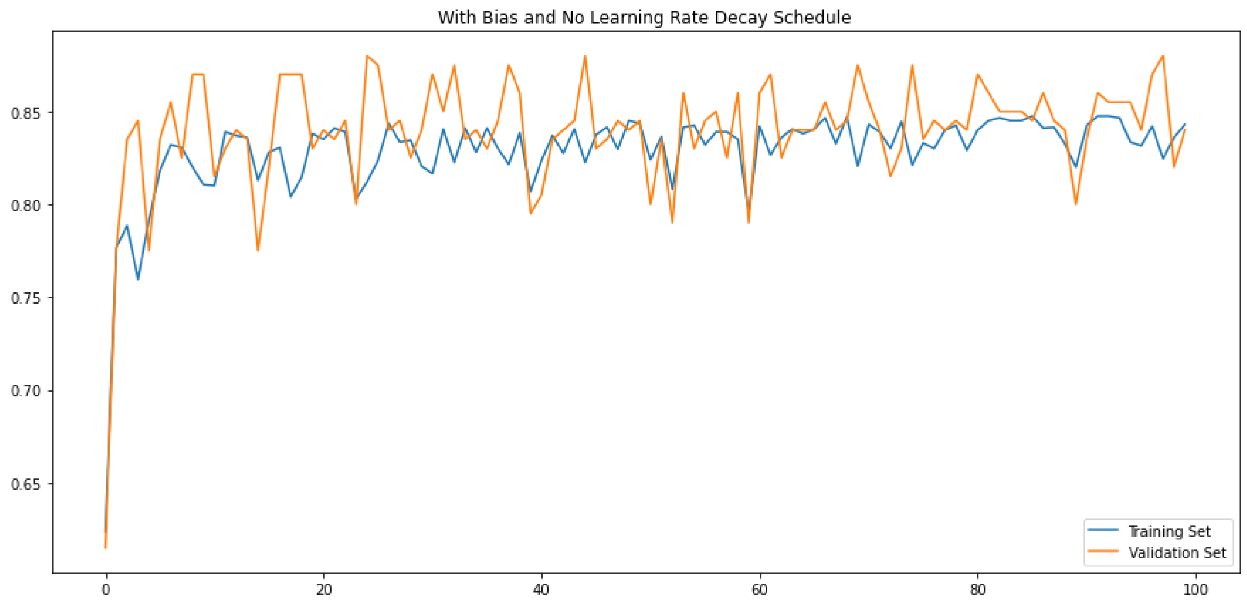
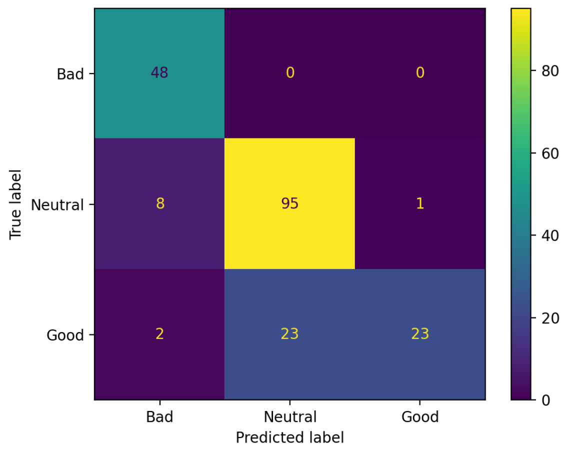
1. (4.0 points) Implement a simple feed-forward neural network in PyTorch to make classifications on the “insurability” dataset. The architecture should be the same as the one covered in class (see Slide 11-8). You can implement the neural network as a class or by using in-line code. You should use the SGD optimizer(), and you must implement the softmax() function yourself. You may apply any transformations to the data that you deem important. Train your network one observation at a time. Experiment with three different hyper-parameter settings of your choice (e.g. bias/no bias terms, learning rate, learning rate decay schedule, stopping criteria, temperature, etc.). In addition to your code, please provide

I. Learning curves for the training and validation datasets,



With the baseline, we have a validation accuracy of 0.88 and a test accuracy of 0.84.

