CMPE 597 Sp. Tp. Deep Learning: Assignment 1

Group 10: Tarik Can Ozden – Ahmet Firat Gamsiz

Source code link: https://drive.google.com/file/d/1lozUKABoqCYzu2O6z8KZpXd8Gbxn0x7j/view?usp=sharing

I. Classification with Cross-Entropy

1. Implementation from Scratch

a) Design a multi-layer perceptron for this task. Please explain how you chose the depth and width of the network and activation functions.

- I selected ReLU as the activation function for the hidden layers due to its non-saturating nature, which helps with the vanishing gradients problem and its empirical success in deep networks. And for the last layer, the output layer uses the Softmax activation, which is standard for multi-class classification tasks.
- I fixed the width of all hidden layers at 128 neurons. I chose this value as a compromise between the high-dimensional input space and the lower-dimensional output, providing enough capacity to model complex patterns without overfitting.
- Rather than varying depth and width simultaneously, I held the width constant and searched different depths to evaluate the network's performance. With a depth of 1 hidden layer, the model achieved a validation accuracy of 75.8% (loss: 0.6964). Increasing the depth to 2 layers improved the accuracy to 83.85% (loss: 0.5105), displaying better abstraction. However, with three hidden layers, the accuracy slightly decreased to 83.25% (loss: 0.5288), suggesting potential overfitting or diminishing returns in representational benefit at that depth, given the dataset size. However, the PyTorch implementation performed best with three hidden layers, so I chose 3 for both implementations.

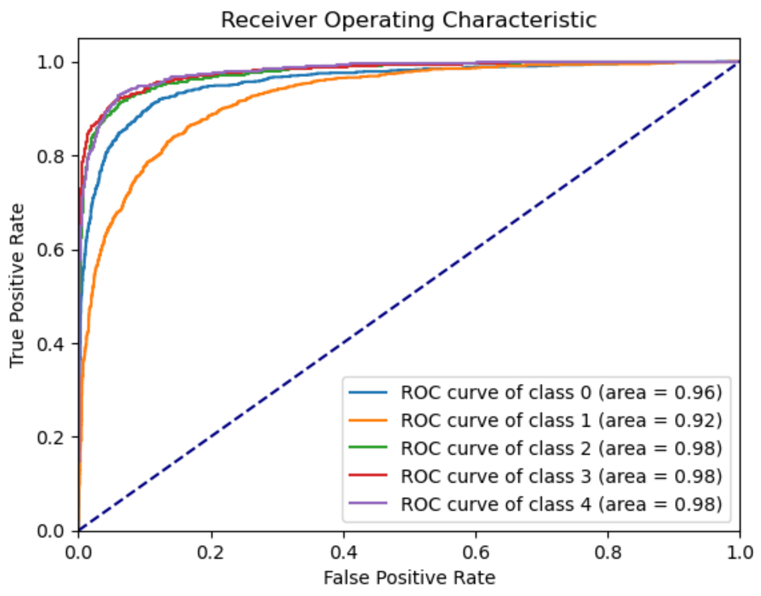
b) Implement stochastic gradient descent with momentum. You may set the momentum value to 0.9. Please feel free to tune it if this value does not work for you.

- The momentum value works well, and I achieved similar accuracy scores with PyTorch.

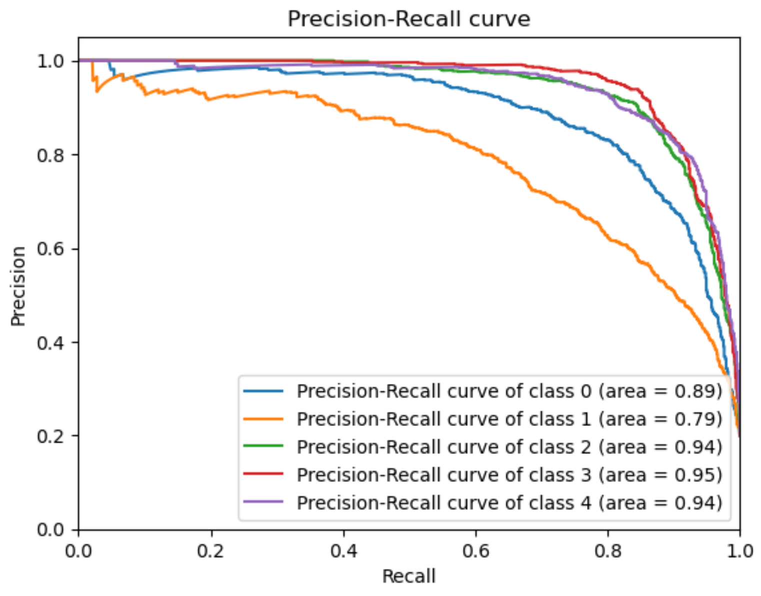
c) Evaluate the performance of your network on the test set provided to you. You may use classification accuracy, precision, recall, the area under the ROC curve, and the area under precision-recall curve metric implementations of scikit-learn.

