CS440 Final Project Report

Author: Khyaati Sigicherla

1. Overview of Project Files

This project implements three different classification algorithms —*Perceptron*, a *scratch-built neural network*, and a *PyTorch-based neural network*— to handle two different classification tasks. (1) classify handwritten digits (multi-class) or (2) classify face vs. non-face images (binary).

Brief summary of each file in the repository:

1. **`dataset.py`**
    - Contains helper functions to load image data (digits or faces) from text files.
    - Each dataset (train, validation, test) is contained in a pair of files: one with the ASCII-art images, another with the labels.
    - The functions `load_digit_data` and `load_face_data` read the lines, parse the ASCII images into NumPy arrays, and return `(X_train, y_train), (X_val, y_val), (X_test, y_test)` tuples.
    - Each image is flattened into a 1D vector
2. **`perceptron.py`**
    - Contains `OneVsAllPerceptron` class, which trains one binary perceptron per class (i.e., a one-vs-rest strategy) for multi-class tasks.
    - Each perceptron tracks a weight vector (and bias, if enabled) for each class.
3. **`nn_scratch.py`**
    - Implements a **three-layer neural network** from scratch using NumPy.
    - Uses ReLU activation in hidden layers and either softmax (multi-class) or sigmoid (binary) at the output.
    - Performs forward propagation, computes cross-entropy loss (with optional L2 regularization), and uses manual backpropagation to update weights.
4. **`nn_pytorch.py`**
    - Implements a **three-layer neural network** in **PyTorch**.
    - Defines a `NeuralNetPyTorch` class with two hidden layers (ReLU activated) and one output layer.
    - Provides a `train_pytorch_model` function to handle mini-batch training with Adam/SGD optimizer and the appropriate loss (`CrossEntropyLoss` for multi-class, `BCEWithLogitsLoss` for binary).

5. **`train.py`**
    - This file is the main file to run the project !
    - Uses `argparse` to parse command-line arguments such as:
        - `--task` (`digits` or `faces`)
        - `--model` (`perceptron` , `nn_scratch` , `nn_pytorch`)
        - `--epochs` , `--lr` , `--batch_size` , etc.
    - Loads the corresponding dataset (`digits` or `faces`), normalizes the features, and sets the correct number of classes (10 for digits, 2 for faces).
    - Trains the selected model on various fractions (0.1 to 1.0) of the training data multiple times to get average performance and standard deviation
    - Print results (training time, test accuracy, test error, test accuracy standard devication).
    - These final results are also visualized with plots.

2. How to Run `train.py`

1. **Ensure you have the required dependencies** (NumPy, PyTorch, Matplotlib, etc.):
2. **Navigate to the project directory**, where `train.py` is located.
3. **Run `train.py`** with desired arguments. Examples:

```
# Example 1: Train the scratch-built neural network on the digit dataset:
python train.py --task digits --model nn_scratch --epochs 10 --lr 0.01

# Example 2: Train the one-vs-all perceptron on the face dataset,
#            using 5 epochs and lr=0.005:
python train.py --task faces --model perceptron --epochs 5 --lr 0.005

# Example 3: Train the PyTorch neural network on the digit dataset with 20
epochs:
python train.py --task digits --model nn_pytorch --epochs 20
```

`train.py` will appropriate load either digit or face data, depending on `--task` and construct the specified model `--model` . It will train the model across a range of fractions of the training set (0.1, 0.2, …, 1.0), and print out the average training time and test accuracy across multiple runs per fraction.

---

3. The Three Algorithms (with Parameter Values)

Now we will delve into a detailed summary of each algorithm including the hyperparameters it accepts and some key design considerations.

3.1 Perceptron

Location: `perceptron.py`, class `OneVsAllPerceptron`.

1. **Architecture
   - For *multi-class classification*, trains one *binary* perceptron for each class c.
   - Each perceptron sees examples of class c as label +1 and everything else as -1.
   - At prediction time, each class's perceptron produces a *score*, and the predicted label is the class with the highest score.
   - **One-vs-All** vs. One-vs-One: simpler to implement, though less efficient for many classes.
   - **Zero initialization** for weights/bias.
   - L2 Regularization where lambda = 0.01
2. **Update Rule**
   - For each sample (x_i, y_i), compute the predicted sign $\hat{y}_{ic}$ for each class c.
   - If it is incorrect (sign mismatch), update the corresponding weight vector: $W_c \leftarrow W_c + \eta \times (\text{target}) \times x_i$
   - Update the bias term if `use_bias=True`.
3. **Parameter Values**
   - **`lr`**: Learning rate. Default is 0.01 in the constructor.
   - **`epochs`**: Number of passes through the training data (default 10).
   - **`use_bias`**: Boolean to indicate whether a bias term is used (default `True`).
   - **`input_dim`**: Number of features (automatically set).
   - **`num_classes`**: 2 or 10, depending on the dataset.

3.2 Neural Network from Scratch

Location: `nn_scratch.py`, class `NeuralNetScratch`.

1. **Architecture**
   - A **3-layer feed-forward** network:
     - Input layer → Hidden layer 1 → Hidden layer 2 → Output layer.
     - ReLU activation in hidden layers.
     - Output layer uses softmax (multi-class) or sigmoid (binary).
2. **Parameter Values**
   - **`input_dim`**: Number of features (e.g., 784 for digits).

- **hidden_dim1, hidden_dim2** : Sizes of hidden layers. By default, 64/32 for digits, 128/64 for faces.
- **output_dim** : 10 (digits, multi-class) or 1 (faces, binary).
- **lr** : Learning rate (default 0.01).
- **epochs** : Number of full passes (default 10).
- **batch_size** : Mini-batch size (default 32).
- **reg_lambda** : L2 regularization strength (default 0.0 or 0.001).

3. **Forward Propagation**
   - $Z1 = XW1 + b1,\ A1 = \mathrm{ReLU}(Z1)$
   - $Z2 = A1W2 + b2,\ A2 = \mathrm{ReLU}(Z2)$
   - $Z3 = A2W3 + b3$
   - $A3 = \mathrm{softmax}(Z3)$ if multi-class, else $\mathrm{sigmoid}(Z3)$.
4. **Loss & Backpropagation**
   - **Cross-entropy** loss is used.
   - **L2 regularization** adds $\frac{\lambda}{2N}$ times the sum of squared weights.
   - We manually compute gradients $(dW1, dW2, dW3, \ldots)$ via chain rule.
5. **Design Choices**
   - **ReLU** in hidden layers for better gradient flow.
   - **"He initialization"**:
     - it initializes weights from a normal distribution with a mean of 0 and a variance of 2/n, where 'n' is the number of input units to a neuron.
     - the variance is inversely proportional to the number of input units, ensuring that the gradients don't become too small or too large during training
   - **Mini-batch training** (batch size 32).
   - **One-hot labels** for multi-class tasks.

