==========HW3==========

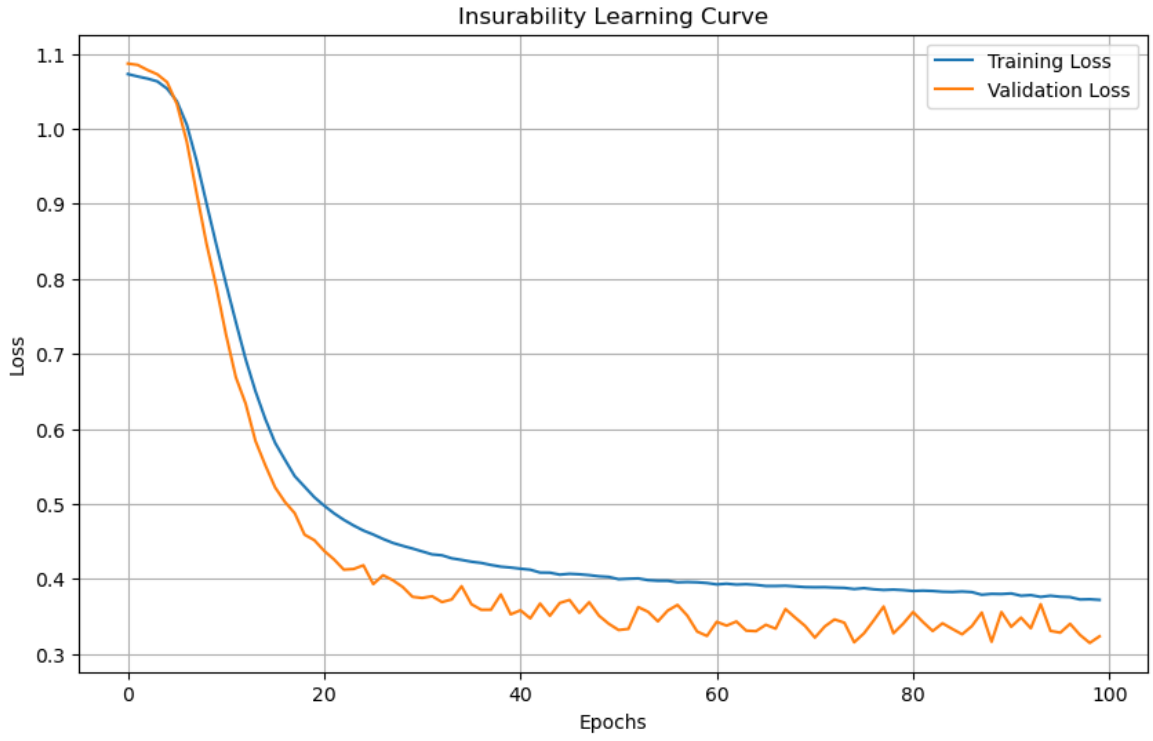
Chen Si
Dong Shu
Ding Zhang

Note: Please refer to the jupyter notebook for the following visualizations

1. (8.0 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 12-8). 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, (ii) final test results as a confusion matrix and F1 score, (iii) a description of why using a neural network on this dataset is a bad idea, and (iv) short discussion of the hyper-parameters that you selected and their impact.

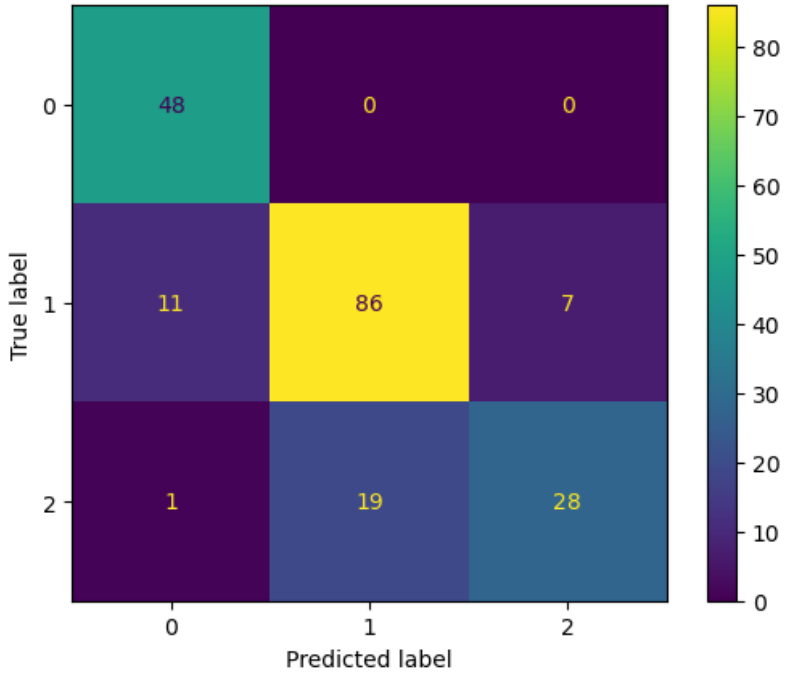
(i) learning curves for the training and validation datasets