- I use the model weights at epoch 42 with a validation loss of 0.5288 and an accuracy of 0.8325 by selecting the best validation loss.

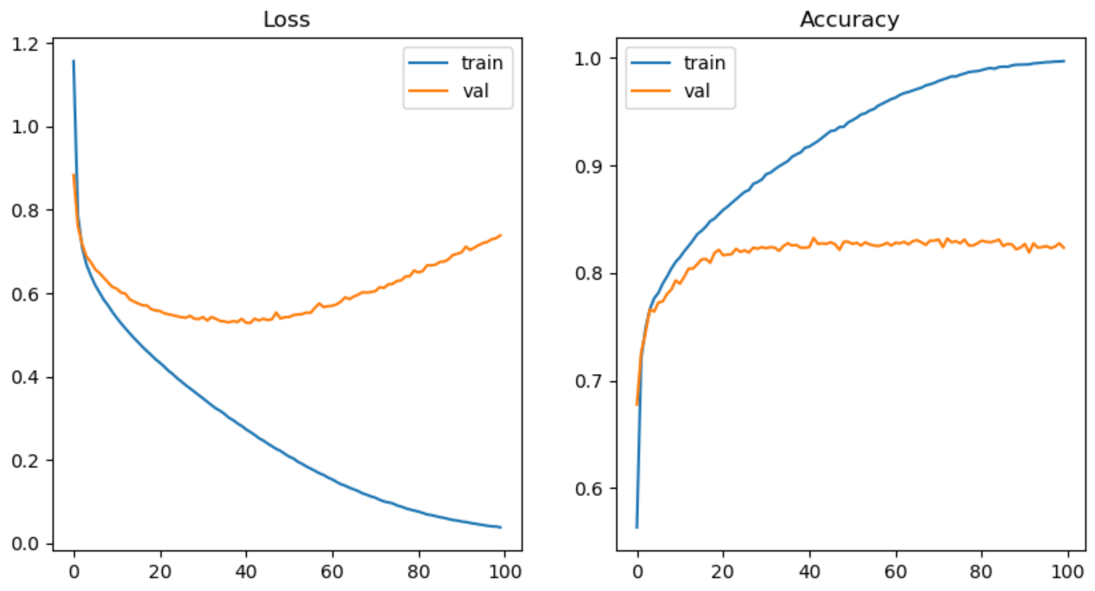
- Classification accuracy: 0.8242
- Precision: 0.8238
- Recall: 0.8241
- The area under the ROC curve:



- The area under Precision-Recall curve:



d) Please plot the changes in loss, training, and validation accuracies during training. You may spare a random subset from the training set for validation.



e) Clearly explain how you chose the hyperparameters.

- I did a grid search on a small set of hyperparameters. The space was for learning rate = [0.001, 0.01, 0.1], for momentum = [0.8, 0.9], and for batch size = [32, 64, 128]. I didn't search for epoch count since I selected the best-performing epoch according to validation loss over 100 epochs. I never observed the model is underfit for 100 epochs. The best-performing hyperparameter set was (0.01, 0.9, 64). I set the learning rate to 0.001 for the PyTorch model because it was converging and overfitting extremely fast with 0.01.

f) Discuss what you tried to improve your network's performance.

- Other than the discussions from previous questions, I implemented Kaiming initialization, which has been shown to work better with ReLU activation to prevent vanishing and exploding gradient problems.
- I created a validation split from the training set to prevent overfitting and track model generalization. I saved the model weights corresponding to the best validation loss, allowing me to reload the most generalizable model later rather than just the final one.
- I implemented stability techniques to ensure robustness during training and avoid numerical issues. In the Softmax layer, I subtracted the maximum value from the logits before exponentiating, which prevents numerical overflow. In the

CrossEntropyLoss, I clipped the predicted probabilities to a minimum of 1e-15 to avoid log(0) scenarios resulting in undefined loss values or NaNs. These small adjustments make the loss function stable even in edge cases.