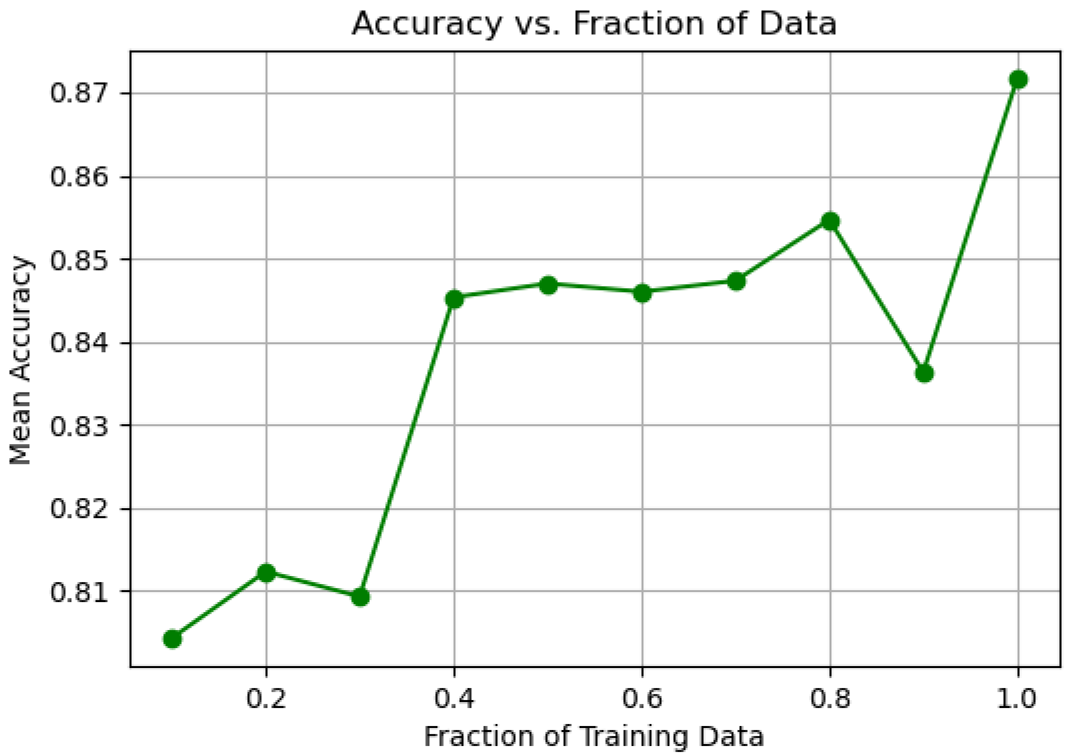
3.3 Neural Network in PyTorch

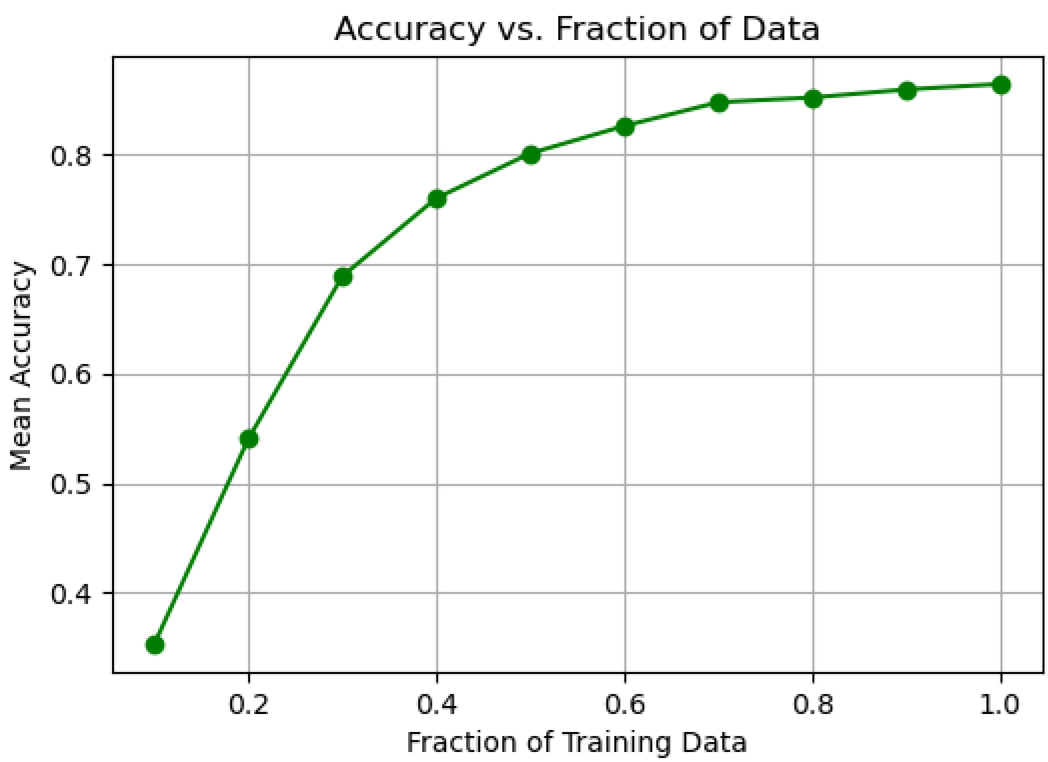
Location: `nn_pytorch.py`, classes `NeuralNetPyTorch` and function `train_pytorch_model`.

1. **Architecture**
   - Similar 3-layer NN with two ReLU hidden layers.
   - `self.fc1` → `self.fc2` → `self.fc3`.
   - No explicit softmax layer since PyTorch's `CrossEntropyLoss` covers that internally.
2. **Parameter Values**
   - **input_dim** : Number of input features.
   - **hidden_dim1, hidden_dim2** : Hidden layer sizes (e.g., 64/32 or 128/64).
   - **output_dim** : 10 (digits) or 1 (faces).

- **`epochs`** : Default 10, or set via `--epochs`.
- **`lr`** : Learning rate for the optimizer (default 0.01).
- **`batch_size`** : Default 32, can be changed via `--batch_size`.
- **`weight_decay`** : L2 regularization parameter (default 0.001).
- **`multi_class`** : Whether to use `CrossEntropyLoss` or `BCEWithLogitsLoss`.

3. **Training in `train_pytorch_model`**
   - Creates an optimizer, Adam by default (experimented with SGD).
   - Observe that Adam (first) outperforms SGD (first) for example on the digits dataset as even with 10% of the train data it exhibits over 70% accuracy whereas with SGD with 10% of the train data it has accuracy below 40%. However at 100% of train data both work fairly "well" in that they have over 80% accuracy.





- Chooses `nn.CrossEntropyLoss()` if multi-class, `nn.BCEWithLogitsLoss()` if binary.
- Loops over epochs, shuffles data, processes mini-batches, calls `.backward()`, and `optimizer.step()`.
- Periodically prints training accuracy.

4. **Design Choices**

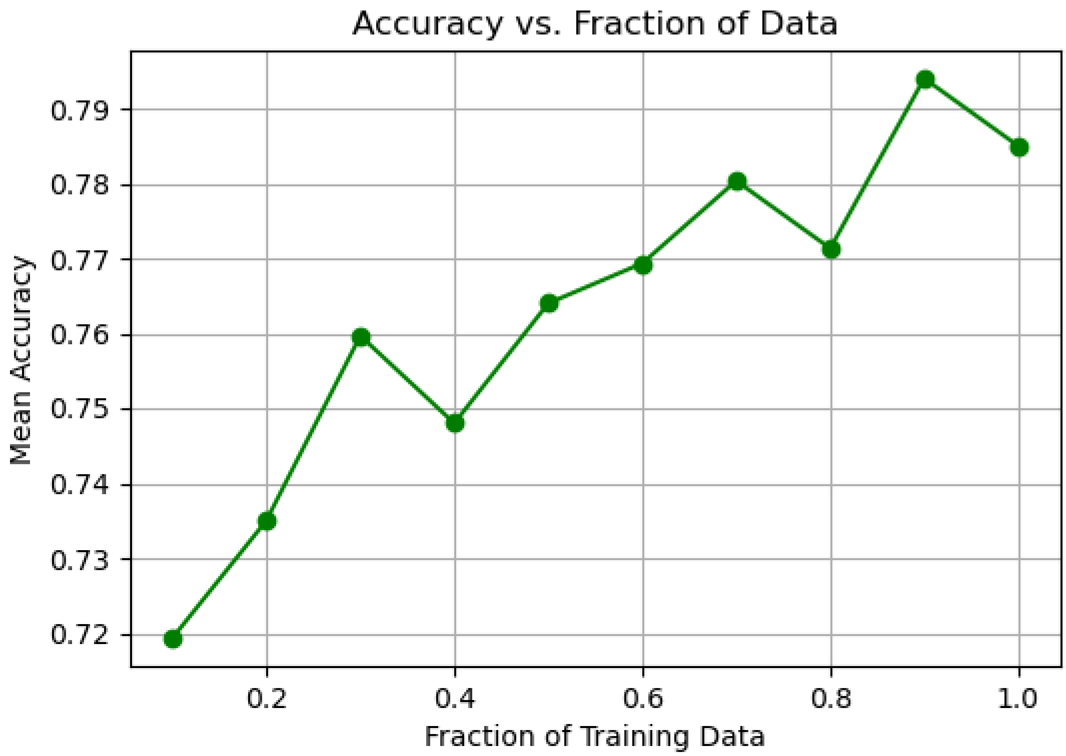
- **ReLU** in hidden layers, no final activation (handled by the loss).
- nn.Sigmoid() --> low accuracy on both datasets; opt for "rectified linear unit" ReLU R(z) = max(0,z).
- For multi-class digits, no need to add nn.Softmax(dim=1) at the end, since we are using CrossEntropyLoss which applies log-softmax internally
- **Adam** optimizer for automatic gradient-based updates