II. Final test results as a confusion matrix and F1 score



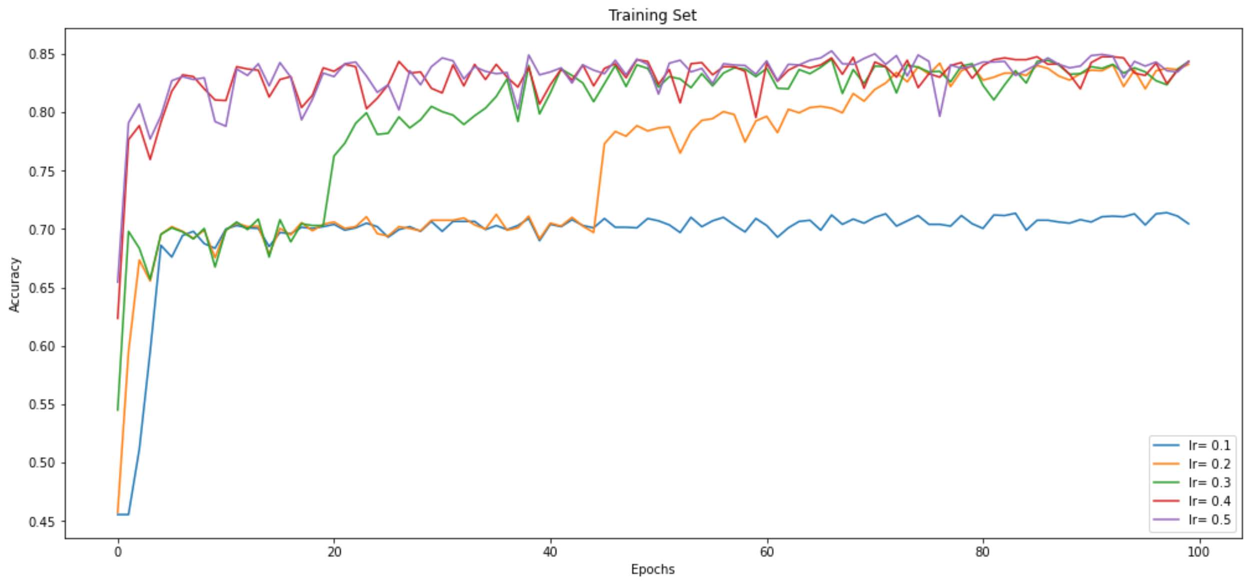
Top Validation Accuracy: 0.88
Test Accuracy is: 0.84
F1 Score:
Bad is 0.9142857142857143,
Neutral is 0.8660714285714286,
Good is 0.647887323943662

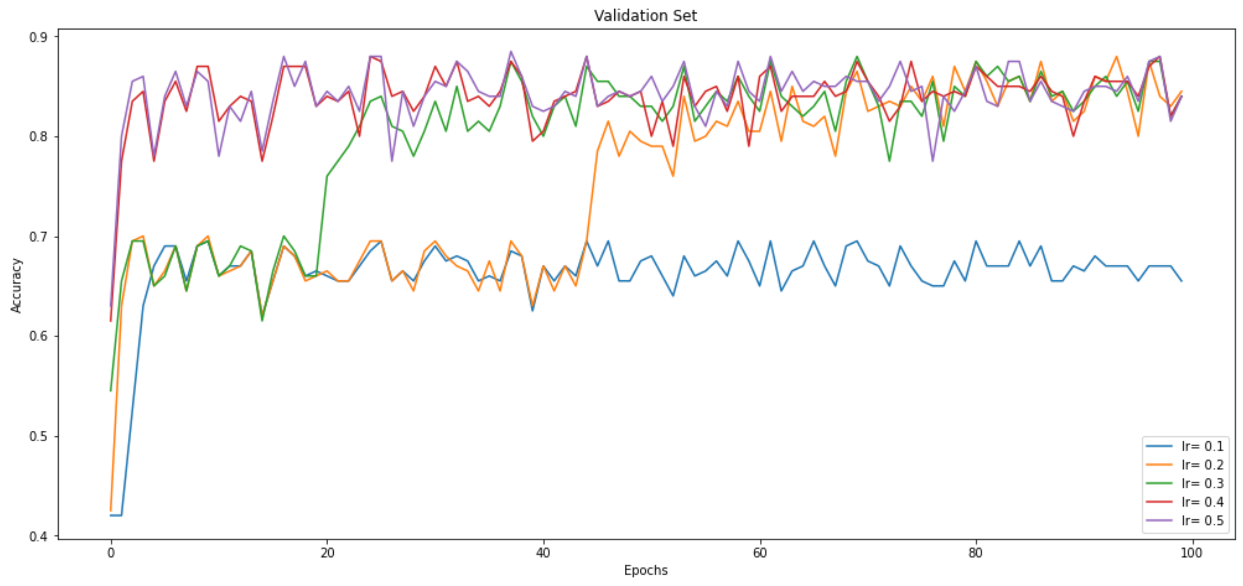
III. A description of why using a neural network on this dataset is a bad idea.

