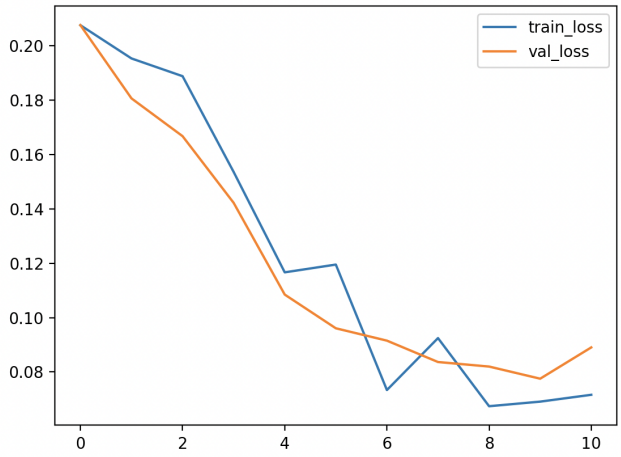
(8.0points)Implement a simple feed-forward neural network inPyTorch to make classifications on the “insurability” dataset. The architecture should be the same as the one covered in class (see Slide 12-8). You can implement the neural network as a class or by using in-line code. You should use the SGD 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
Actual on the Y axis and predicted on the X axis
[[22 24 2]
[1 89 14]
[0 0 48]]
f1:
[0.61971831 0.8202765 0.85714286]

all info:

	precision	recall	f1-score	support
0	0.96	0.46	0.62	48
1	0.79	0.86	0.82	104
2	0.75	1.00	0.86	48
accuracy			0.80	200
macro avg	0.83	0.77	0.77	200
weighted avg	0.82	0.80	0.78	200

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

- Morally should not use technology make opinionated statements about people's lives, is unethical and leads to a lot of bias and discrimination

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

- Tried learning rate of 1e-3. This caused our neural network to get stuck in a bad local minimum. Our network predicted only Neutral for every observation
- Learning rate of 1e-2 and momentum of 0.9. Adam uses a momentum feature, which helps keep weights moving quicker if they continually point in one direction. This was enough to get us out of the bad local maximum and got us similar results.
- Learning rate of 1e-1. This is what we eventually settled on. It got us out of the bad local maximum and got us the good results seen in the graph.

2. (8.0points)Implement neuralnetwork 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.

(i) Describe your design choices, provide a short rationale for each and explain how this network differs from the one you implemented for Part 1.

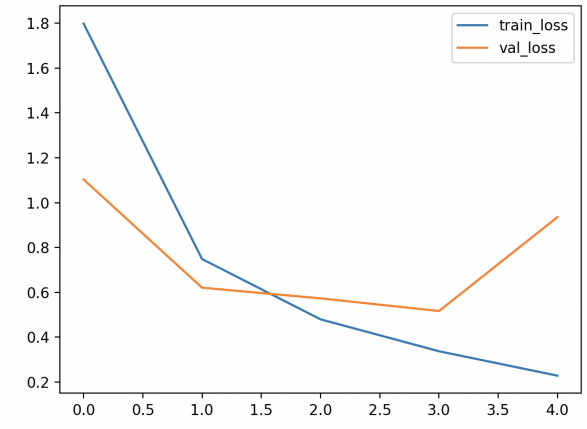
This network differs in

- Structure: it has 2 hidden layers, instead of 1 like in #1.
- Activation function: uses ReLU instead of sigmoid for better efficiency, avoiding vanishing gradients problem
- Loss function: Cross Entropy Loss instead of MSE or NLL,
    - does not do one-hot encoding for the labels nor has a manual softmax function after the output of the network as we use CrossEntropyLoss(), which has a built in softmax function that takes in logits instead of probabilities & label indices instead of one-hot encodings.
- Optimizer: adam instead of SGD as it dynamic adjusts learning rates based on individual weights, leading to faster model convergence, rather than SGD's single learning rate

(ii) Compare your results to Homework #2 and analyze why your results may be different.

From the confusion matrix and accuracy scores below, we can see that our results are fairly similar to the KNN classifier from Homework #2. However, our KNN model from Homework #2 had a slightly higher accuracy (90.5% vs 86%) than our neural network does. This could be due to errors in our model architecture, our loss function, our optimizer, or our hyperparameters, and with further tuning we potentially could get it to 90%+. This neural network, however, was a lot better than the K-means classifier (64% accuracy) which is a good sign.

(iii) In addition, please provide the learning curves and confusion matrix as described in Part 1.



confusion matrix:

```
[[17  0  0  0  0  1  0  0  0  0]
 [ 0 26  0  0  0  1  0  0  0  0]
 [ 1  1 16  0  0  0  0  1  0  0]
 [ 0  0  2 16  0  0  0  0  0  0]
 [ 0  0  0  0 24  0  0  0  0  1]
 [ 0  0  0  0  0  9  0  0  3  1]
 [ 0  0  0  0  0  0 13  0  0  0]
 [ 1  0  0  0  1  0  0 22  0  0]
 [ 1  2  1  1  0  3  0  0 12  1]
 [ 2  1  0  0  2  0  0  0  0 17]]
```

f1:

```
[0.85      0.9122807  0.84210526 0.91428571 0.92307692 0.66666667
 1.        0.93617021 0.66666667 0.80952381]
```

all info:

