Intro to AI - Final Project

Sai Veeramachaneni (sav115), Ryan Donofrio (rd1036), Aveesh Patel (ap2165)

May 2025

1 Introduction

Character Recognition and face detection. In this project we build and compare:

- A linear perceptron classifier
- A custom three layer neural network implemented from scratch
- A three layer neural network implemented in PyTorch

We evaluate all models on two dataset digits and faces, using varying fractions of training data then analyze accuracy and computational cost.

2 Algorithms

2.1 Perceptron Algorithm

The Perceptron Algorithm works by learning a set of weights that separate data points into distinct classes using a linear decision boundary.

1. Initialization

```
def __init__(self, input_size, num_classes):
    self.weights = np.random.randn(input_size, num_classes) * 0.01
    self.bias = np.zeros((1, num_classes))
```

   Starts by initializing random weights and a bias of zero for each class

2. Forward Pass

```
def forward(self, x):
    return np.dot(x, self.weights) + self.bias
```

Returns this equation $score = x * W + b$

3. Training

```
def train(self, X, y, lr = 0.01, epochs = 100):
for epoch in range(epochs):
    for i in range(X.shape[0]):
        # Forward
        scores = self.forward(X[i:i+1])

        pred = np.argmax(scores)

        if pred != y[i]:
            self.weights[:, y[i]] += lr * X[i]
            self.weights[:, pred] -= lr * X[i]
            self.bias[0, y[i]] += lr
            self.bias[0, pred] -= lr
```

It computes the score and gets the predicted class. If the prediction is incorrect, it will increase the weight of the correct and decrease the weight of the incorrectly predicted class.

4. Prediction

```
def predict(self, X):
    scores = self.forward(X)
    return np.argmax(scores, axis=1)
```

Returns the class with the highest score

5. Evaluate

```
def evaluate(self, X, y):
    preds = self.predict(X)
    return np.mean(preds == y)
```

Returns the accuracy of predicted labels to true labels

2.2 Custom Neural Network

1. Initialization

```
def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
    self.W1 = np.random.randn(input_size, hidden_size1) * 0.01
    self.b1 = np.zeros((1,hidden_size1))

    self.W2 = np.random.randn(hidden_size1, hidden_size2) * 0.01
    self.b2 = np.zeros((1,hidden_size2))

    self.W3 = np.random.randn(hidden_size2, output_size) * 0.01
    self.b3 = np.zeros((1,output_size))
```

   Creates a network with three layers of weights and biases. The weights are initialized with small random values and biases are set to 0.

2. Forward Pass

```
def forward(self, x):
    self.z1 = np.dot(x, self.W1) + self.b1
    self.a1 = self.relu(self.z1)

    self.z2 = np.dot(self.a1, self.W2) + self.b2
    self.a2 = self.relu(self.z2)

    self.z3 = np.dot(self.a2, self.W3) + self.b3
    self.a3 = self.softmax(self.z3)

    return self.a3
```

   The input is multiplied by the weights and added to biases, and after each of the hidden layers the ReLU function is added to create non-linearity. The final output is put into a softmax function to create the class probabilities.

3. Backward Pass (Backpropagation)

```
def backward(self, x, y, lr):
    m = x.shape[0]

    dz3 = self.a3.copy()
    dz3[range(m), y] -= 1
```

```
        dz3 /= m

        dW3 = np.dot(self.a2.T, dz3)
        db3 = np.sum(dz3, axis=0, keepdims=True)

        dz2 = np.dot(dz3, self.W3.T) * self.relu_derivative(self.z2)
        dW2 = np.dot(self.a1.T, dz2)
        db2 = np.sum(dz2, axis=0, keepdims=True)

        dz1 = np.dot(dz2, self.W2.T) * self.relu_derivative(self.z1)
        dW1 = np.dot(x.T, dz1)
        db1 = np.sum(dz1, axis=0, keepdims=True)

        self.W3 -= lr * dW3
        self.b3 -= lr * db3
        self.W2 -= lr * dW2
        self.b2 -= lr * db2
        self.W1 -= lr * dW1
        self.b1 -= lr * db1
```

The gradient of the loss is computed using the prediction and true label. The gradients are propagated backward through the network using chain rule and using the derivative of ReLU. Each layer's weights and biases are updated using the gradients and the learning rate.

4. Training

```
    def train(self, X, y, lr = 0.01, epochs = 100, batch_size = 32):
        for epoch in range(epochs):

            idx = np.arange(X.shape[0])
            np.random.shuffle(idx)

            for i in range(0, X.shape[0], batch_size):
                batch_idx = idx[i:i+batch_size]
                X_batch = X[batch_idx]
                y_batch = y[batch_idx]

                self.forward(X_batch)

                self.backward(X_batch, y_batch, lr)
```

The data is shuffled each epoch to make sure the model does not learn the order of data and prevent overfitting. Then it is divided into batches and

does the forward and backward pass for each batch.

5. Prediction

```
def predict(self, X):
    scores = self.forward(X)
    return np.argmax(scores, axis=1)
```

Uses forward pass to get softmax scores and selects the class with the highest score.

6. Evaluate

```
def evaluate(self, X, y):
    preds = self.predict(X)
    return np.mean(preds == y)
```

Compares the predicted labels to the true labels.

