Face and Digit Classification - Project Report

Matthew Jojy, Terry Nguyen,
Veer Ashar, Shashank Madhavan

May 5, 11:55pm

1 Intro

In this project, we designed two classifiers: a perceptron and a three-layer neural network. We trained and tested our classifiers on a set of handwritten digit images and a set of face images with edges already detected.

We implemented the following:
(a) Perceptron algorithm
(b) Three-layer neural network (input layer, hidden layer 1, hidden layer 2, output) using our own implementation of the forward pass, back-propagation, and weight update
(c) Three-layer neural network (input layer, hidden layer 1, hidden layer 2, output) using the PyTorch library

We trained the algorithms on the training datasets with the associated labels in 10% increments of the data points available. Then, we compared the performance of the two algorithms using the testing datasets and output the time needed for training as a function of the number of data points used, and the prediction error (and standard deviation) as a function of the number of data points used.

2 Parsing the Datasets

Each image is represented in a text file, where symbols # and + denote foreground pixels (converted to 1), and spaces represent background pixels (converted to 0). Our custom data loader function reads in the image and label files. It first loads the labels into a NumPy array. Then, for each image, it reads the raw text lines corresponding to that image, flattens the 2D representation into a 1D vector, and converts the characters to the corresponding binary pixel values. The result is a 2D matrix where each row corresponds to a flattened image, which can then be used as input for the classifiers.

Figure 1: Example from digit and face datasets showing digit 3 and a valid face

3 Perceptron

The Perceptron algorithm is a linear classifier that assigns a weight to each input feature (in this case, we use pixels in images) to predict classes (digits or face/non-face labels). During training, the algorithm is provided with examples and corresponding labels, updating its weights iteratively over several passes (epochs) through the training dataset.

In our implementation, the Perceptron uses a weight vector $w^{(c)}$ for each class c. Each weight vector has the same dimension as the input, which is the number of pixels in a flattened image. For digit classification, $c \in \{0, 1, \ldots, 9\}$ and for face classification, $c \in \{\text{face}, \text{not face}\}$.

Algorithm steps:

1. All weights $w^{(c)}$ are initialized to zero.
2. The algorithm passes through the dataset in multiple epochs. Each epoch consists of:
   (a) Forward propagation: For a training instance x_i, compute activation (score) for each class using dot product: $a^{(c)} = w^{(c)} \cdot x_i$ The predicted class $\hat{y}_i$ is $\hat{y}_i = \arg\max_c a^{(c)}$, the class with the maximum score.
   (b) If $\hat{y}_i = y_i$ (predicted is correct), no update is made. If $\hat{y}_i \neq y_i$ (predicted is wrong), the model classified incorrectly. We compute the error vector $\delta^{(c)} = \begin{cases} x_i & \text{if } c = y_i \\ -x_i & \text{if } c = \hat{y}_i \\ 0 & \text{otherwise} \end{cases}$
   (c) Update weight: For the predicted class and the correct class, the weights are updated as follows:

$$w^{(y_i)} \leftarrow w^{(y_i)} + \eta x_i \quad \text{(reward the correct class)}$$

$$w^{(\hat{y}_i)} \leftarrow w^{(\hat{y}_i)} - \eta x_i \quad \text{(penalize the wrong class)}$$

   (η is the learning rate)

If the prediction is wrong, the algorithm updates the weights by adding the input vector to the weight vector of the correct class, then subtracting it from the weight vector of the predicted class. This adjustment will push future predictions closer to the correct label and allows the model to learn over time. The Perceptron converges if the data is linearly separable, but it lacks the ability to model nonlinear patterns due to the absence of hidden layers or activation functions like in neural networks.

4 Three-layer neural network with our implementation

The three-layer neural network we implemented consists of two hidden layers with ReLU activation functions and an output layer with a softmax activation.

Input Layer $\rightarrow$ Hidden Layer 1 (ReLU) $\rightarrow$ Hidden Layer 2 (ReLU) $\rightarrow$ Output Layer (Softmax)

Input layer: accepts flattened image vectors (784×1 2D array for 28×28 images).
Hidden layers: two layers using ReLU activation.
Output layer: applies softmax function to produce probabilities for each class.
During training, the true class labels are converted to one-hot encoded vectors to compute the cross-entropy loss.