---

4. Graphs & Algorithm Comparison

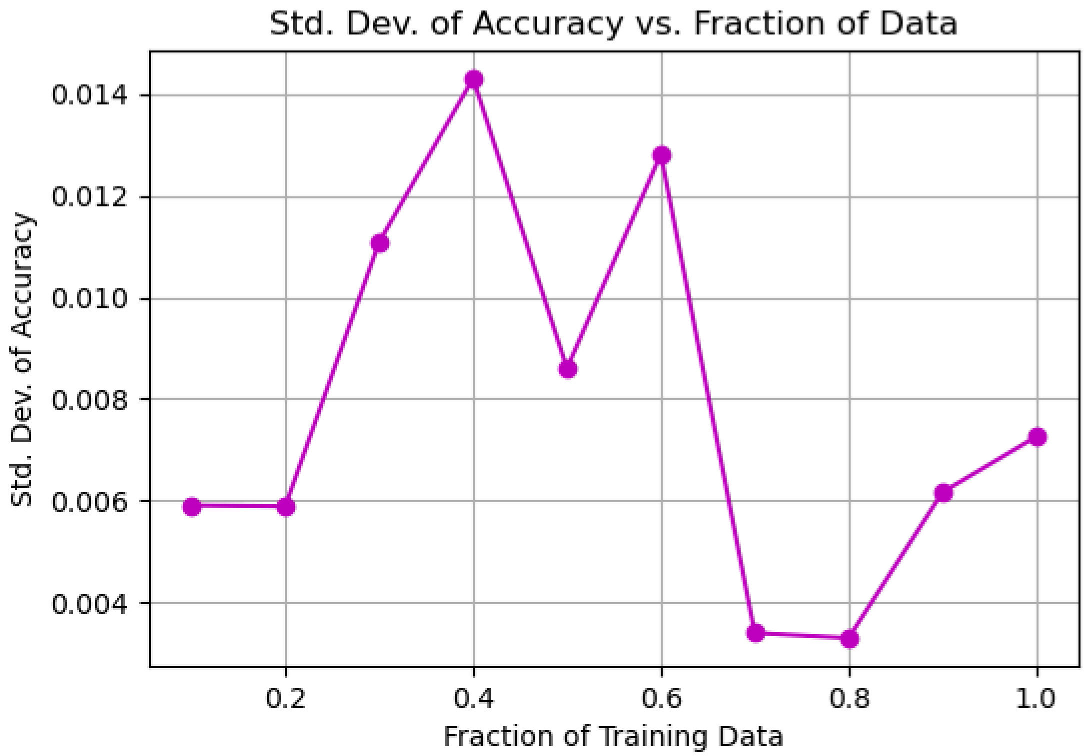
TASK: Digit Classification (Multi-Classs)

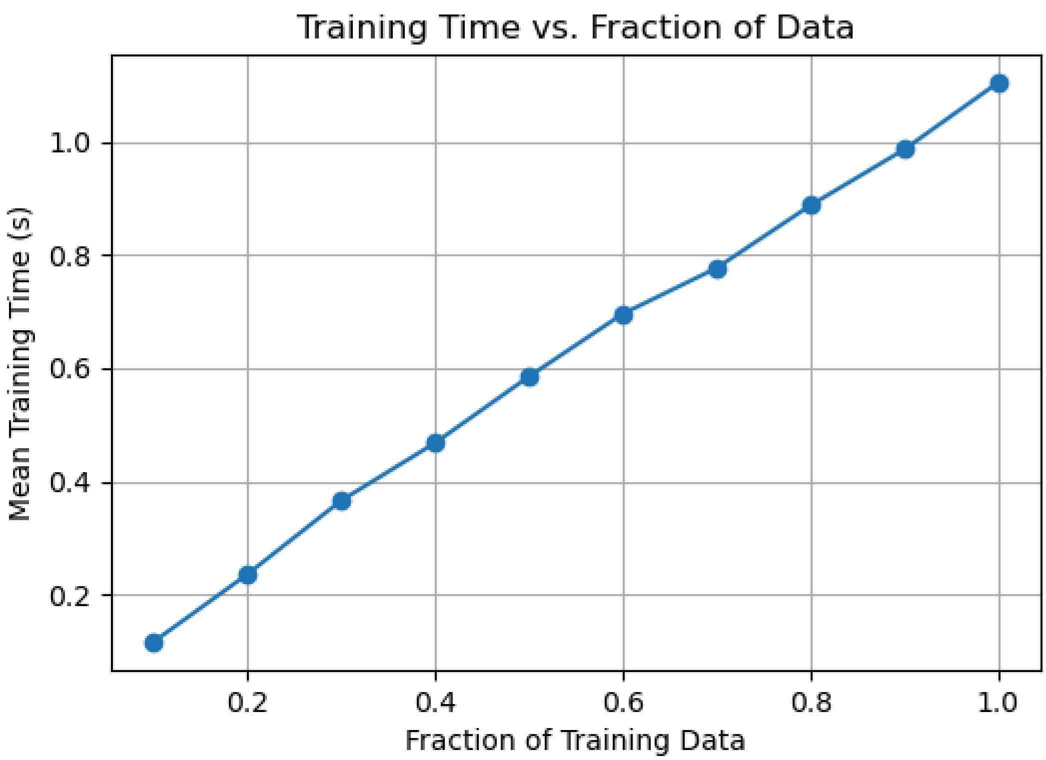
perceptron:



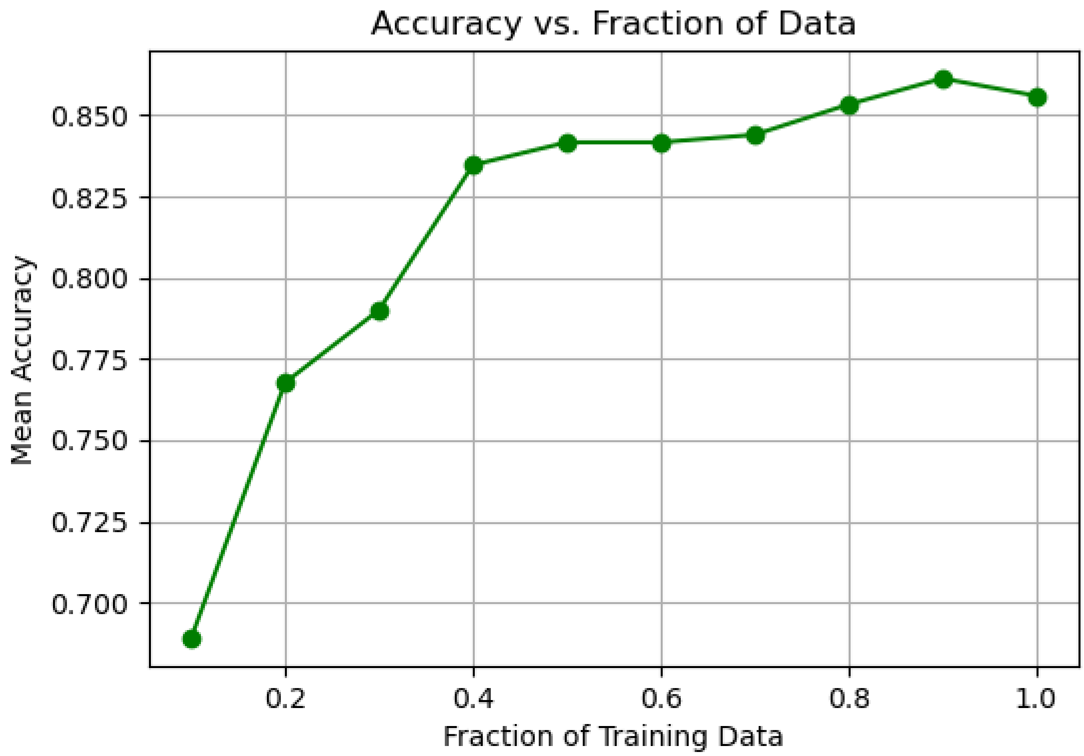
Accuracy vs. Fraction of Data
Mean Accuracy
0.72
0.73
0.74
0.75
0.76
0.77
0.78
0.79
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data