2. **Implementation with Deep Learning Libraries**

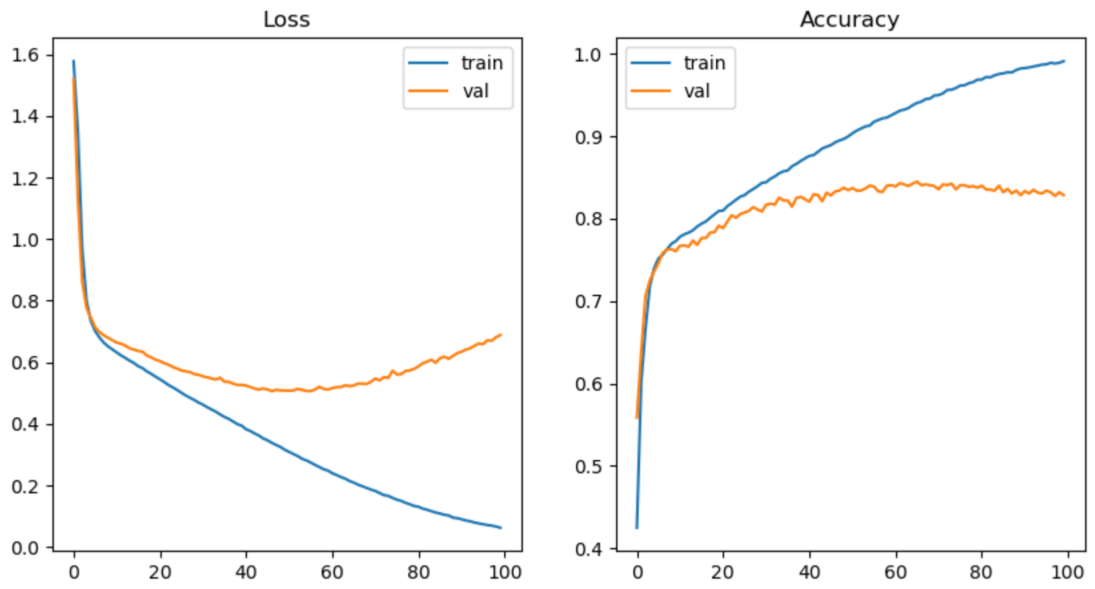
a. Now implement the same architecture and train your model with the same optimizer and the hyperparameters in part 1 using a deep learning library.

b. Compare the gradients you obtain using your backpropagation and the library's implementation for the first iteration.

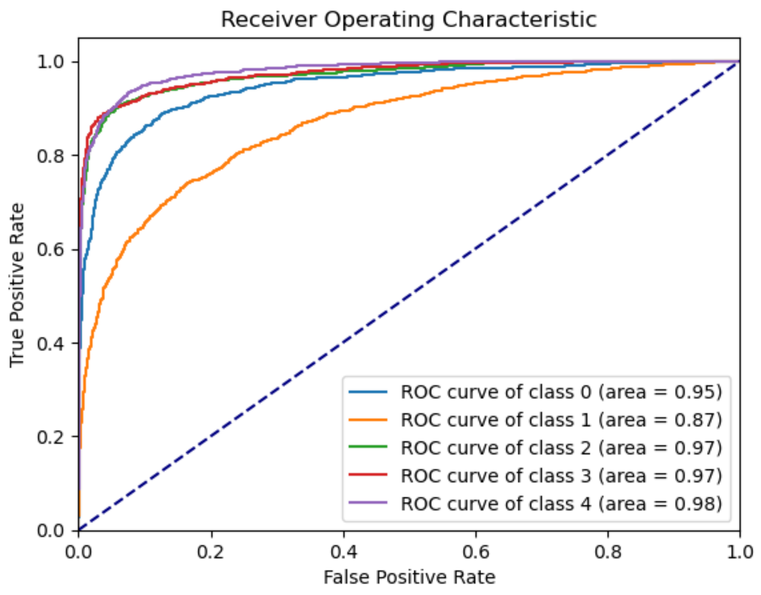
- When we compare the gradients, we observe that they are close in magnitude even though they are not exactly the same. We hypothesize there could be a few reasons for this difference. Even though we use np.float32 to match PyTorch's precision, there might be some small differences due to how floating-point operations are handled internally, which can lead to different rounding errors. PyTorch implements more advanced numerical stability tricks, which can lead to more numerical differences. Moreover, PyTorch combines softmax and cross-entropy loss while calculating gradients for simpler and more accurate calculations. We decided not to do this to keep our layers and functions modular. Finally, PyTorch's autograd engine is highly optimized, and mathematical operations, like summation and averaging, probably have more accuracy than the Numpy implementation.

c. Compare the test classification accuracies and loss plots for both implementations.