The cost is huge for training the neural network, especially when we are using stochastic gradient descent which updates the model for each observation. There are better alternatives like a decision tree that is well qualified for this task and it takes less amount of time. Given the small datasets (only 2000 observations), the model can easily overfit by “remembering” the datasets.

IV. A short discussion of the hyper-parameters that you selected and their impact.

Difference Learning Rate:





When the learning rate is low (0.1), the model could not improve anymore after it reaches the accuracy of about 60%-70%. This situation changed after the learning rate increased. Starting from a learning rate of 0.2 to 0.5, the accuracy of the model could finally reach the level of over 80%, however, the higher the learning rate is, the less time it takes to reach the final state (where the performance stops improving), before that, they all meet the bottleneck at the accuracy around 60%-70%.

With lr = 0.1:
Top Validation Accuracy: 0.695
Test Accuracy is: 0.705

With lr = 0.2:
Top Validation Accuracy: 0.88
Test Accuracy is: 0.835

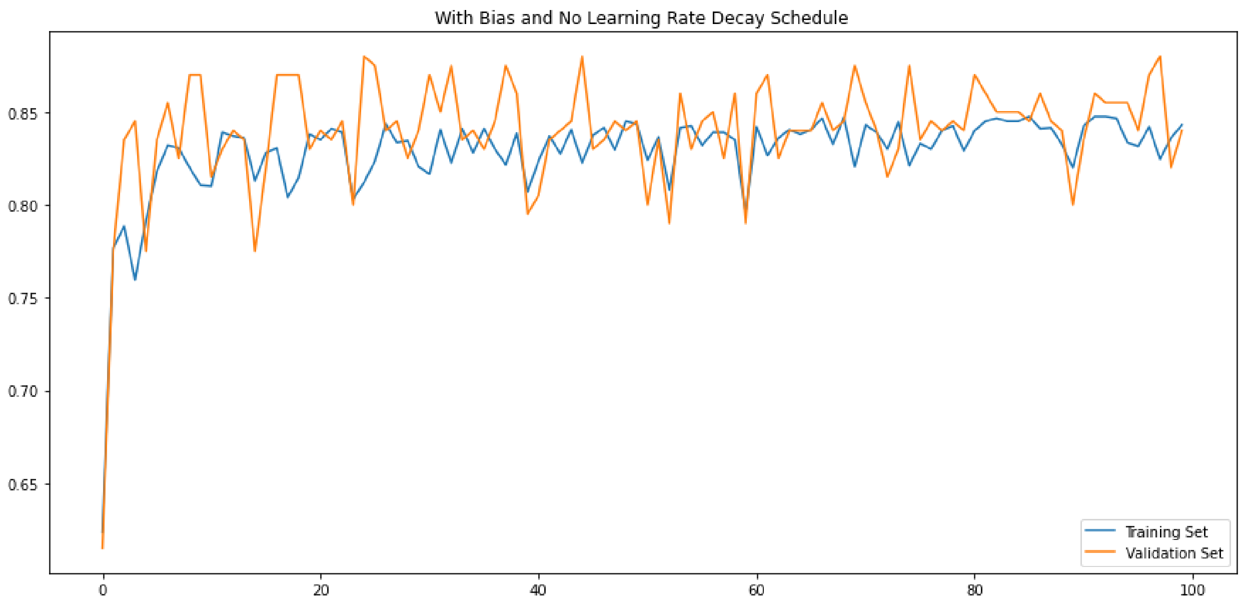
With lr = 0.3:
Top Validation Accuracy: 0.88
Test Accuracy is: 0.835