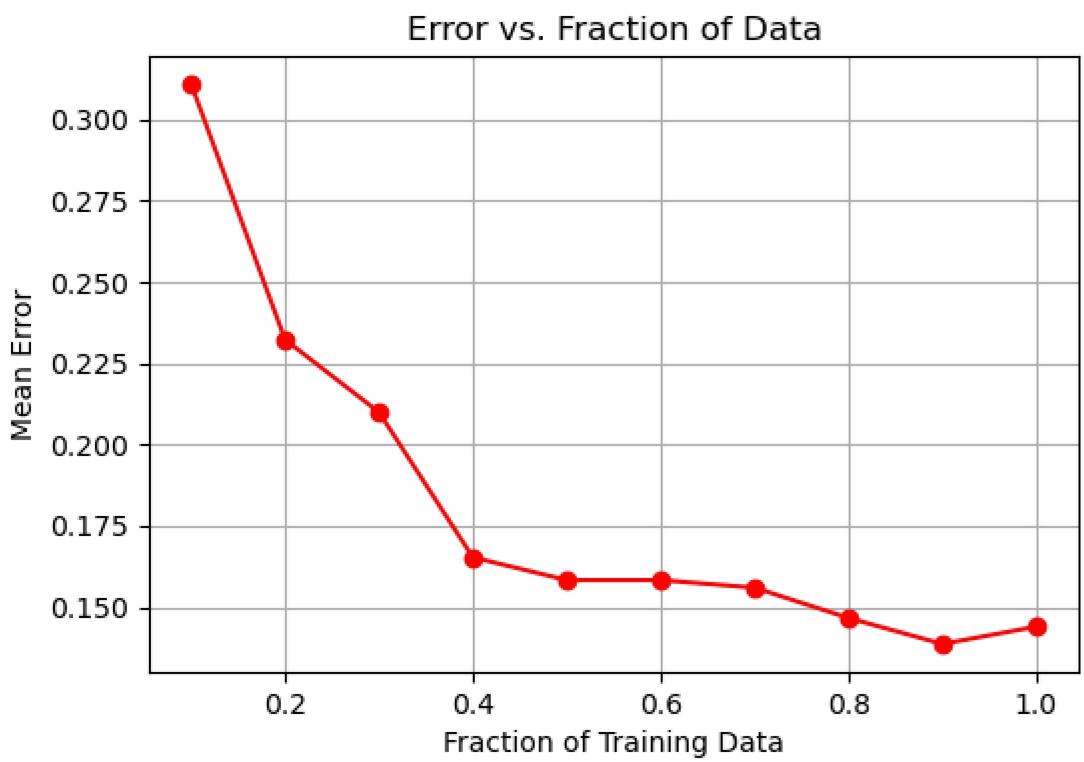
Error vs. Fraction of Data
Mean Error
0.21
0.22
0.23
0.24
0.25
0.26
0.27
0.28
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data

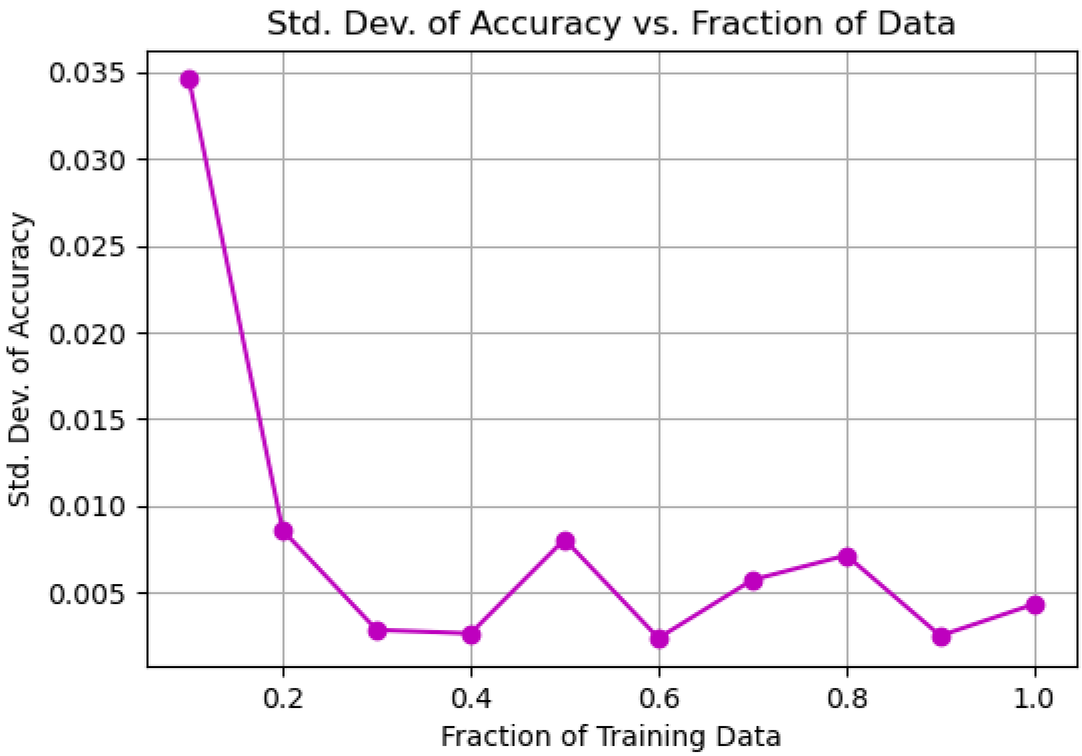

Std. Dev. of Accuracy vs. Fraction of Data
Std. Dev. of Accuracy
0.014
0.012
0.010
0.008
0.006
0.004
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data


Training Time vs. Fraction of Data
Mean Training Time (s)
1.0
0.8
0.6
0.4
0.2
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data

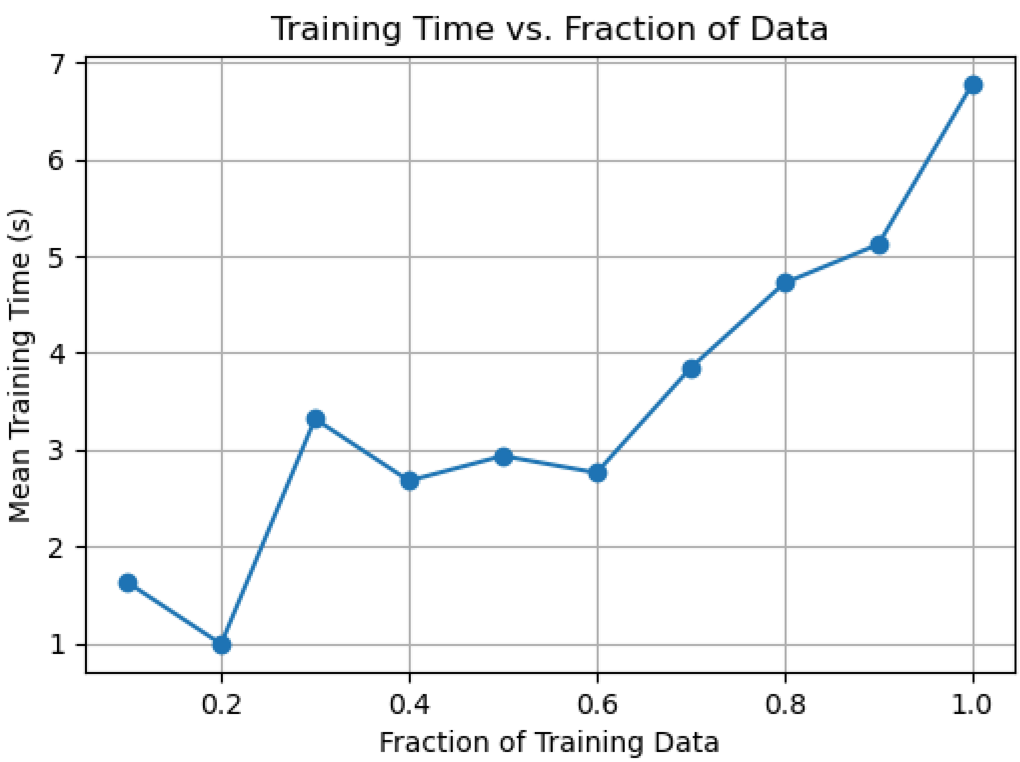
nn_scratch


Accuracy vs. Fraction of Data
Mean Accuracy
0.850
0.825
0.800
0.775
0.750
0.725
0.700
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data