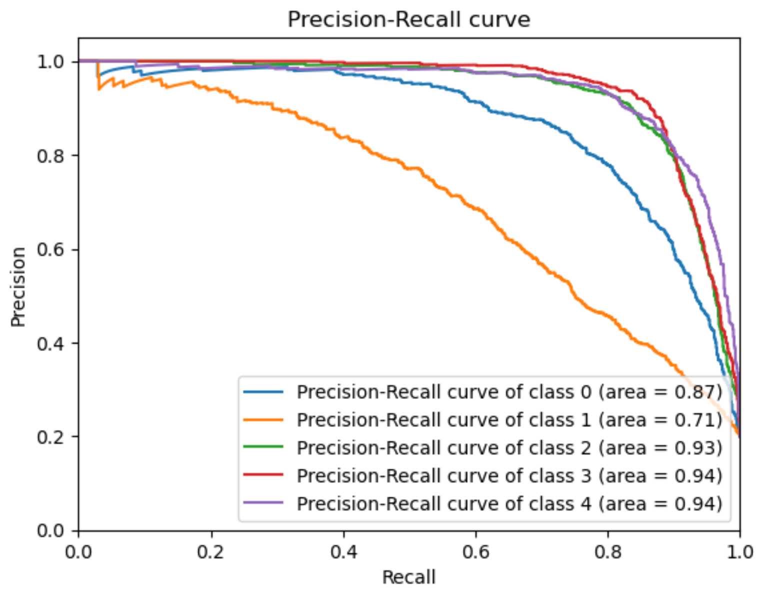
- By selecting the best validation loss, I use the model weights at epoch 47 with a validation loss of 0.5060 and an accuracy of 0.8330.
- Classification accuracy: 0.842
- Precision: 0.8415
- Recall: 0.842
- Loss curves:



- The area under the ROC curve:



- The area under Precision-Recall curve:



- The results are very close to our custom model, which shows that we performed a very close implementation of PyTorch's autograd.

II. Classification with Word Embeddings

1. Implementation from Scratch

a) Design two multi-layer perceptrons for this task. Please explain how you chose the depth and width of the networks and activation functions.

- I decided to have embeddings of size 32 for both images and word embeddings at the end. I have tried smaller values than 32: 8 and 16, but they didn't capture enough information and the accuracies were low, so I moved to 32 as the final output embedding size.
- For that purpose, I created an MLP for image embedding with 3 layers, with neuron sizes of 512, 512, 32. I chose ReLU as the activation function of hidden layers, and I chose not to use any activation function for the output layer as it outputs an embedding vector with no constraints. So, the pipeline for MLP image is 784 (input) → linear 512 → ReLU → linear 512 → ReLU → linear 32.
- I created an MLP with two layers, with neuron numbers 32 and 32, for word embeddings. I chose to use GloVE embedding with size 100, as it was an intermediate value. Again, the activation function of the output layer is ReLU, and there is no activation function for the output layer as the case is similar to the previous one. I chose to use only two layers, as I did not use this MLP for training, i.e., I did not update the parameters of this MLP. In contrast, in the training, I updated the parameters of only image NN so that the output of that model converges to the

outputs of this MLP, preventing zigzagging when updating the parameters of both and reducing calculation time significantly.

b) Implement stochastic gradient descent with momentum. You may set the momentum value to 0.9. Please feel free to tune it if this value does not work for you.