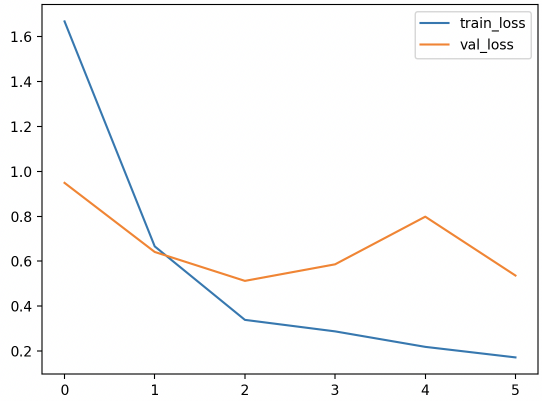
	precision	recall	f1-score	support
0	0.77	0.94	0.85	18
1	0.87	0.96	0.91	27
2	0.84	0.84	0.84	19
3	0.94	0.89	0.91	18
4	0.89	0.96	0.92	25
5	0.64	0.69	0.67	13
6	1.00	1.00	1.00	13
7	0.96	0.92	0.94	24
8	0.80	0.57	0.67	21
9	0.85	0.77	0.81	22
accuracy			0.86	200
macro avg	0.86	0.86	0.85	200
weighted avg	0.86	0.86	0.86	200

3. (4.0points)Add a regularizer of your choice to the 10-class classifier that you implemented for Part 2.

(i) Describe your regularization, the motivation for your choice and analyze the impact of the performance of the classifier.

Used the Adam optimizer's built-in parameter weight_decay as a regularization method as it also adapts the weights to prevent overfitting, but focuses on reducing weight magnitude overtime rather than adding penalty terms to the loss function. It's good to note that if we used SGD as the optimizer, the weight_decay would be functionally the same as L2 regularization.

(ii) This analysis should include a comparison of learning curves and performance.



confusion matrix:

```
[[14  0  1  0  0  2  1  0  0  0]
 [ 0 26  0  0  0  0  0  0  1  0]
 [ 1  1 17  0  0  0  0  0  0  0]
 [ 0  0  1 16  0  0  0  0  1  0]
 [ 0  0  0  0 24  0  0  0  0  1]
 [ 0  0  0  0  0  7  0  0  5  1]
 [ 0  0  0  0  0  0 13  0  0  0]
 [ 1  0  1  0  1  0  0 21  0  0]
 [ 1  1  0  1  0  1  0  0 17  0]
 [ 1  1  0  0  3  0  1  0  0 16]]
```

f1:

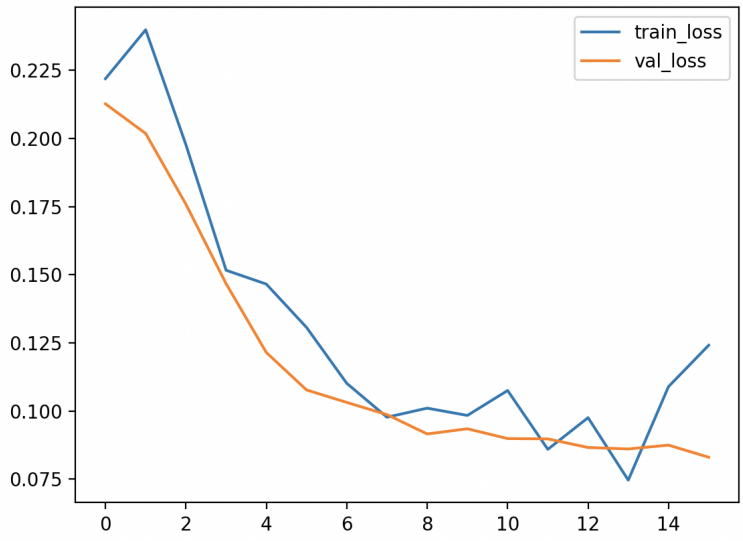
```
[0.77777778 0.92857143 0.87179487 0.91428571 0.90566038 0.60869565
 0.92857143 0.93333333 0.75555556 0.8       ]
```

all info:

	precision	recall	f1-score	support
0	0.78	0.78	0.78	18
1	0.90	0.96	0.93	27
2	0.85	0.89	0.87	19
3	0.94	0.89	0.91	18

4	0.86	0.96	0.91	25
5	0.70	0.54	0.61	13
6	0.87	1.00	0.93	13
7	1.00	0.88	0.93	24
8	0.71	0.81	0.76	21
9	0.89	0.73	0.80	22
accuracy			0.85	200
macro avg	0.85	0.84	0.84	200
weighted avg	0.86	0.85	0.85	200

4. (1.0 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.



confusion matrix:
[[27 19 2]
[5 85 14]
[0 0 48]]
f1:
[0.675 0.81730769 0.85714286]
all info:

	precision	recall	f1-score	support
0	0.84	0.56	0.68	48

1	0.82	0.82	0.82	104
2	0.75	1.00	0.86	48
accuracy			0.80	200
macro avg	0.80	0.79	0.78	200
weighted avg	0.81	0.80	0.79	200

The scores look very similar to that of problem 1. We used the same learning rate of 1e-1 to escape the local minimum, and upon plotting our loss over time, we saw very similar results. However, the initial SGD network took less time to hit the bottom plateau on the validation loss, which makes sense as this network has no bias term and thus fewer parameters. They still ended up with very similar accuracies, however.