Final Project Report: Face and Digit Classification

Names: Sri Akshara Kollu, Mia Wu

RUID: 205008034, 217001567

NetID: sk2342, cw984

Introduction

This project focuses on implementing and comparing the Perceptron Algorithm, a three-layer neural network with our own implementation of the forward and backward propagation algorithms, and a three-layer neural network using PyTorch. These models were tested on incrementally increasing training data from 10% to 100% on two datasets: digit classification and face classification in order to assess the performance of each model by its accuracy.

Design

Perceptron:

The Perceptron algorithm has a weight dictionary, where each key is a class label (0–9 for digits, 0–1 for faces), and the corresponding value is a Counter object that stores feature weights. For each training instance in the training data, we calculate a score for every label using the dot product between the training instance and the weights. If this predicted label does not match the true label, the algorithm updates the weights by adding the input vector to the true class's weight vector and subtracting it from the predicted incorrect class's weight vector.. The entire process is repeated for a fixed number of iterations, and for each percentage of training data, we average the results over five runs to calculate the mean accuracy and standard deviation.

Three-layer Neural Network - Our Implementation (nn_scratch):

The neural network from scratch uses three layers: an input layer, two hidden layers, and an output layer. We define weight matrices and bias vectors for each layer and initialize them using Xavier initialization. In the forward pass, we first multiply the input matrix with the weight matrix, add the bias vector, and apply the ReLU activation. This output becomes the input to the second hidden layer, which is processed similarly. The output layer computes a raw score which is passed through a numerically stable softmax function to produce class probabilities. The backward pass starts by computing the difference between predicted probabilities and one-hot encoded ground truth labels. This error is then back propagated through the network using the chain rule, computing gradients for each weight and bias. These gradients are multiplied by a learning rate and subtracted from the current weights to update the model. This implementation is fully vectorized using NumPy arrays and avoids loops in favor of efficient matrix operations. The training loop processes all data in batch mode and runs for 10 epochs per training percentage.

Three-layer Neural Network - Pytorch (torchNN):

The PyTorch model has two hidden layers and an input/output layer. It uses the nn.Module class to define the layers and their connections. The forward pass uses built-in ReLU and softmax layers from torch.nn.functional. Unlike our scratch model, PyTorch uses automatic differentiation to compute gradients, so we only need to define the loss function (cross-entropy) and call loss.backward() to compute gradients. We define a training loop that iterates over batches, performs forward and backward passes, and updates parameters each epoch using SGD.

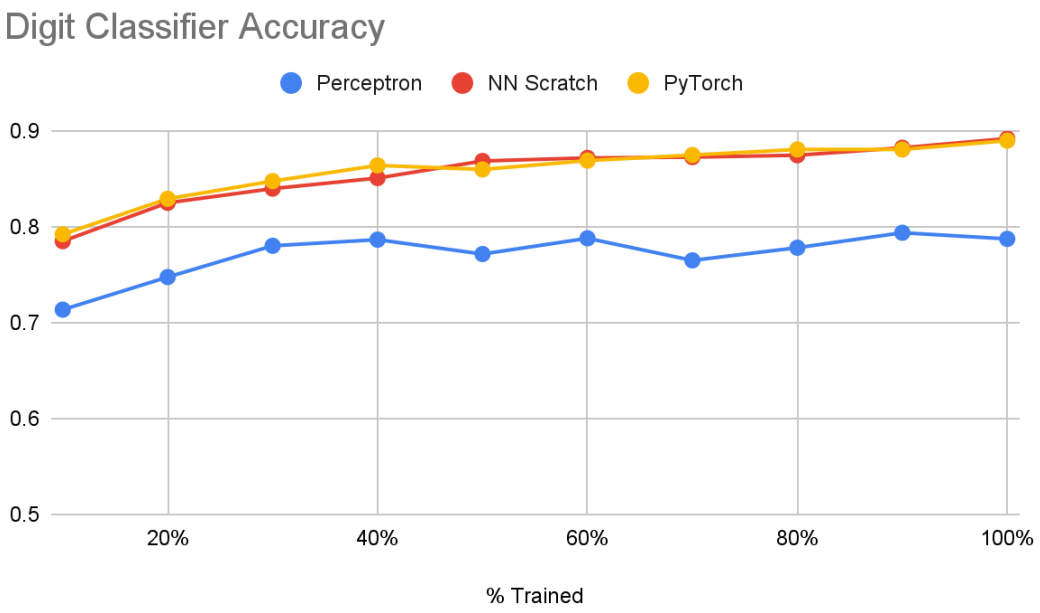
We track validation accuracy after each epoch and keep the best-performing model. The use of PyTorch removes the need to manually calculate gradients.