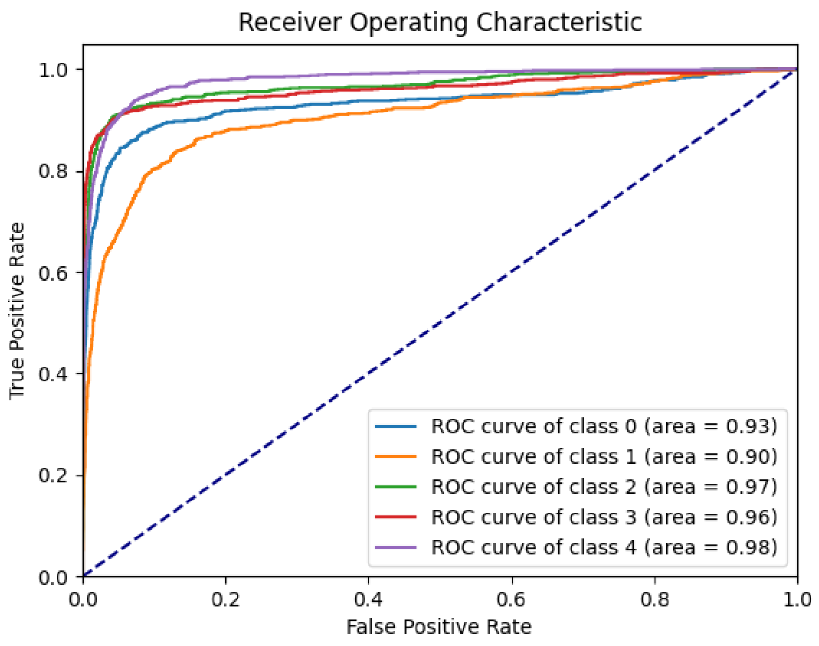
- The momentum value worked well, and I achieved similar accuracy scores with PyTorch, so I kept it as it was. I got the best score with a learning rate of 0.1 for SGD.

c) Please explain how you choose the loss function. Feel free to explore and add more than one penalty term if it helps.

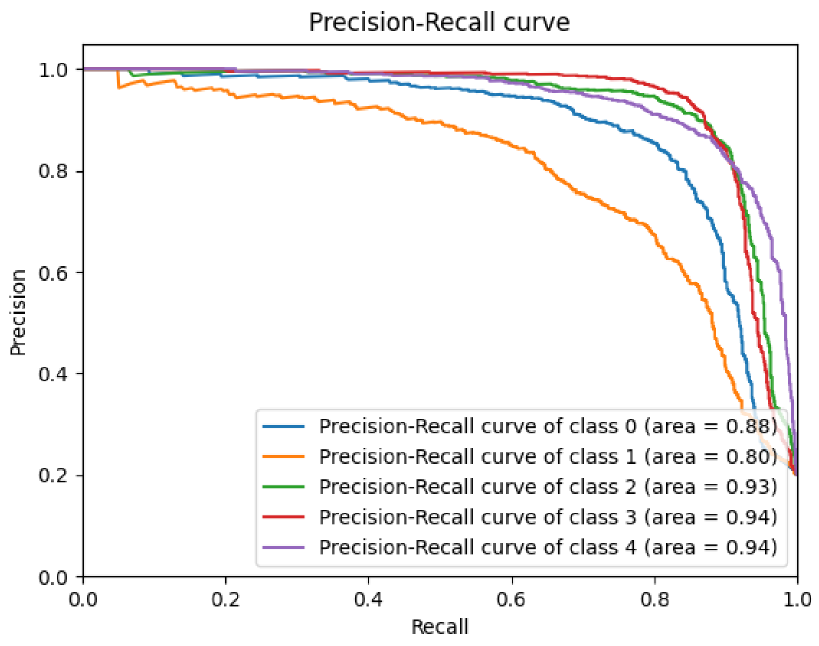
- After normalizing the embeddings, I chose the loss function to be cosine embedding loss, which is 1 – cos(emb1, emb2). To compare the similarities of embeddings, cosine similarity is widely used, especially in information retrieval, and I wanted to apply it here as well, which produced nice results.

d) Evaluate the performance of your network on the test set provided to you. You may use classification accuracy, precision, recall, the area under the ROC curve, and the area under the precision-recall curve metric implementations of scikit-learn.