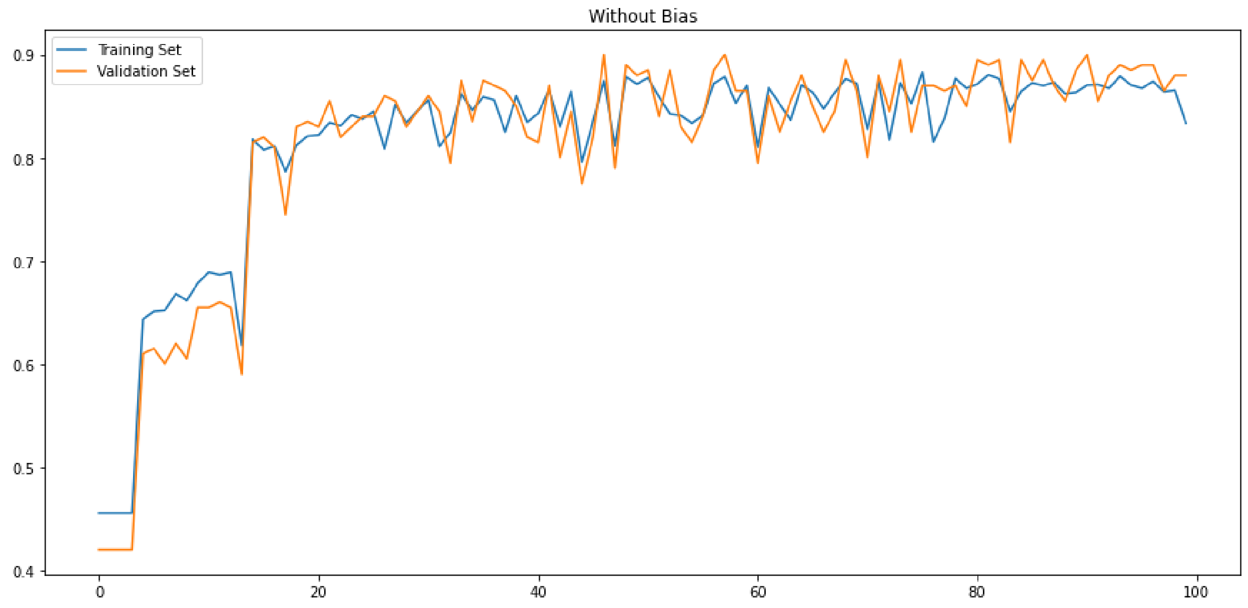
With lr = 0.4:
Top Validation Accuracy: 0.88
Test Accuracy is: 0.84

With lr = 0.5:
Top Validation Accuracy: 0.885
Test Accuracy is: 0.82

The test accuracy, as well as the validation accuracy, tend to be higher when the learning rate is higher, however, there is no linear correlation found between the learning rate and the test accuracy.

With/Without Bias:

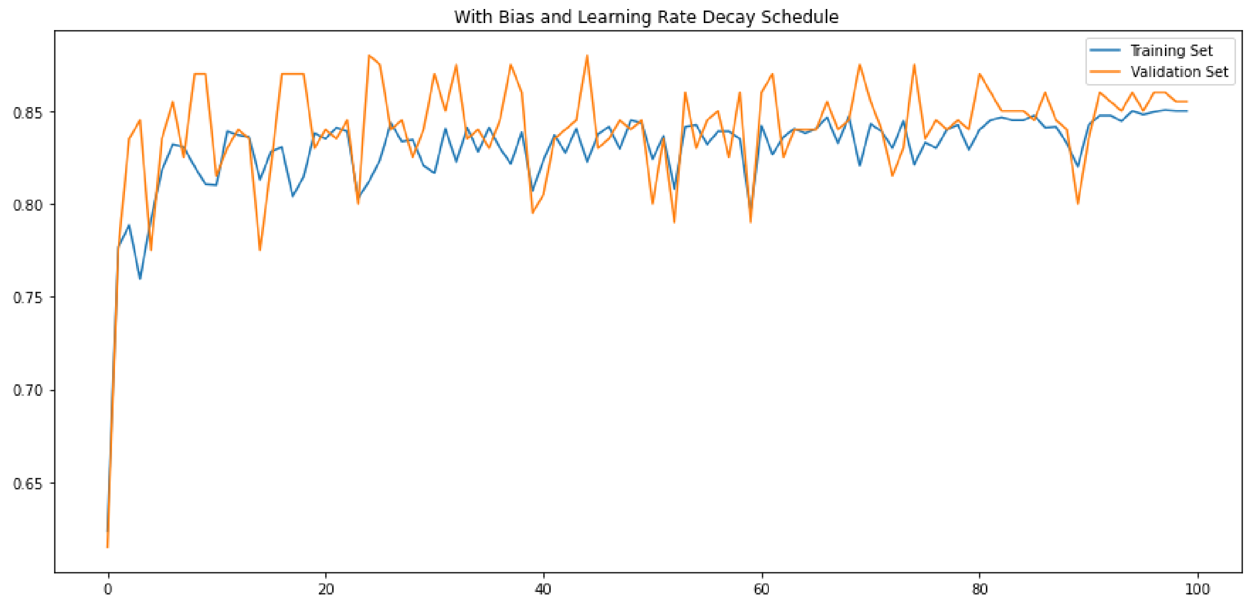


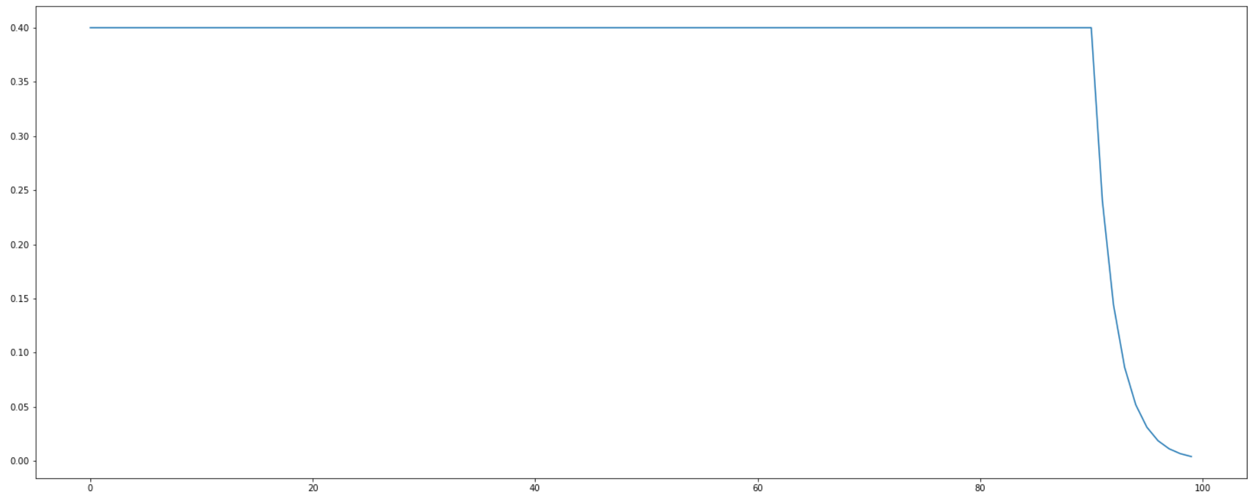


Top Validation Accuracy: 0.9
Test Accuracy is: 0.865

With bias, the model learns fast and reaches a relatively good performance at the very beginning, and it stops improving afterward. Without bias, the model meets a bottleneck at the accuracy of about 60%-70%, and it takes a few more epochs to break through the bottleneck and reach the final state (where the performance stops improving).

With Learning Rate Decay Schedule:





Top Validation Accuracy: 0.88
Test Accuracy is: 0.83

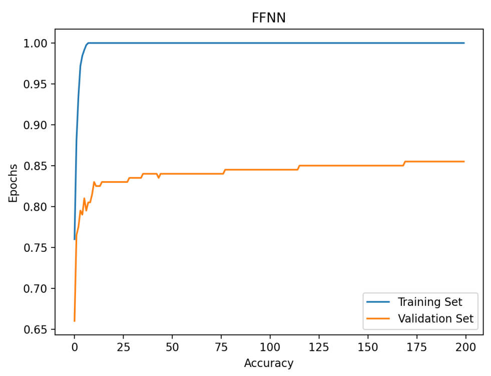
Based on figure two, we can see that we didn't apply learning rate decay until trained 90% epochs, as the learning rate smoothly decreased, the accuracy of the model did not improve. But we can see that the learning curve is less fluctuated and two curves finally converge at a certain point. Other than the last ten epochs, the former 90 epochs have almost the same shape.

