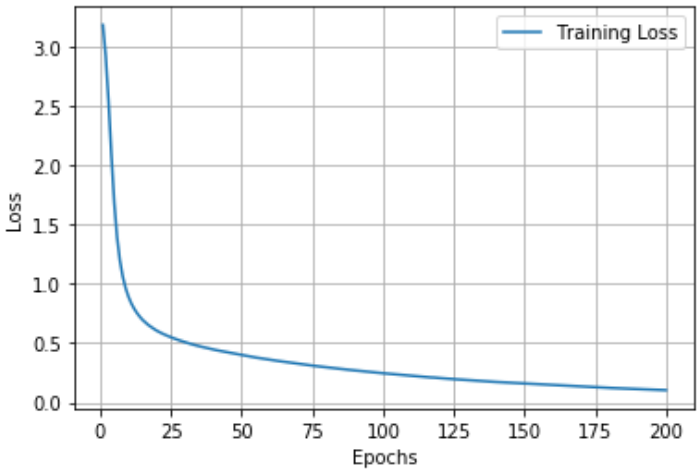
Single hidden layer becomes the bottleneck towards the increase of accuracies even with adaptive learning.

Part E – sklearn.MLPClassifier

Model:
clf = MLPClassifier(hidden_layer_sizes=(100, 100), activation='relu', solver='sgd', learning_rate='adaptive', batch_size=100)

Train Accuracy: 0.9874615384615385
Test Accuracy: 0.9041538461538462

The test results are similar to part 2d but **sklearn library took less epochs to converge** onto a stable accuracy over test and validation data set. This means, convergence criteria of library is better than that implemented in part 2d.



Part F - Self Implementation till part d + cross-entropy loss

I have implemented cross entropy loss. Since we want to predict probabilities, it would be logical for us to define softmax nonlinearity on top of our network and compute loss given predicted probabilities. However, there is a better way to do so. If we write down the expression for crossentropy as a function of softmax logits (a), you'll see:

$$loss = -log\, e^{\frac{a_{correct}}{\sum_i e^{a_i}}}$$

If we take a closer look, we'll see that it can be rewritten as:

$$loss = -a_{correct} + log \sum_i e^{a_i}$$

It's called Log-softmax and it's better than naive log(softmax(a)) in all aspects:

- Better numerical stability
- Easier to get derivative right
- Marginally faster to compute

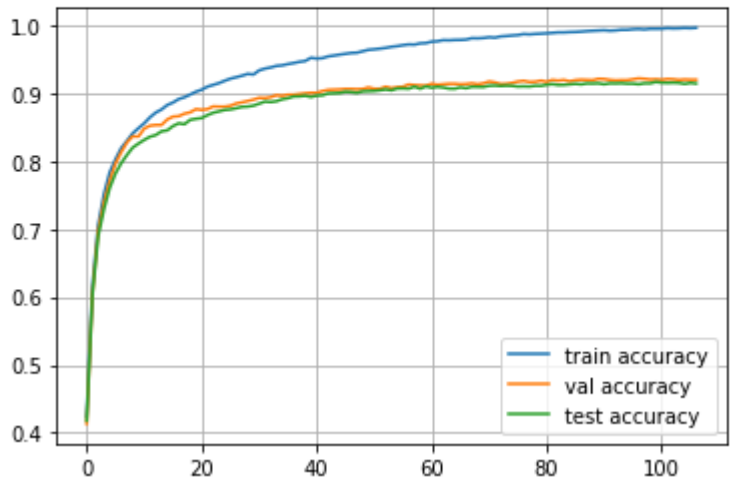
Model:

hyperparams = {'arch': [100, 100], 'r': 26, 'lr': 0.1, 'mini_batch_size': 100, 'loss': 'mse', 'learning_rate':'adaptive', 'activation':'relu'}

Convergence condition:

```
if(epoch>6 and abs(np.average(train_log[-5:])-train_log[-6])<0.0001):
  break
```

```
Epoch 106
Train accuracy: 0.9965
Val accuracy: 0.92
Test accuracy:  0.9144615384615384
```



Prediction accuracy in this model is surely comparative to part 2e. Even the accuracy is better for this specific data set. The possible reason might be the limit of max_iter in our sklearn model. The learning rate mentioned above is highly data dependent.