Learning Curves (the results may change between different runs):



(ii) final test results as a confusion matrix and F1 score

Confusion Matrix (the results may change between different runs):



Classification Report:

	precision	recall	f1-score	support
0	0.80	1.00	0.89	48
1	0.82	0.83	0.82	104
2	0.80	0.58	0.67	48
accuracy			0.81	200
macro avg	0.81	0.80	0.80	200
weighted avg	0.81	0.81	0.80	200

(iii) a description of why using a neural network on this dataset is a bad idea

There may be several reasons:

- The imbalance of the dataset itself may partially contribute to model bias. As we can tell from the confusion matrix, the number of class “1” is more than twice more than the number of class “0” and “2”. This might lead to model bias.
- The dimensionality of the dataset is also quite few for neural networks. We only have three features in the dataset, which lack significant complexity for the neural network to generalize the prediction results.
- The given FFNN structure might be too simple. We only have two hidden layers, which might not be enough to capture enough complexity of the relationship between features and labels.
- Overfitting is also a possible reason, despite the simplicity of the model, it might still remember the training set and could not generalize to unseen testset data.

(iv) short discussion of the hyper-parameters that you selected and their impact

We have chosen and experimented several hyper-parameters: we have tried three different learning rates (lr = 0.001, lr = 0.01, lr = 0.1); five weight decays (wd = 0, wd = 0.0001, wd = 0.001, wd = 0.01, and wd = 0.1); as well as four number of epochs (50, 100, 300, 600). After testing out different combinations of hyper-parameters, the following set of parameters resulted the best results:

- Learning Rate (lr=0.01):
    - Learning rate controls the step size at which the neural network updates its weights during training. A higher learning rate can lead to faster convergence, but it may risk overshooting the optimal weights and causing the model to diverge. A lower learning rate might lead to more stable training but can be slower to converge.
    - In our case, lr=0.01 is a moderate learning rate choice. It strikes a balance between convergence speed and stability, and results in the best performance metrics.
- Batch Size (batch_size=1):
    - Batch size determines how many data samples are used in each forward and backward pass during training. Smaller batch sizes can provide a more stochastic gradient descent, which can introduce noise but might help escape local minima. Larger batch sizes can offer more stable updates but may require more memory.
    - As required, we set our batch size to 1.
- Number of Epochs (num_epochs=100):
    - The number of epochs defines how many times I want my model to iterate over the entire training dataset. Too few epochs may result in underfitting, while too many epochs can lead to overfitting.