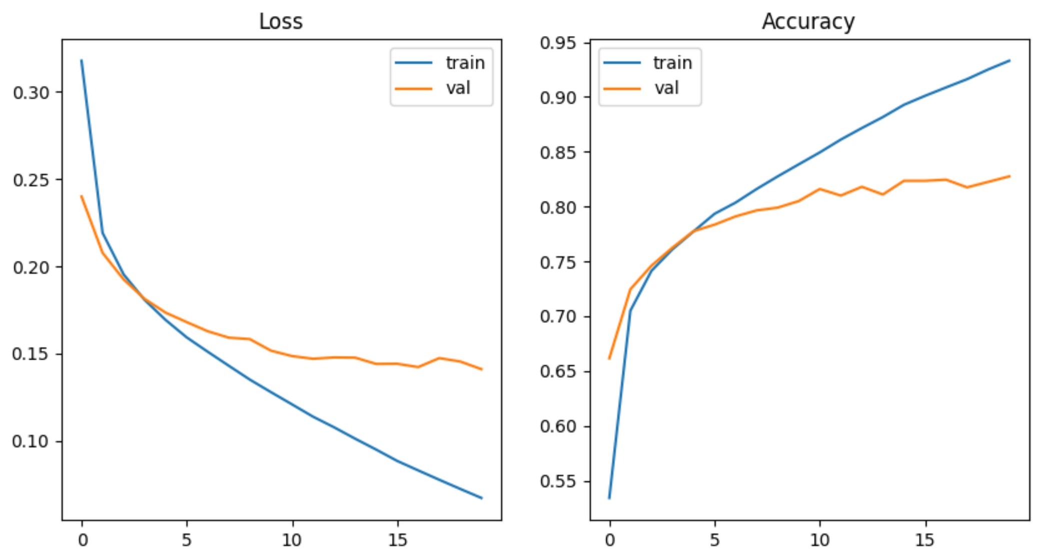
- I use the model weights at epoch 20 with a validation loss of 0.084 and a validation accuracy of 0.8195.
- Classification accuracy: 0.8174
- Precision: 0.8251
- Recall: 0.8173
- F1: 0.8193
- The area under the ROC curve:



- The area under Precision-Recall curve:



e) Please plot the changes in loss, training, and validation accuracies during training. If you have multiple loss terms, also plot them separately.



f) Clearly explain how you chose the hyperparameters.

- As I have explained in part a, I have chosen the final embedding size of 32 as it produced better results than 8 and 16, with more information inside the embeddings. To make the model as trainable as possible, I have set the hidden layers' neuron size to 512 for the image, and I set the word embedding hidden layer neuron size to 32 as I didn't train it. I have experimented with different learning rate and momentum values, and I obtained the best results with lr = 0.1 for momentum value of 0.9.

g) Could you train your network for this task? Please discuss the challenges you faced and how you overcame them.

- Yes, I could train my network successfully. The challenging part was deciding not to train the second MLP (for the word embeddings), which helped a lot with the training as they were zigzagging across each other, trying to converge to each other, and overstepping. I decided to remove the steps for the second MLP and saw fruitful training after that. The second challenging part was defining the backward propagation function. I had an error with the calculation, so I had to check math stackexchange forums for the correctness of the partial derivatives, and then I corrected my backpropagation function successfully.

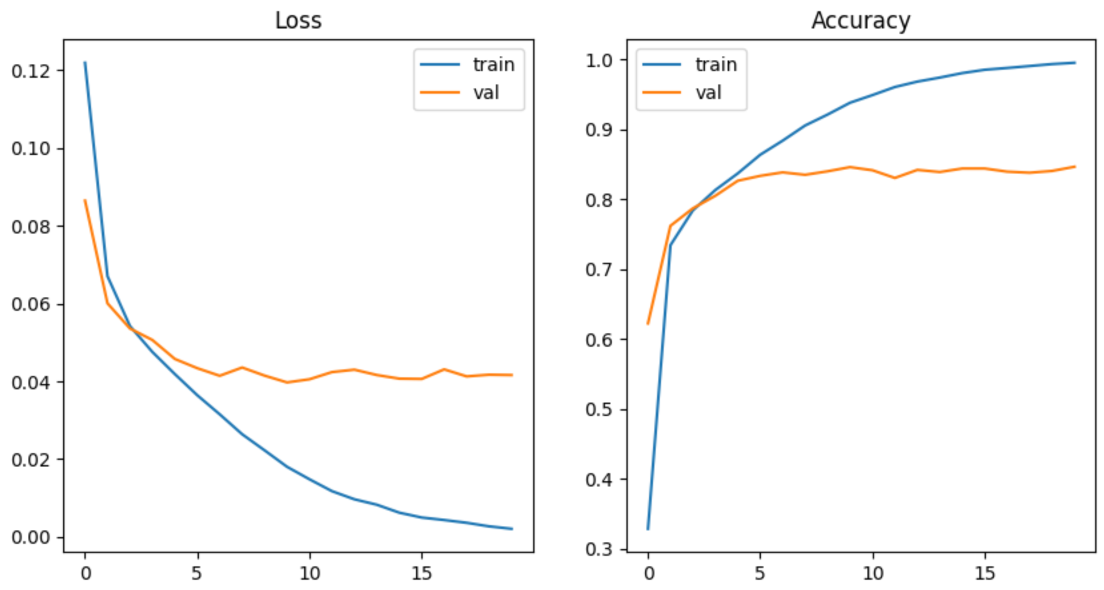
2. Implementation with Deep Learning Libraries

a) Now implement the same architecture and train your model with the same optimizer and the hyperparameters using a deep learning library.