Error vs. Fraction of Data
Mean Error
0.300
0.275
0.250
0.225
0.200
0.175
0.150
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data

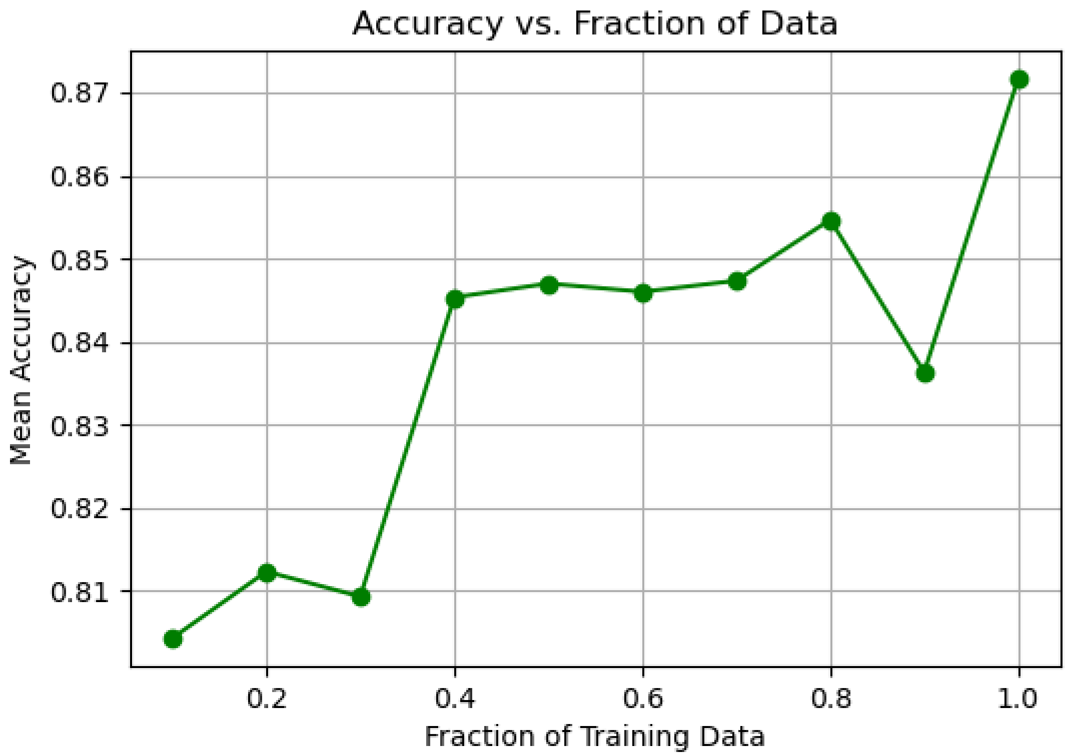


Std. Dev. of Accuracy vs. Fraction of Data
Std. Dev. of Accuracy
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data

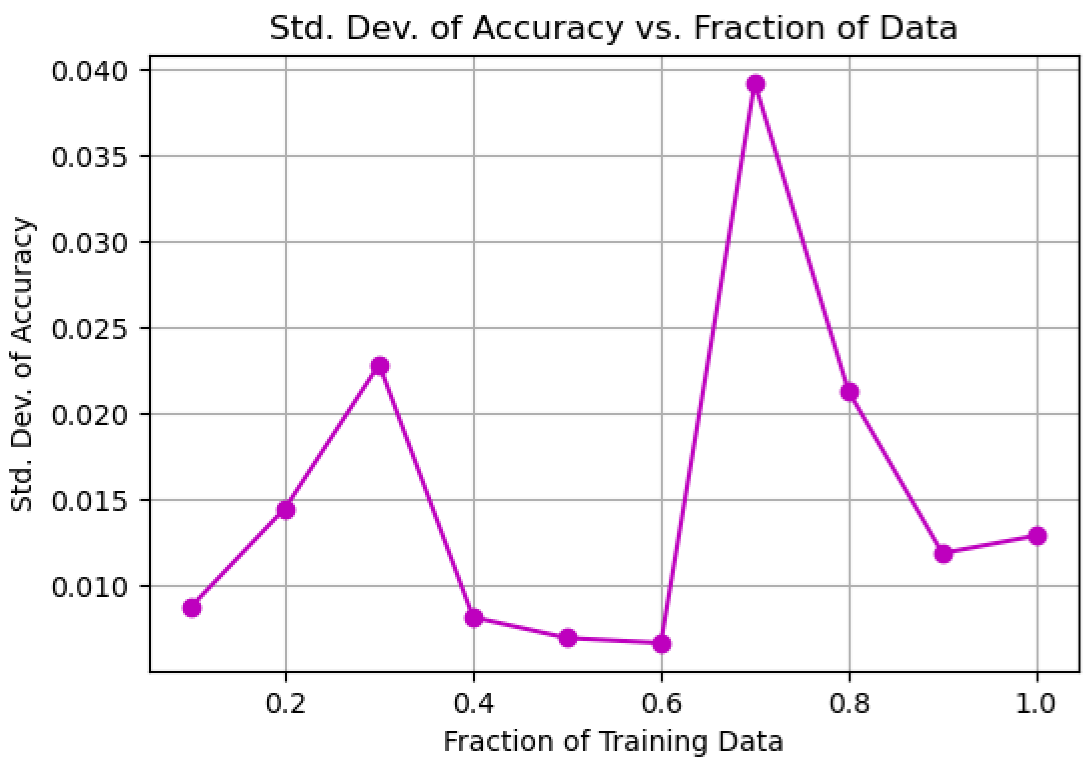


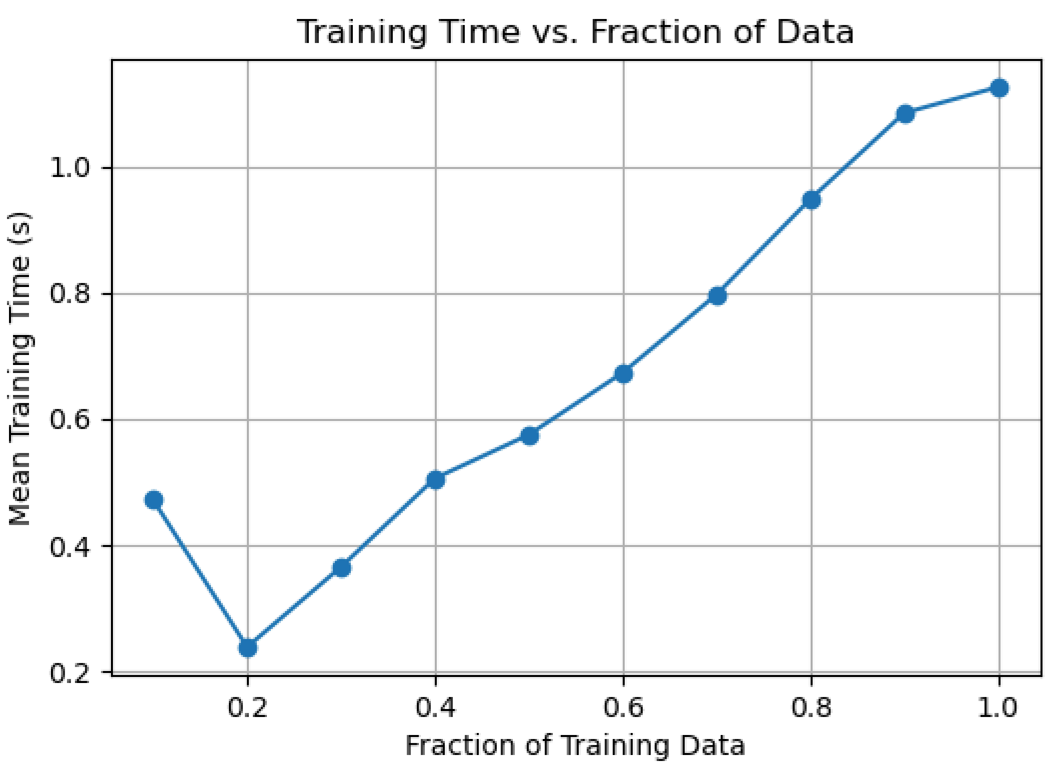
Training Time vs. Fraction of Data
Mean Training Time (s)
7
6
5
4
3
2
1
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data

nn_pytorch


Accuracy vs. Fraction of Data
Mean Accuracy
0.87
0.86
0.85
0.84
0.83
0.82
0.81
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data


