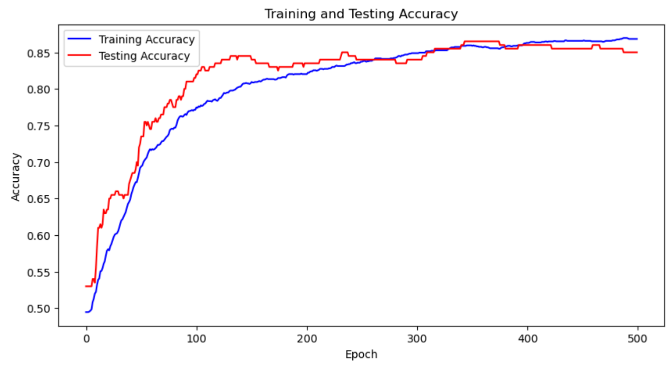
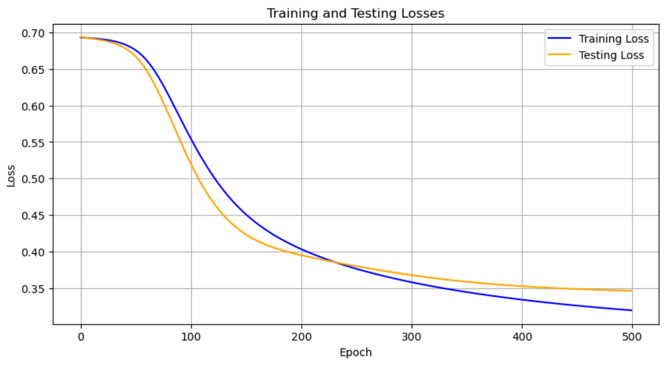
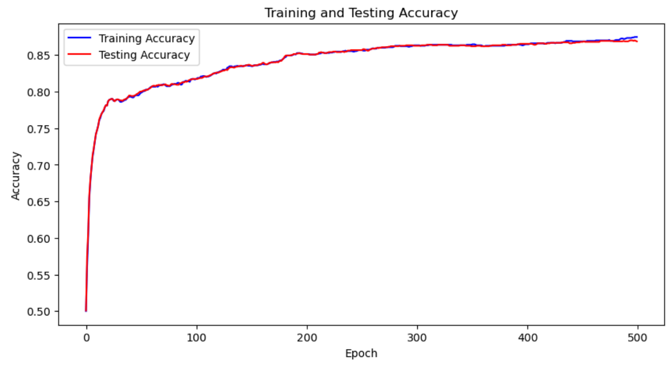
nn Training and testing Accuracy:



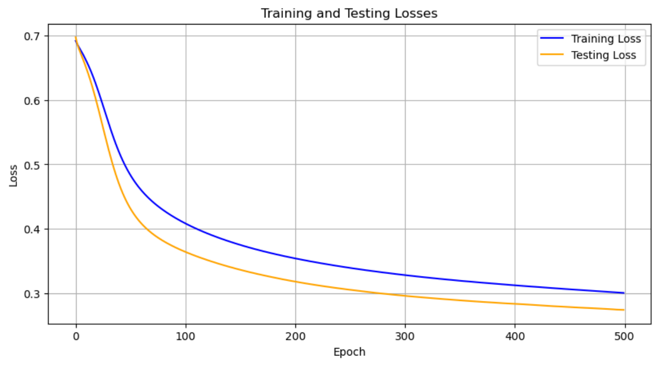
nn Training and Testing Loss:



Reference Training and Testing Accuracy:



Reference Training and Testing Loss:



The graph in the top left, representing 'Training and Testing Accuracy' from my implementation, shows convergence at around 85% accuracy for both training and testing, aligning with my expectations and similar as the Pytorch implementation at the bottom left. However, compared to the PyTorch implementation accuracy graph, that top left graph appears somewhat noisier. This discrepancy likely stems from the fact that I employed only the fundamental version of gradient descent in my custom implementation, whereas I utilized the SGD optimizer in the PyTorch implementation. The SGD optimizer is known to aid in stabilizing the training process, resulting in smoother accuracy trends.

In addition to these observations, it's worth noting that the PyTorch graph showcases an early and consistent convergence, indicative of the SGD optimizer's efficiency at escaping suboptimal local minima, a feature my basic gradient descent lacks. The difference in noise levels between the two graphs may also suggest that PyTorch's implementation benefits from optimized computations under the hood, which my implementation could potentially match by incorporating more sophisticated optimization techniques or fine-tuning hyperparameters.

In comparing the upper right loss graph from my implementation with the bottom right loss graph from the PyTorch implementation I found both of them have a steep initial decline in the training loss, indicating that early in training, the model rapidly improved its performance on the training data. This is probably because the initial parameter values were quite far from the optimal, and the model was able to quickly learn from the data. However, after this initial phase, the rate of loss reduction slows down significantly, probably indicating that the model starts to converge towards a local minimum.

Furthermore, the PyTorch implementation exhibits a less steep and gradual decline in training loss than my implementation. This consistent approach towards minimizing loss could be attributed to the optimizations within PyTorch's SGD. Also, the testing loss in the PyTorch graph closely tracks the training loss, maintaining a nearly parallel trajectory, which suggests that the model is generalizing well without overfitting. The slight divergence between the training and testing loss in the custom implementation could be an early sign of overfitting, as the model starts to learn patterns specific to the training data that do not generalize to unseen data.