2.3 PyTorch Neural Network

1. Initialization

```
class TorchNN(nn.Module):
    def __init__(self, input_size, hidden1, hidden2, output_size):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(input_size, hidden1),
            nn.ReLU(inplace=True),
            nn.Linear(hidden1, hidden2),
            nn.ReLU(inplace=True),
            nn.Linear(hidden2, output_size)
        )
```

Defines three layer feedforward network with ReLU activations

2. Forward Pass

```
    def forward(self, x):
        return self.model(x)
```

Computes raw logits by passing input through sequential model

3. Training Loop

```
def train_and_evaluate(model, X_train, y_train, X_val, y_val, config):
    train_loader = DataLoader(TensorDataset(X_train, y_train),
                              batch_size=config.batch_size,
                              shuffle=True)
    optimizer = optim.SGD(model.parameters(),
                          lr=config.lr,
                          weight_decay=1e-4)
    criterion = nn.CrossEntropyLoss()
    for epoch in range(config.epochs):
        model.train()
        for xb, yb in train_loader:
            xb, yb = xb.to(config.device), yb.to(config.device)
            optimizer.zero_grad()
            logits = model(xb)
            loss = criterion(logits, yb)
            loss.backward()
            optimizer.step()
```

Uses SGD and cross entropy loss, runs for 50 epochs with batch size 128

4. Evaluation

```
def evaluate(loader):
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for xb, yb in loader:
            preds = model(xb).argmax(dim=1)
            correct += (preds == yb).sum().item()
            total += yb.size(0)
    return correct / total
```

Measures accuracy on validation or test DataLoader

3 Evaluation Setup

We sample 10%, 20%, ..., 100% of each training set repeat each split 5 times with different random seeds and record mean ± std of accuracy and training time

Hyperparameters All experiments use learning rate 0.01. We vary batch size and epochs per model:

Model	Epochs	Batch Size
Perceptron	100	N/A
Custom NN	100	32
PyTorch NN	50	128

Table 1: Training hyperparameters for each model

4 Results

Framework	Model	Dataset	%	Acc. $\pm$ Std	Time (s) $\pm$ Std
scratch	perceptron	digits	100	0.7746 ± 0.0056	3.89 ± 0.007
scratch	perceptron	faces	100	0.8760 ± 0.0137	1.62 ± 0.012
scratch	customNN	digits	100	0.8170 ± 0.0070	33.44 ± 4.86
scratch	customNN	faces	100	0.8667 ± 0.0267	8.96 ± 0.88
pytorch	customNN	digits	100	0.8148 ± 0.0067	2.49 ± 0.05
pytorch	customNN	faces	100	0.7267 ± 0.0532	0.66 ± 0.014

Table 2: Mean±std of test accuracy and training time at 100% data

5 Conclusion

We implemented three classifiers, evaluated across data fractions, and analyzed performance and timing

- **Perceptron:** On digits it plateaus around 78% because handwritten characters are not linearly separable in pixel space. On faces edge based features are simpler allowing to reach 88% accuracy
- **CustomNN:** Added nonlinearity and hidden layers let it steadily improve to 82% (digits) and 87% (faces). However Python backprop is computationally expensive training on 100% of digits can take up to 33s
- **PyTorchNN:** Matches customNN's accuracy within 1–2% but trains 3×–5× faster with efficient optimizers
- **Trade offs:** Custom implementations deepen algorithmic understanding but at cost of speed. Frameworks like PyTorch enable rapid experimentation

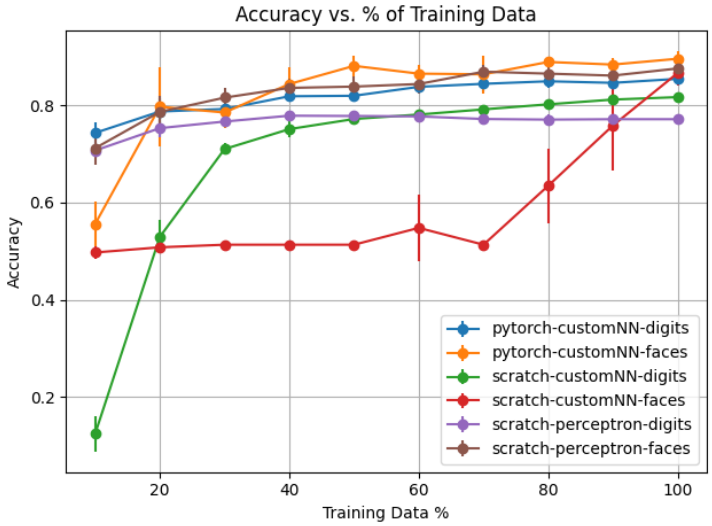

Figure 1: Accuracy vs. % of Training Data

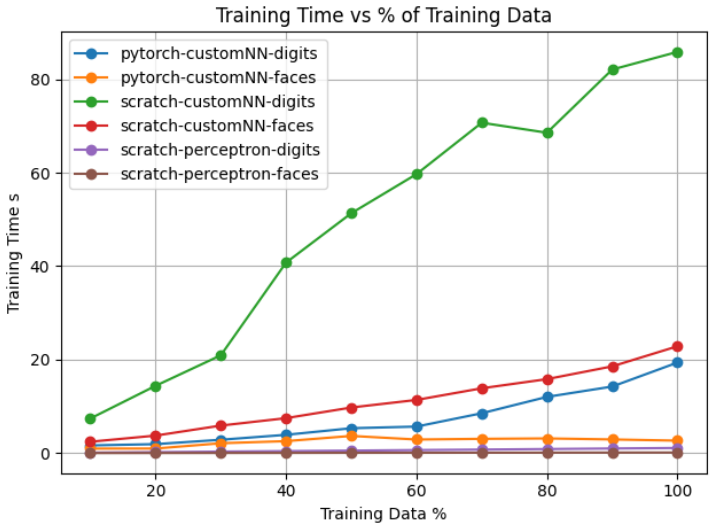

Figure 2: Training Time vs. % of Training Data