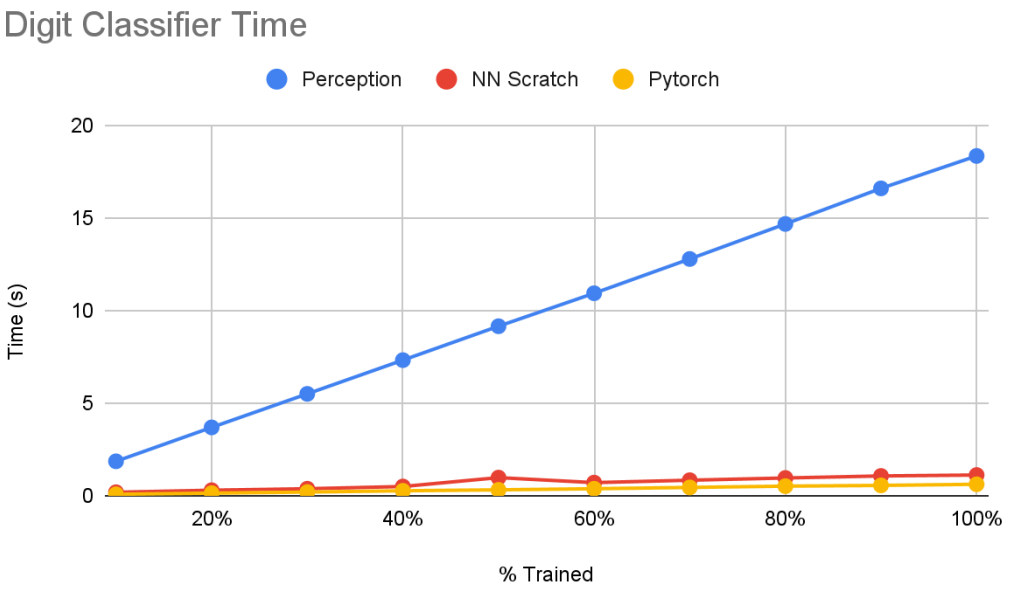
Results:

We evaluated each model's accuracy on test data across training sizes from 10% to 100%, repeating each test five times and reporting mean and standard deviation.

For digit classification:

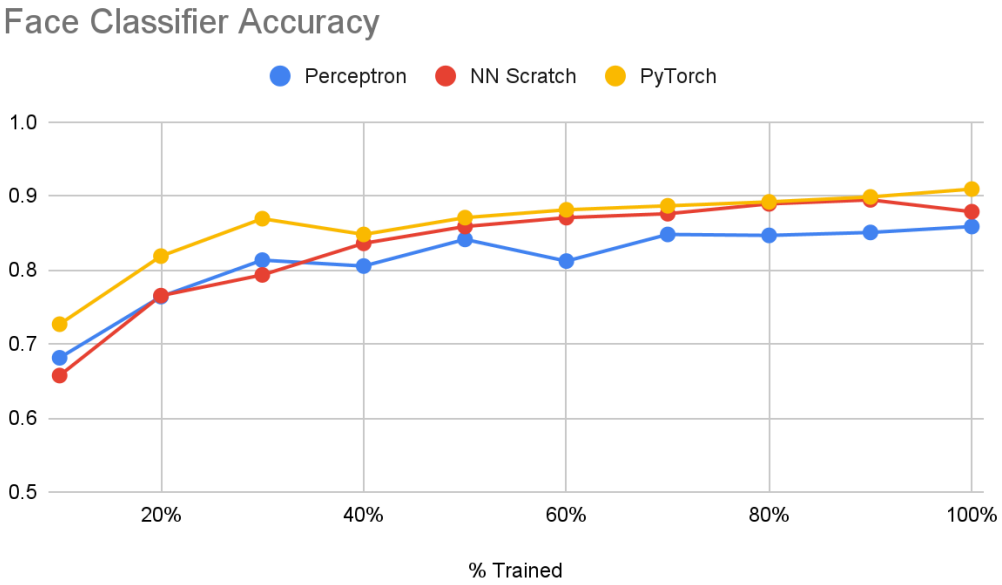
- Perceptron: Accuracy at 100% of training data – 79.36%, runtime ≈ 18.34 seconds
- NN_Scratch: Accuracy at 100% of training data – 89.18%, runtime ≈ 1.12 seconds
- PyTorch: Accuracy at 100% of training data – 88.96%, runtime ≈ 0.62 seconds