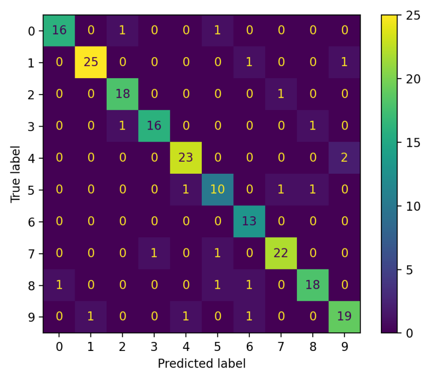
2. (4.0 points) Implement neural network in PyTorch with an architecture of your choosing (a deep feed-forward neural network is fine) to perform 10-class classification on the MNIST dataset. Apply the same data transformations that you used for Homework #2. You are encouraged to use a different optimizer and the cross-entropy loss function that comes with PyTorch. Describe your design choices, provide a short rationale for each and explain how this network differs from the one you implemented for Part 1. Compare your results to Homework #2 and analyze why your results may be different. In addition, please provide the learning curves and confusion matrix as described in Part 1.

Design choice:

1. Optimizer change from SGD to Adam
   The change of optimizer actually improves the accuracy of the model. Also, the learning curve becomes smooth compared to all the learning curves we got in part 1. The change of optimizer also comes with the change of batch size
2. The batch size change from 1 to 64
   The increase of batch size combining the change of optimizer actually improves the accuracy of the model. More importantly, it largely shortened the running time.
3. One more hidden layer and the change of the size of each layer
   Different from part 1, there are 784 pixels of each input, thus, we have the input size as 784, the first hidden layer has the size 400, and the second hidden layer with the size of 150, and the output layer is 10.

Compared to HW2:
This Neural Network model actually did as well as the KNN and better than the Kmeans. However, typically there is no training process in KNN, it has to go through every single data within the training set each time we want to do a prediction. This would be a downside of KNN especially when the training set is huge. Also, it took only a few epochs to train the NN to get a good enough accuracy. By increasing the batch size, we can shorten the training time relatively. After all, NN required less training time than both KNN and Kmeans, and it provided as good prediction accuracy as the KNN.



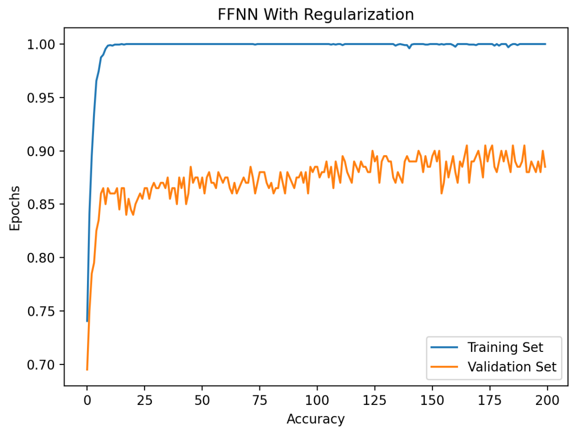


Top Validation Accuracy: 0.855
Test Accuracy is:0.9

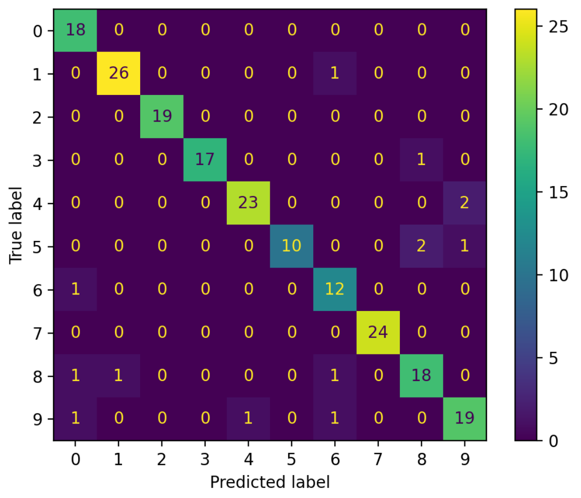
F1 Score:

0 is 0.9142857142857143	5 is 0.7692307692307693
1 is 0.9433962264150944	6 is 0.896551724137931
2 is 0.9230769230769231	7 is 0.9166666666666666
3 is 0.9142857142857143	8 is 0.8780487804878048
4 is 0.92	9 is 0.8636363636363636

3. (2.0 points) Add a regularizer of you choice to the 10-class classifier that you implemented for Part 2. Describe your regularization, the motivation for your choice and analyze the impact of the performance of the classifier. This analysis should include a comparison of learning curves and performance.