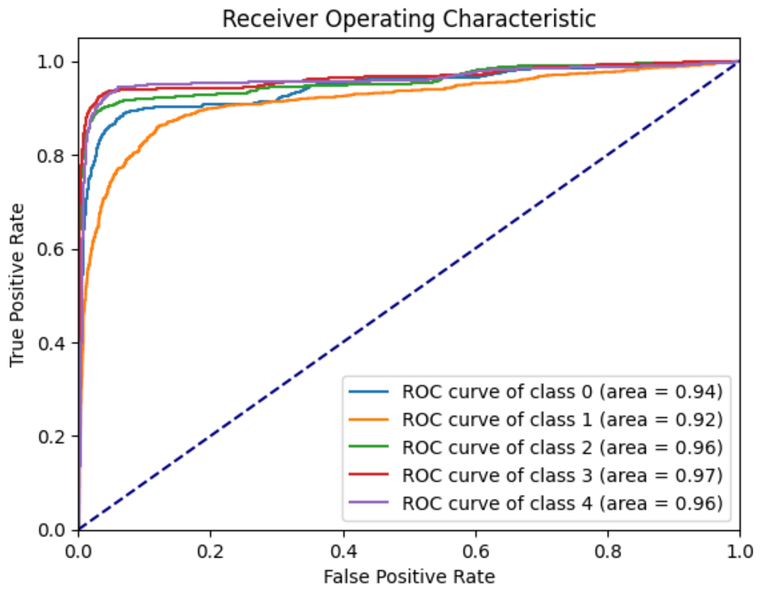
- I have successfully implemented the same architecture with PyTorch. The accuracies and loss plots are below.

b) Compare the test classification accuracies and loss plots for both implementations.

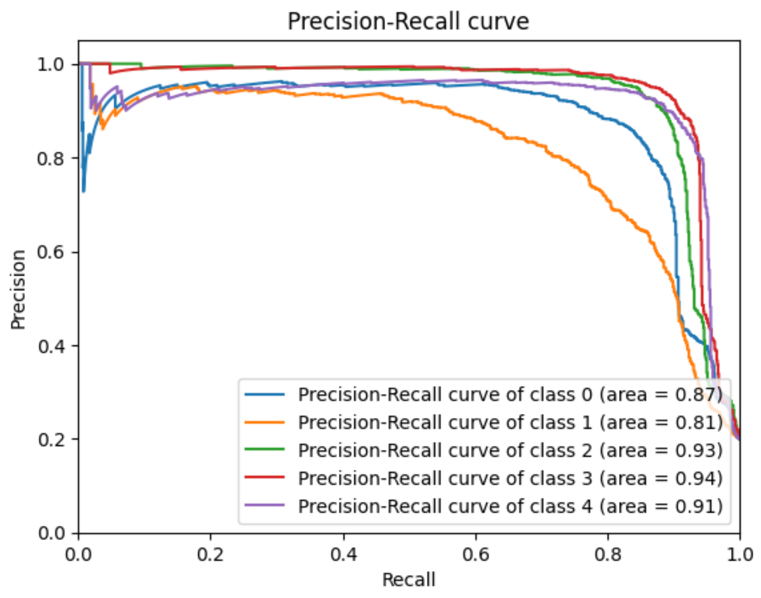
- Classification accuracy: 0.858
- Precision: 0.8582
- Recall: 0.858
- F1: 0.857
- Loss curves:



- The area under the ROC curve:



- The area under Precision-Recall curve:



- The results are close to our custom model, with similar convergence. The small difference in the final accuracies might be because of different initializations and better-optimized backward and step functions implemented in PyTorch. The difference in the loss function might be because of a different implementation. However, the loss curves, the ROC, and the precision-recall curves are similar, and the final results are close to each other.