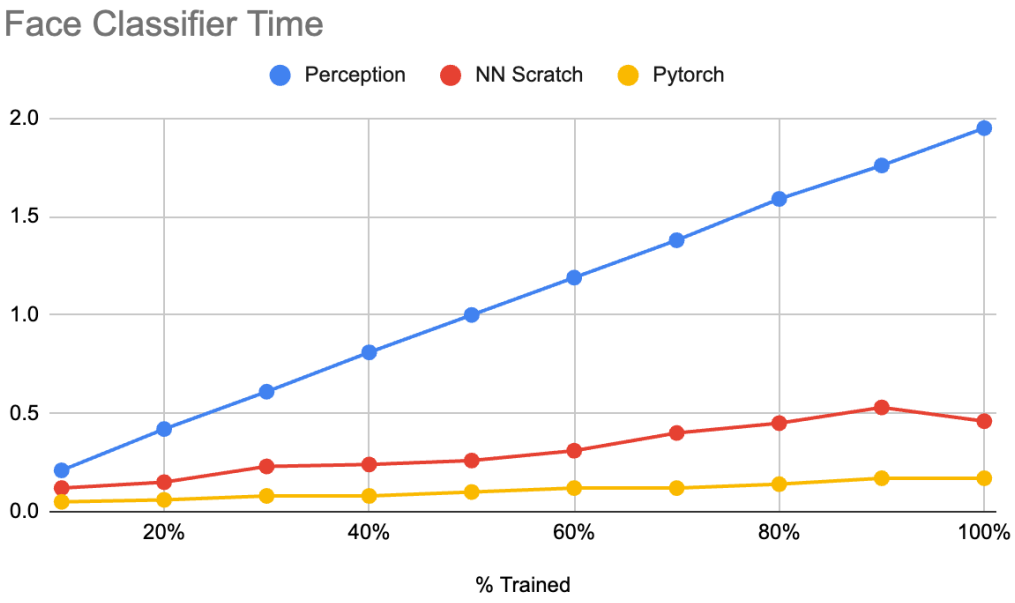




For face classification:

- Perceptron: Accuracy at 100% of training data – 85.87%, runtime ≈ 1.95 seconds
- NN_Scratch: Accuracy at 100% of training data – 89.47%, runtime ≈ 0.53 seconds
- PyTorch: Accuracy at 100% of training data – 90.93%, runtime ≈ 0.17 seconds





Discussion:

In evaluating both runtime and accuracy, trends emerged among the three algorithms as shown in the graphs above. The Perceptron algorithm was the slowest in both the digit and face classification. It updates weights one sample at a time and uses Python Counter objects instead of optimized NumPy arrays. These Counters loop over features individually and cannot use matrix operations, which adds significant time cost. The digit classification takes longer than face classification since it has 10 classes (0–9), so the model has to calculate and compare scores for more labels, while face classification only has 2 labels (0 and 1), which makes it faster. Our NN_Scratch model was significantly faster than the Perceptron algorithm because it uses vectorized NumPy operations to process entire batches at once, improving performance by avoiding slow Python loops. PyTorch was the fastest in the face classification task, likely due to its optimized backend and efficient memory handling. However, in digit classification, NN_Scratch was slightly faster, showing that custom vectorized implementations can sometimes

outperform general-purpose libraries on certain tasks. In terms of accuracy, both neural network models outperformed the Perceptron. At 100% training data, the Perceptron reached 79.36% accuracy for digit classification and 85.87% for face classification. NN_Scratch achieved 89.18% and 89.47%, respectively. PyTorch yielded the highest accuracy overall, with 88.96% on digits and 90.93% on faces.

Conclusion:

This project gave us the opportunity to have experience with machine learning models and allowed us to see how choices affect both performance and accuracy. Implementing the Perceptron helped us understand how simple linear classifiers operate by updating weights based on mistakes, one sample at a time. We saw how this approach can work well for binary classification tasks like face detection but struggles with more complex, multi-class problems like digit classification as seen by the longer runtime. Building a three-layer neural network from scratch deepened our understanding of concepts such as forward propagation, backpropagation, weight initialization. Using PyTorch allowed us to see how machine learning libraries help model training through built-in modules and optimizers like Stochastic Gradient Descent. This project introduced us to machine learning which is a growing field in the technology sector and something we're excited to learn more about in the future.