Weight initialization:

- He initialization for the hidden layers with ReLU activation: $W \sim \mathcal{N}\left(0, \frac{2}{\text{number of inputs to the layer}}\right)$

- Xavier initialization for the output layer with softmax: $W \sim \mathcal{N}\left(0, \frac{1}{\text{number of inputs to the layer}}\right)$
- Initialize bias terms to 0.

The forward pass uses these layer activations:

$$\begin{aligned} Z_1 &= XW_1 + b_1, \quad A_1 = \text{ReLU}(Z_1) \\ Z_2 &= A_1W_2 + b_2, \quad A_2 = \text{ReLU}(Z_2) \\ Z_3 &= A_2W_3 + b_3, \quad \hat{Y} = \text{softmax}(Z_3) \end{aligned}$$

In backpropagation, the gradients are computed using the chain rule:

- $\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$
- Gradients of the weights include the regularization term.
- Parameters are updated using gradient descent: $W := W - \eta \cdot \frac{\partial \mathcal{L}}{\partial W}$

During training, we use mini-batch gradient descent. For each epoch:

1. Shuffle the training data.
2. Divide dataset into mini batches.
3. Then each batch goes through forward propagation, loss computation, backpropagation, and weight update.

During prediction, we run a forward pass and the predicted class is $\hat{y} = \arg\max_c \hat{y}_c$ (class with max score).

5 Three-layer neural network using Pytorch library

Using the PyTorch library, we implemented another three-layer neural network. This network has two hidden layers with ReLU activations, batch normalization after each hidden layer, and an output layer. PyTorch's built-in functionality helps to greatly simplify the implementation.

The first fully connected layer `fc1` maps the input to `hidden1` neurons.
The second fully connected layer `fc2` maps from `hidden1` to `hidden2` neurons.
The last fully connected layer `fc3` produces outputs for each class, with dimension `output_dim`.

We also apply batch normalization after each hidden layer (using `nn.BatchNorm1d` from PyTorch) to normalize the inputs to each layer, which helps with stability during training. The ReLU activation function is then used after each batch normalization.

The forward pass computes the following:
Hidden Layer 1: $x\text{ReLU}(\text{BatchNorm1}(\text{fc1}(x)))$
Hidden Layer 2: $x\text{ReLU}(\text{BatchNorm2}(\text{fc2}(x)))$
Output Layer: $\text{output}\text{fc3}(x)$

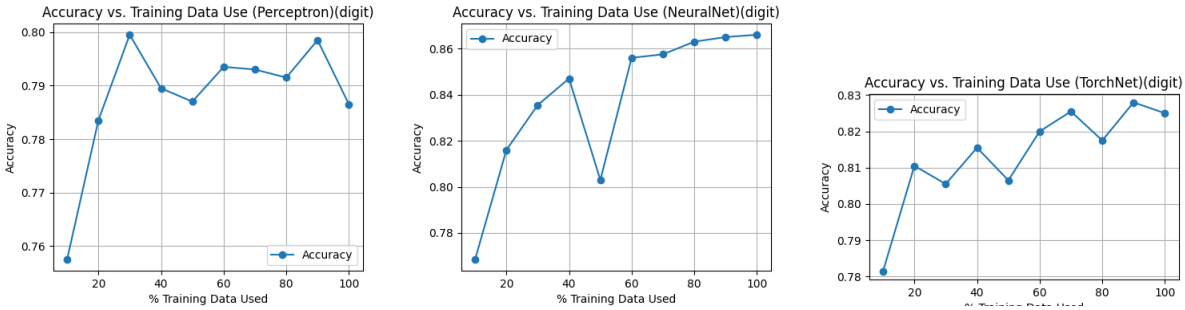
Each hidden layer uses batch normalization followed by the ReLU activation, and the final output is computed by passing the output of the second hidden layer through the output layer.

For training, the PyTorch implementation automatically handles backpropagation and gradient descent.