Top Validation Accuracy: 0.905
Test Accuracy is:0.94
F1 Score:
0 is 0.923076923076923
1 is 0.9629629629629629
2 is 1.0
3 is 0.9714285714285714
4 is 0.9387755102040817
5 is 0.8695652173913044
6 is 0.8571428571428571
7 is 1.0
8 is 0.8571428571428571
9 is 0.8636363636363636



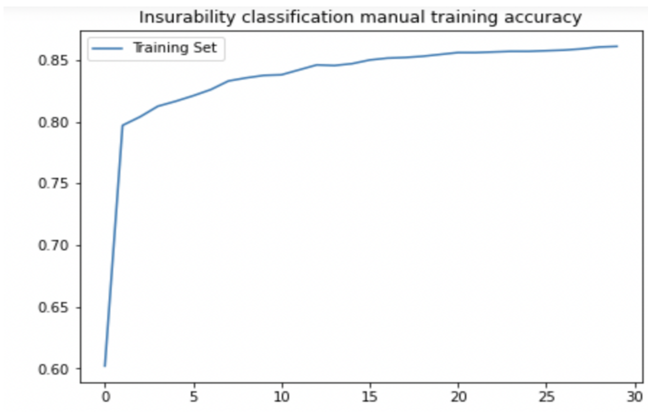
	0	1	2	3	4	5	6	7	8	9
0	18	0	0	0	0	0	0	0	0	0
1	0	26	0	0	0	0	1	0	0	0
2	0	0	19	0	0	0	0	0	0	0
3	0	0	0	17	0	0	0	0	1	0
4	0	0	0	0	23	0	0	0	0	2
5	0	0	0	0	0	10	0	0	2	1
6	1	0	0	0	0	0	12	0	0	0
7	0	0	0	0	0	0	0	24	0	0
8	1	1	0	0	0	0	1	0	18	0
9	1	0	0	0	1	0	1	0	0	19

We chose to use dropout and weight decay to implement our regularization. We chose them because they are easy to implement. Moreover, they are commonly used for regularization for the following reasons.

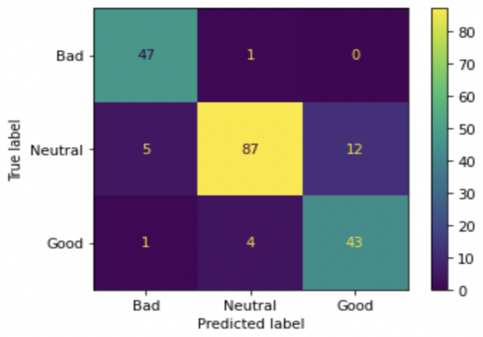
For dropout, which has proven to be an effective technique for regularization and preventing the co-adaptation of neurons as described in the paper Improving neural networks by preventing co-adaptation of feature detectors. The basic idea of dropout is random zeroes of some of the neurons of the input tensor with probability p(we use 0.2) using samples from a Bernoulli distribution. By this technique, some neurons causing the overfitting will be dropped during the training. In terms of weight decay, which is also named L2 Regularization, is a regularization technique applied to the weights of a neural network. We apply wd = 0.1. By applying the weight decay, it actually adds a penalty to the loss function to avoid overfitting. We believe that applying dropout and weight decay, will lead to an increase in the performance.
Actually, with these techniques, our accuracy of the test set increased from 0.9 to 0.94. There is no obvious difference between the shape of the two learning curves, but the validation curve comes closer to the training curve, in other words, the accuracy of the validation set gets improved, we avoid some overfitting. Overall, the improvement of both the test and validation accuracy turns out to be evidence that the regularization techniques make the model more generalizable.

4. (2.0 bonus points) Re-implement the 3-class classification feed-forward neural network from Part 1 by calculating and applying the gradients without a PyTorch optimizer. Do not use a bias term for this part of the assignment. Also, it may be easier to perform Part 4 with inline code (e.g., weight vectors, matrix algebra operations, and your own sigmoid() function).
Note: You can check your gradient calculations by comparing before/after update weight matrices against a parallel implementation using a PyTorch optimizer.

- We implemented the neural networks from scratch using NumPy. All the functions, matrix calculations, etc have been defined in the **classify_insurability_manual** function itself. The number of epochs that we ran was 30 as there was not much change in the accuracy beyond that point.
- Below are the results:



```
F1 Score:
Bad is 0.9306930693069307,
Neutral is 0.8877551020408162,
Good is 0.8349514563106797
```


Bad
Neutral
Good
True label
Bad
Neutral
Good
Predicted label