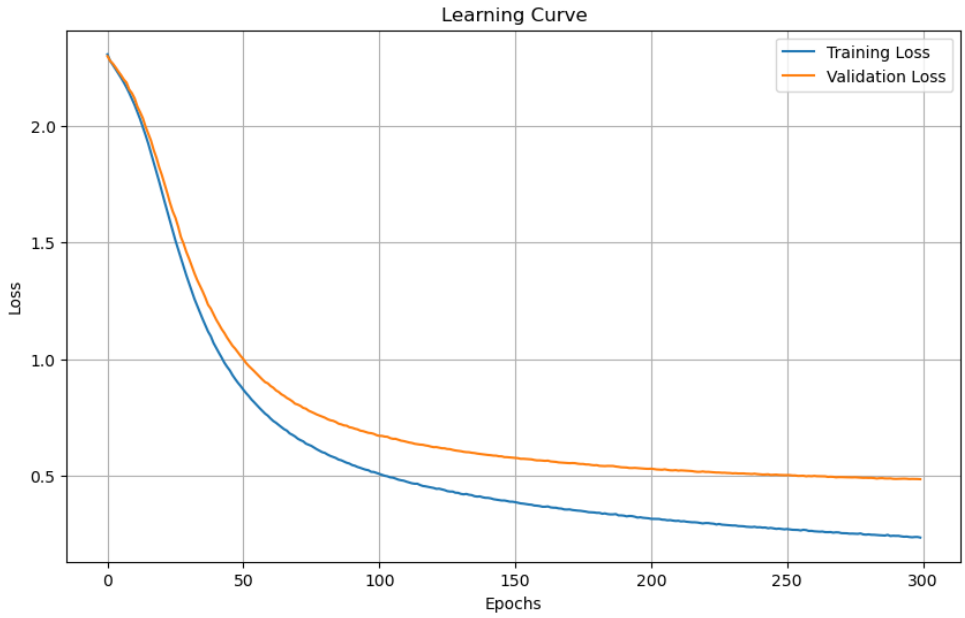
    - After testing out the four different numbers of epochs, we ended up with num_epochs = 100, as it performs the best compared with other choices.
- Weight Decay (wd = 0)
    - Weight decay is a regularization technique. It is similar to the L2 regularization, where we add a small penalty (usually L2 norm) of the weights to the loss function.
    - This will prevent our model from overfitting. It also keeps weight small and avoids the chance of exploding gradients.
    - In our case, we choose weight decay to be 0. The reason is that our feed forward neural network as well as the dataset are simple enough that there is a low chance of encountering exploding gradients. The weights would not grow out of control, and thus adding a weight decay would affect the performance of the model.

2. (8.0 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.

In this problem on categorizing the MNIST dataset from HW2, we have applied the same feed forward neural network defined in problem 1, with the exception of different numbers of input layers and hidden layers. Since the MNIST dataset contains 2000 records of handwritten digits, with each record containing 784 pixels. Thus, the number of input layers should match the number of pixels, which is 784. In addition, we have set the number of hidden layers to be 128, instead of 2 from the previous problem. During our experiment, we found that too few hidden layers would result in a final output of “nan”. This is partly due to the fact that only two hidden layers cannot have the ability to handle the dimensionality of the dataset. Thus, we have increased the number of layers to 128 for better capacity. We still applied the SGD optimizer and cross-entropy loss function.

Our predicted results using neural networks have relatively the same performance compared to the KNN classifier, but significantly outperform the K-Means classifier. Unlike the first problem, the MNIST dataset consists of complex data with high dimensionality. Neural networks are able to handle non-linear relationships better.

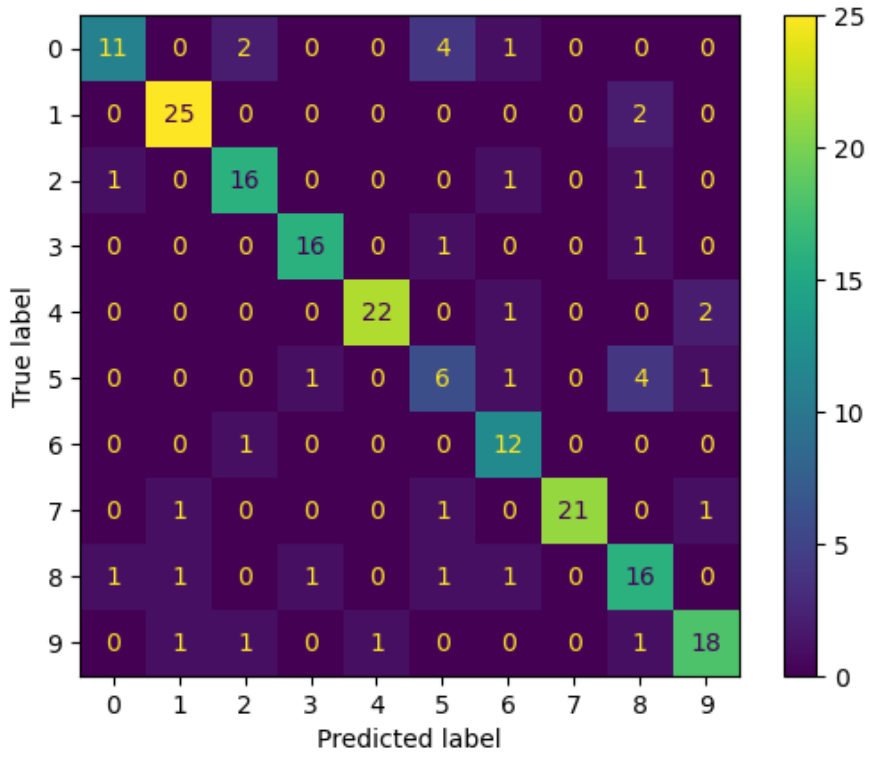
Learning Curve:



Classification Report:

	precision	recall	f1-score	support
0	0.85	0.61	0.71	18
1	0.89	0.93	0.91	27
2	0.80	0.84	0.82	19
3	0.89	0.89	0.89	18
4	0.96	0.88	0.92	25
5	0.46	0.46	0.46	13
6	0.71	0.92	0.80	13
7	1.00	0.88	0.93	24
8	0.64	0.76	0.70	21
9	0.82	0.82	0.82	22
accuracy			0.81	200
macro avg	0.80	0.80	0.80	200
weighted avg	0.83	0.81	0.82	200

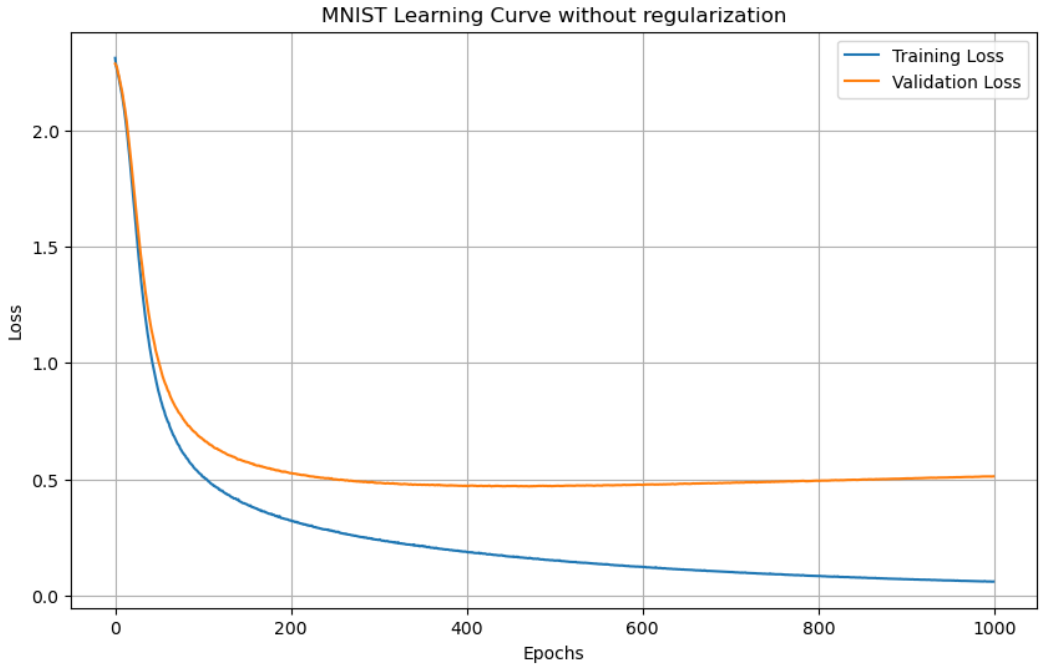
Confusion matrix:

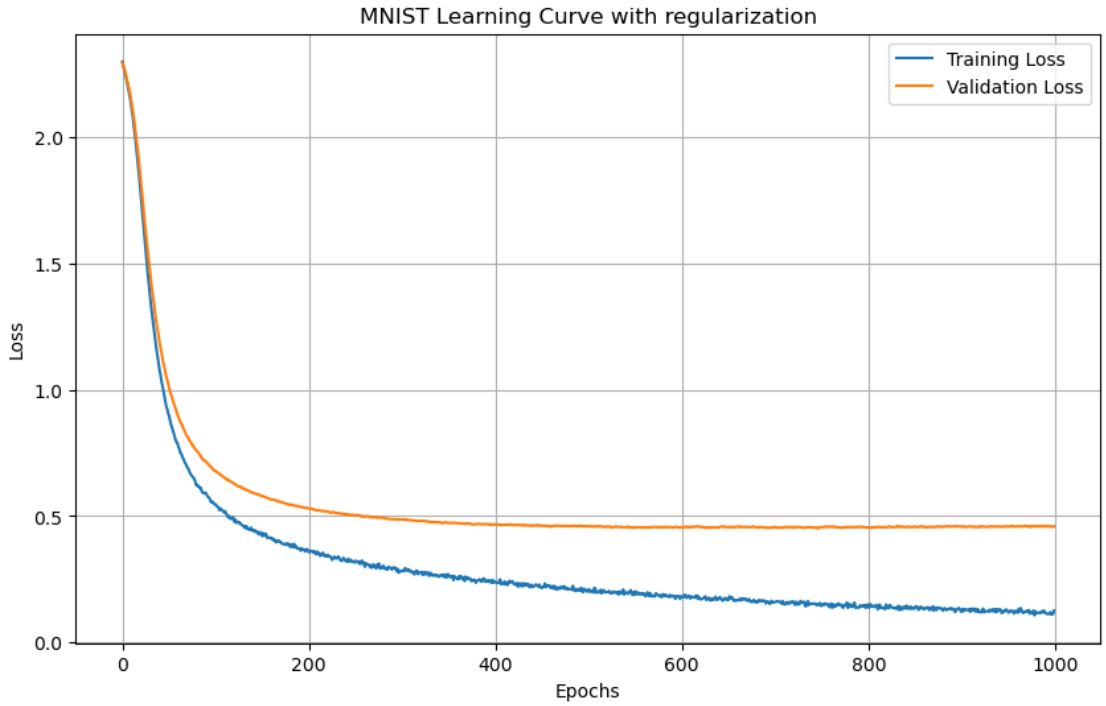


3. (4.0 points) Add a regularizer of your choice to the 10-class classifier that you implemented for Part 2. Describe your regularization, the motivation for your choice and analyze the impact of the performance of the classifier. This analysis should include a comparison of learning curves and performance.

Among different regularization techniques, we choose to adopt the **Dropout** regularization. We have tested different regularization tools such as L1 and L2 regularization, Elastic Net (combination of L1 and L2 regularization), Dropout, and Early Stopping. Dropout outperforms the other regularization tools. Since we are dealing with a relatively small dataset, with only 2000 records, the parameters for L1 and L2 are hard to tune. If we have the parameters being too large, the model would likely underfit and perform poorly; likewise, the model would overfit the data and memorize the training set when the parameters are too small. Early stopping would also result in underfitting if stopped too early. Thus, Dropout would be an ideal choice in our case.

Illustrated below, the two diagrams show the learning curve of our neural network models, the former being without regularization and the latter with regularization. For the purpose of illustrating the overfitting effect, we purposely increase the number of epochs from 300 to 1000. As you can see from the first graph, the validation loss starts to increase after Epoch 600, indicating the sign of overfitting the training set. The second graph shows the learning curve after applying Dropout. Validation loss remained constant after Epoch 400, and eventually did not pass 0.5 compared to the previous one.
The classification reports also indicate that a FFNN with regularization performs better than a non-regularized one (results highlighted in red).





Without Regularization:

	precision	recall	f1-score	support
0	0.87	0.72	0.79	18
1	0.96	0.93	0.94	27
2	0.73	0.84	0.78	19
3	0.81	0.72	0.76	18
4	0.96	0.88	0.92	25
5	0.46	0.46	0.46	13
6	0.65	0.85	0.73	13
7	1.00	0.88	0.93	24
8	0.60	0.71	0.65	21
9	0.82	0.82	0.82	22
accuracy			0.80	200
macro avg	0.79	0.78	0.78	200
weighted avg	**0.81**	**0.80**	**0.80**	200

With Regularization (Dropout):

	precision	recall	f1-score	support

0	0.86	0.67	0.75	18
1	0.96	0.93	0.94	27
2	0.73	0.84	0.78	19
3	0.87	0.72	0.79	18
4	0.95	0.84	0.89	25
5	0.47	0.54	0.50	13
6	0.69	0.85	0.76	13
7	1.00	0.92	0.96	24
8	0.67	0.76	0.71	21
9	0.79	0.86	0.83	22
accuracy			0.81	200
macro avg	0.80	0.79	0.79	200
weighted avg	**0.83**	**0.81**	**0.81**	200