6 Results & Insights

We compared the performance of the perceptron, our custom three-layer neural network, and the PyTorch neural network on both digit and face classification using the provided datasets. We trained each model in increasing percentages of the training dataset, from 10% to 100%, in 10% increments, then output training time and classification accuracy (mean ± standard deviation) averaged over multiple runs.

Digit Classification Results



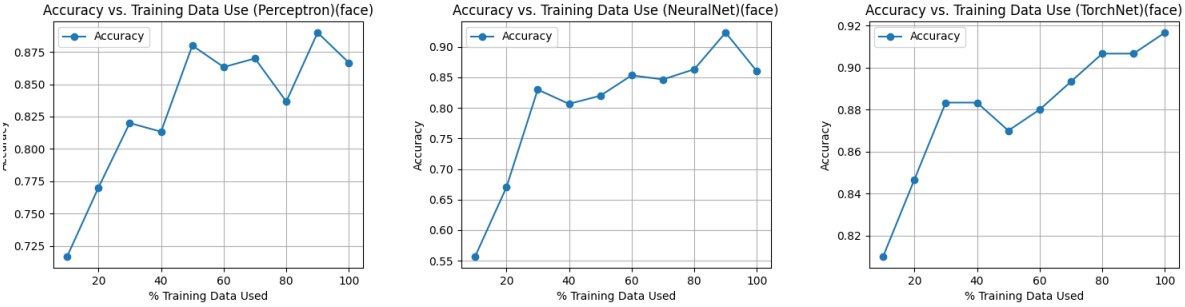
Digit Classification Accuracy: Perceptron, Custom Neural Net, PyTorch Neural Net

The Perceptron performed well, starting out at 75.7% and reaching 78.6% at 100% of the training data. Its training time remained very fast overall (under 0.3s).

However, the custom three-layer neural network outperformed the Perceptron in accuracy for all training data size, starting at 76.5% and staying consistently above 80% after using more than 20% of the training data, peaking at 86.5% at 100% of the data used. Yet, it was significantly slower than the Perceptron, starting out at 0.24 seconds at 10% training data used but taking up to 2.58 seconds at full data used.

The PyTorch three-layer neural network had strong performance, starting out at 78.2% accuracy and reaching 82.5% at 100% of the data used, with very fast speeds. Although its final accuracy was 4% lower than the custom neural network, its speed was significantly better and peaked at 0.26 seconds at 100% data used.

Face Classification Results



Face Classification Accuracy: Perceptron, Custom Neural Net, PyTorch Neural Net

For face classification, all models had higher accuracies compared to digit classification. The Perceptron reached 87% accuracy at full training data used with the fastest training time staying under 0.05 seconds

for all training data sizes. The custom three-layer neural network reached around 86% accuracy, but was significantly slower and took 0.85 seconds at full training data used. The PyTorch neural network had the best accuracy peaking at 91.9% accuracy at full training data used and had training time at most 0.28 seconds, which was faster than the three-layer neural network but still significantly slower than the Perceptron.

Insights

The Perceptron showed strong performance with extremely fast training times for both digit and face classification. It is simple to implement and has strong accuracy, especially for linearly separable data like the face classification. However, its performance plateaus more quickly and it cannot learn complex nonlinear patterns as well as the three-layer neural network with the digit classification task.

On the other hand, the custom three-layer neural network consistently achieved higher accuracy than the Perceptron, especially with increasing training data. Its hidden layers and nonlinear activation functions allowed it to learn more complex relationships in the data, making it more effective in the nonlinear digit classification task. However, this significantly increased training time. It is ideal for when accuracy is prioritized and sufficient computational time and resources are available.

Additionally, we found that the ReLU activation function outperformed the Sigmoid function in both speed and accuracy for the three-layer neural network we implemented. Through trial and error, we adjusted the epochs and found that 15 was best for our neural network to not run too slowly while maintaining high accuracy.

The PyTorch neural network offered the best compromise between speed and performance. It used optimized libraries and efficient tensor operations that allowed it to achieve fast training times while keeping high accuracy levels. It performed nearly as well as the custom neural network in digit classification and outperformed both other models in face classification. While its training time was still slower than the Perceptron, it stayed significantly faster than the custom neural network across both classification tasks.