Error vs. Fraction of Data
Mean Error
0.19
0.18
0.17
0.16
0.15
0.14
0.13
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data





Metric (@ 100 % train data)	Perceptron	NN from scratch	NN PyTorch
Accuracy	≈ 0.79	≈ 0.86	≈ 0.87 ▲ *best*
Std. dev. of accuracy	≈ 0.007	≈ 0.004 ▼ *best*	≈ 0.013
Training time (s)	≈ 1.05 s ▼ *fastest*	≈ 6.8 s	≈ 1.12 s

1 Accuracy (ACC)

- **PyTorch NN** leads with a peak of around **87 %** and showing a steadily rising curve once the model has at least 40% of the training data.
- **Scratch NN** is only slightly lower (~ 85 %), converging smoothly as data grows.
- **Perceptron** trails at roughly **79 %** even with all data.
  If we are purely interested in predictive power, PyTorch NN is the winner, and NN from scratch is a close runner-up, and the perceptron is a distant third.

2 Stability (STD of accuracy across 3 runs)

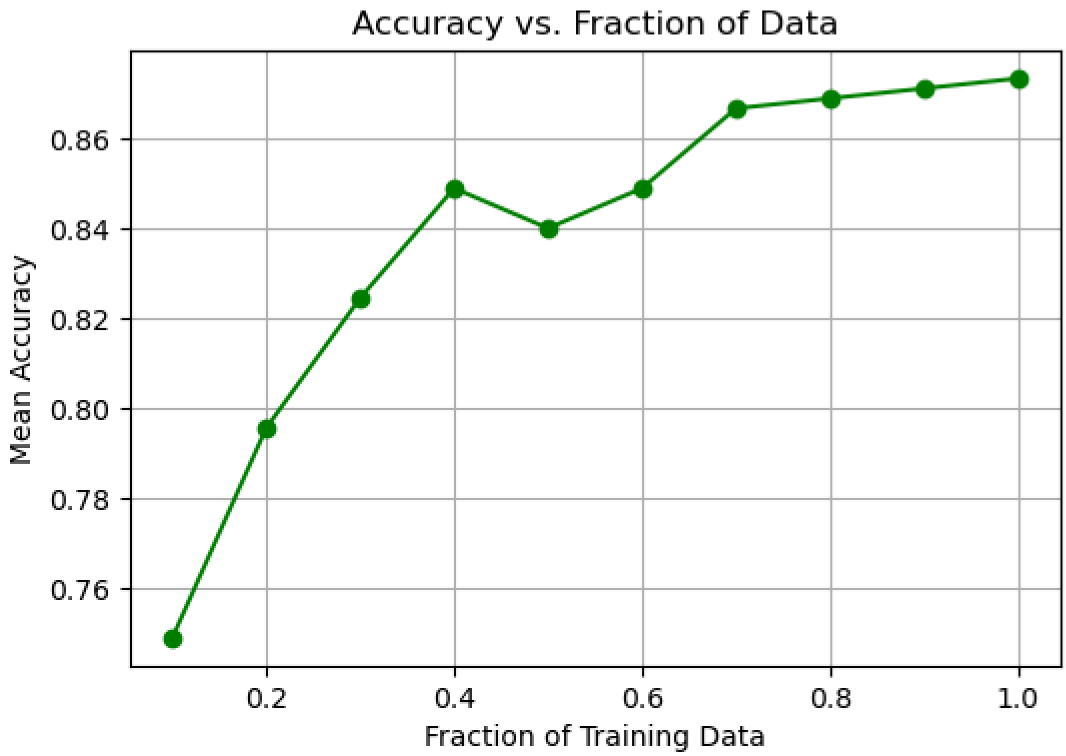
- **Scratch NN** has the **smallest variation** (≤ 0.005 after 40 % data) – once initial weight noise is averaged out, its deterministic NumPy implementation exhibits stability.
- **Perceptron** shows slightly higher spread (~ 0.007–0.014) but still quite low.
- **PyTorch NN** is the most variable (spikes to ~ 0.04 at 70 % data). Why? Likely because there is noise added with Adam's stochastic updates and different mini-batch orders.
  For a classifer that reliably gives same accuracy metrics, choosing the scratch neural net yields nearly identical results; PyTorch needs more runs or a fixed rng seed to increase stability

3 Training time (TIME)

- **Perceptron** is fastest – linear algebra on one weight matrix ~ 1 s at full data.
- **PyTorch NN** slightly slower (~ 1.1 s at 100 %) thanks to vectorized Adam.
- **Scratch NN** is an *order of magnitude* slower (~ 6–7 s at 100 %) because every forward/backward pass is pure-Python NumPy with no parallelism.
  If the priority is speed stick with the perceptron though PyTorch is nearly same speed, while the NumPy scratch model lags behind in terms of time.

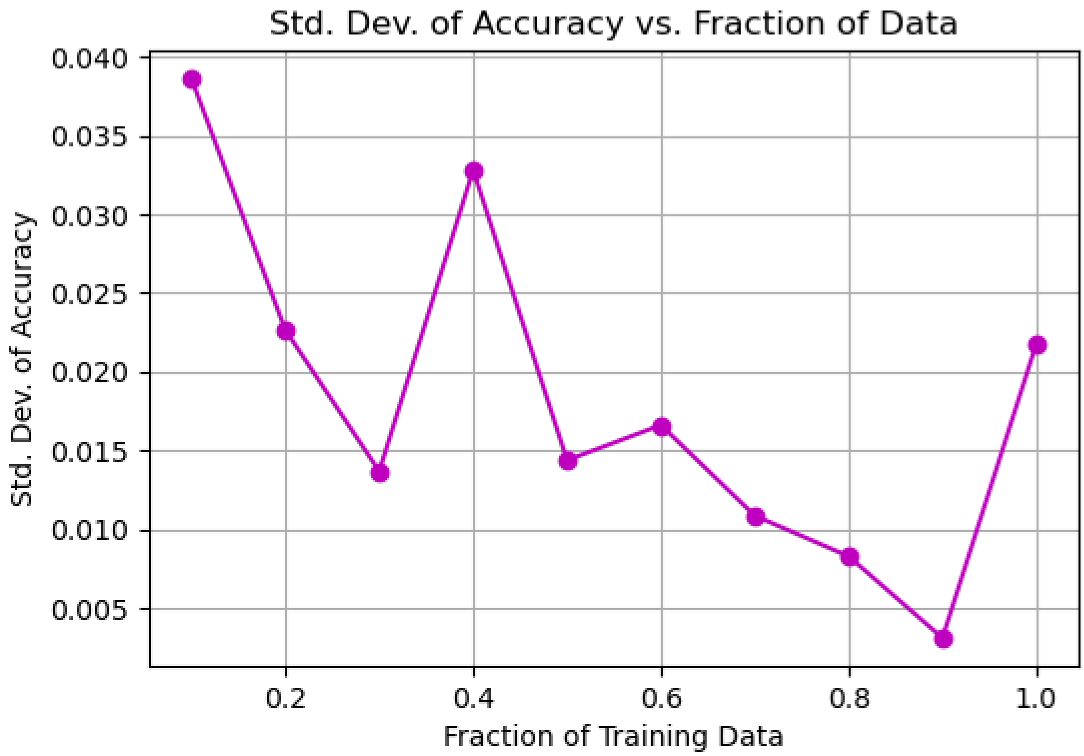
TASK: Face Classification (Binary)

perceptron:

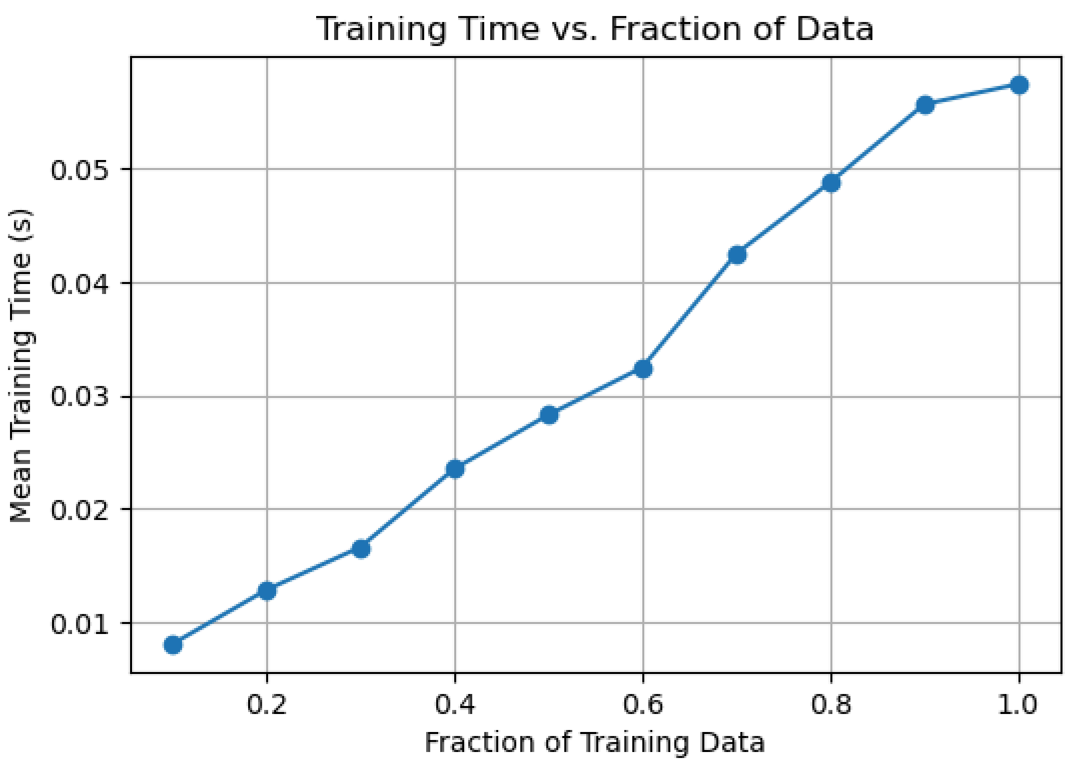


Accuracy vs. Fraction of Data
Mean Accuracy
0.86
0.84
0.82
0.80
0.78
0.76
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data

Error vs. Fraction of Data
Mean Error
0.24
0.22
0.20
0.18
0.16
0.14
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data

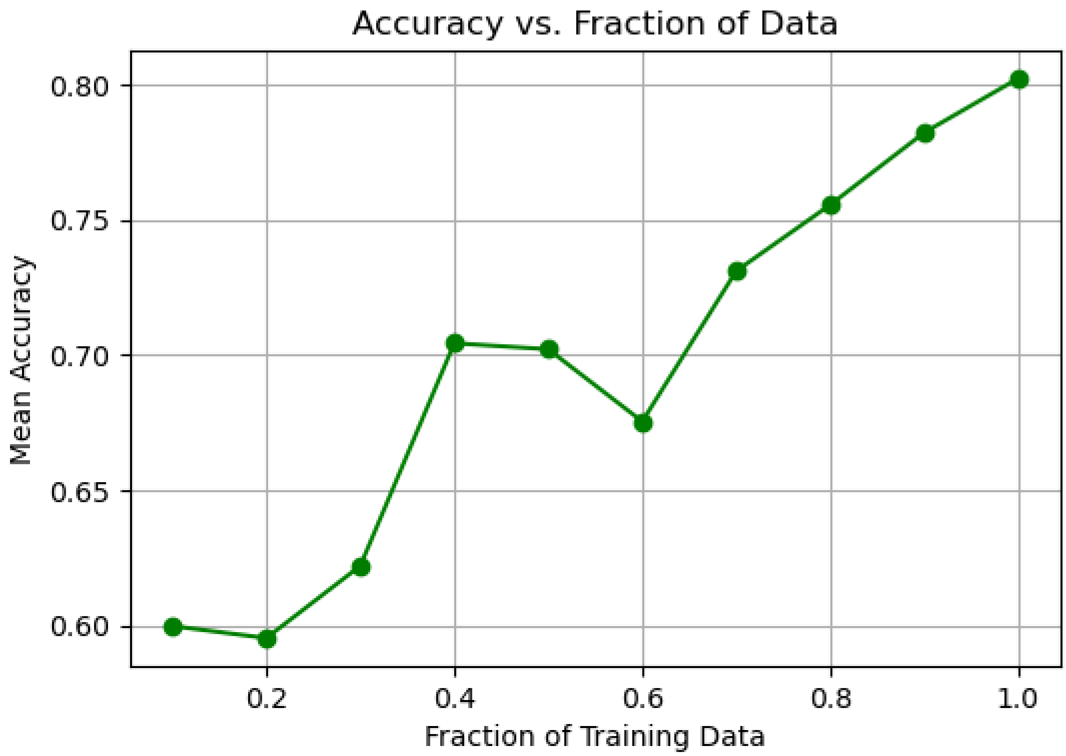


Std. Dev. of Accuracy vs. Fraction of Data
Std. Dev. of Accuracy
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data



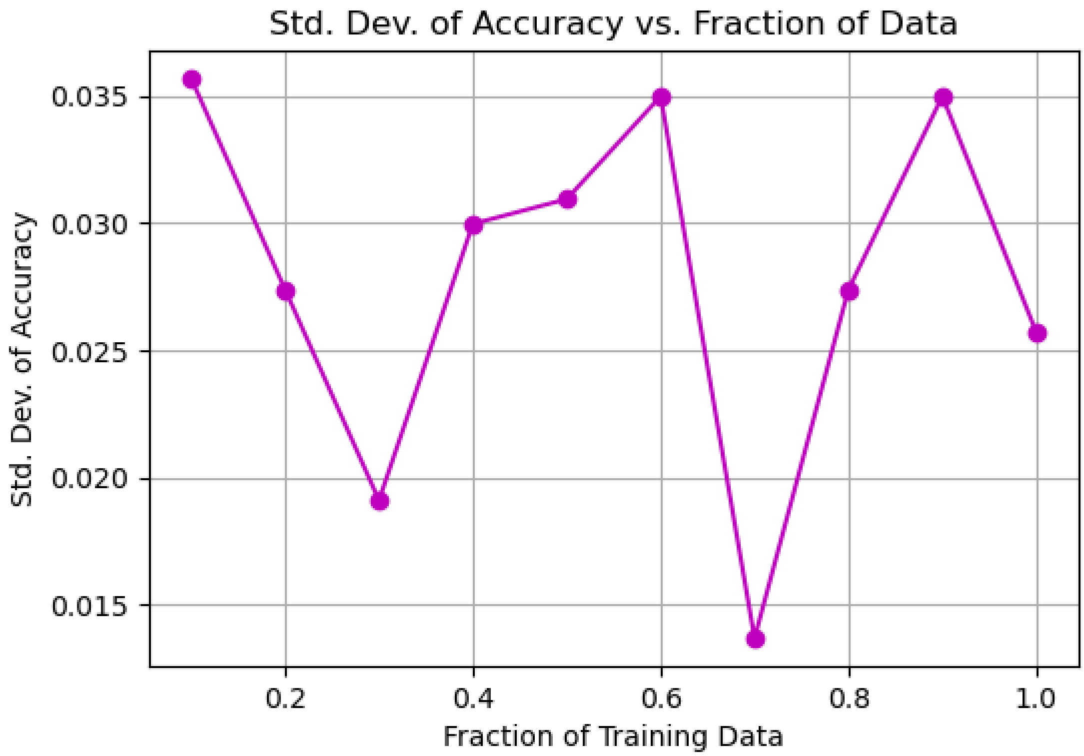
Training Time vs. Fraction of Data
Mean Training Time (s)
0.05
0.04
0.03
0.02
0.01
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data

nn_scratch:

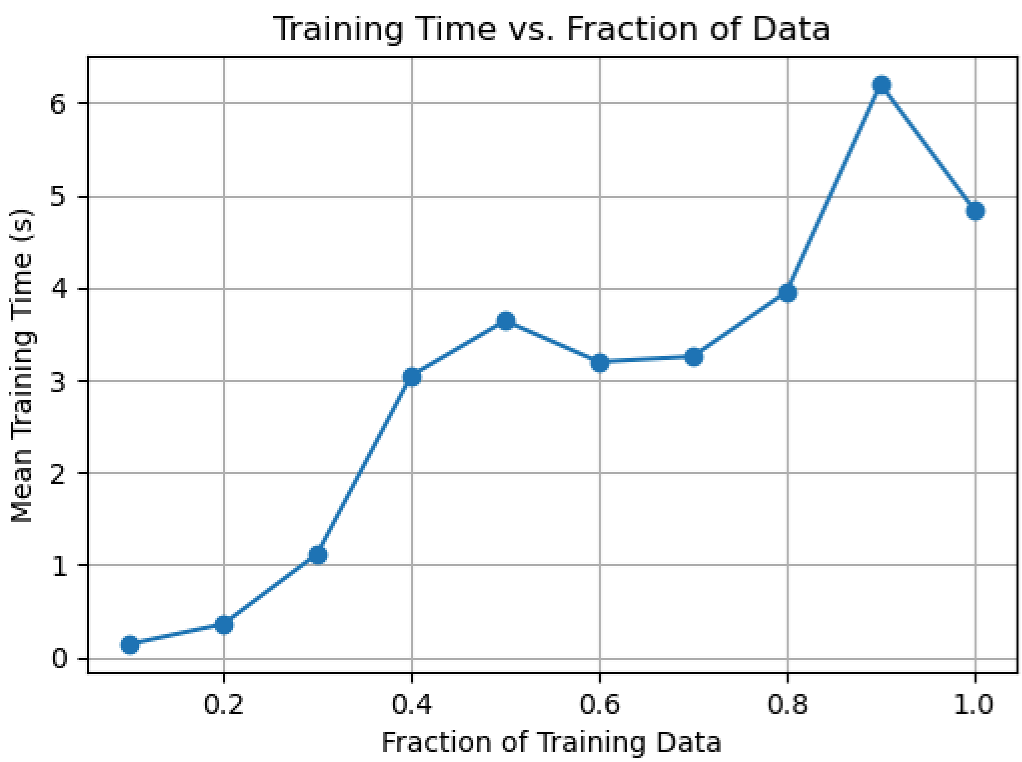


Accuracy vs. Fraction of Data
Mean Accuracy
0.80
0.75
0.70
0.65
0.60
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data

Error vs. Fraction of Data
Mean Error
0.40
0.35
0.30
0.25
0.20
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data

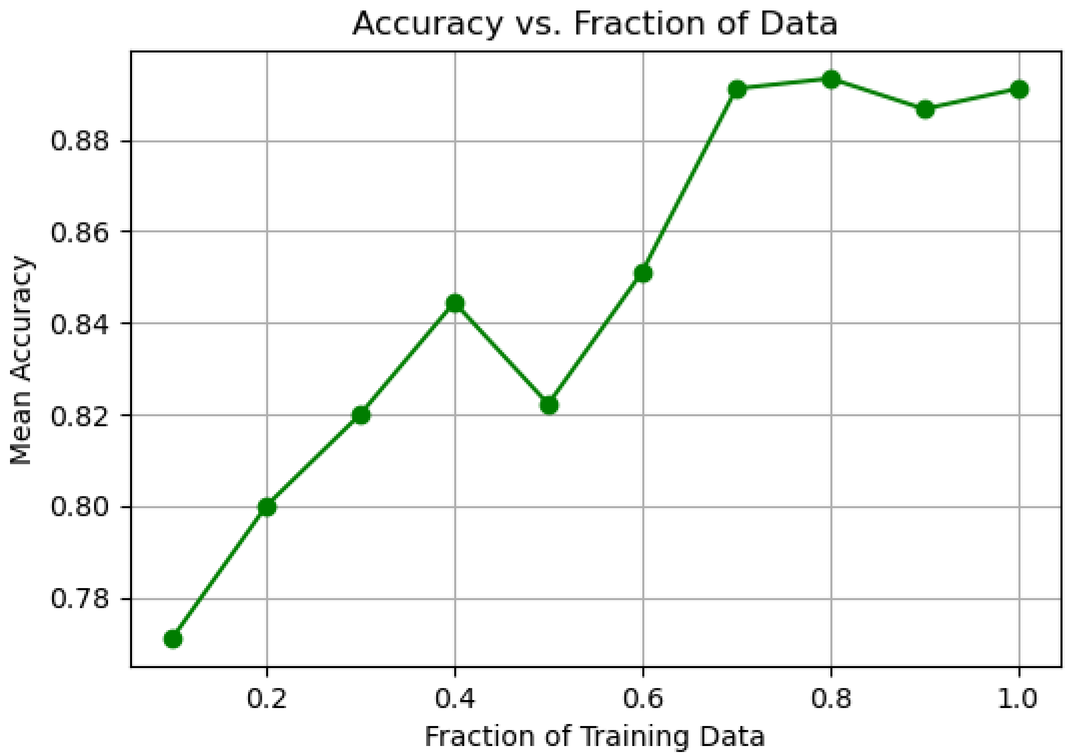


Std. Dev. of Accuracy vs. Fraction of Data
Std. Dev. of Accuracy
0.035
0.030
0.025
0.020
0.015
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data

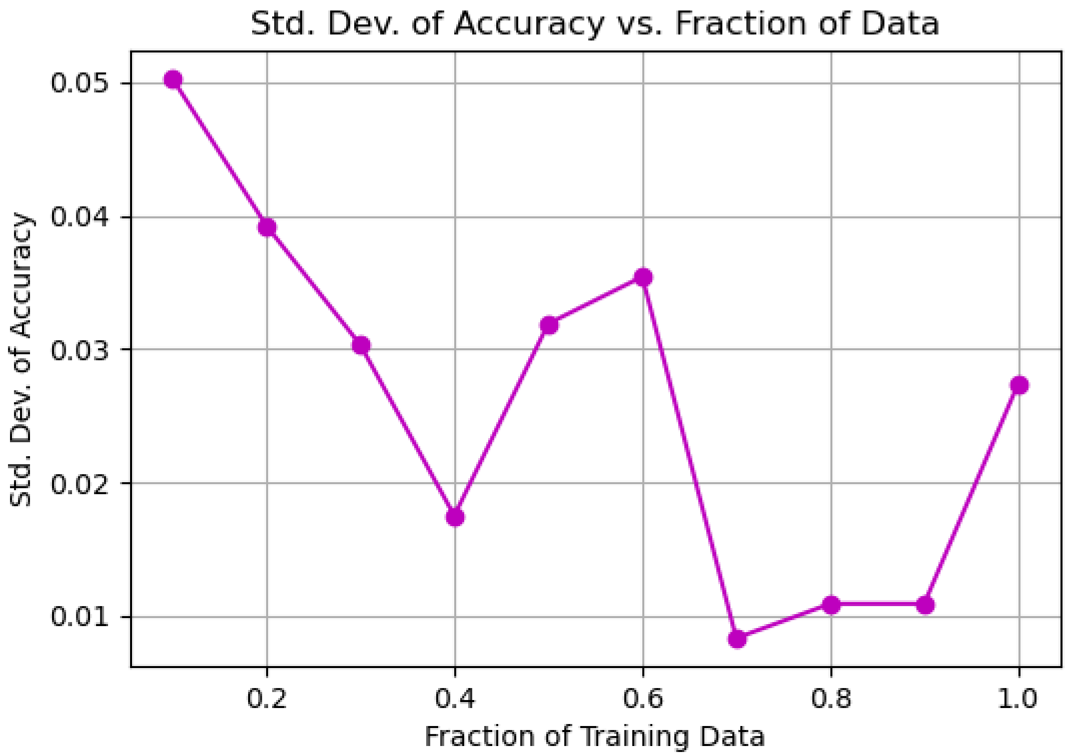


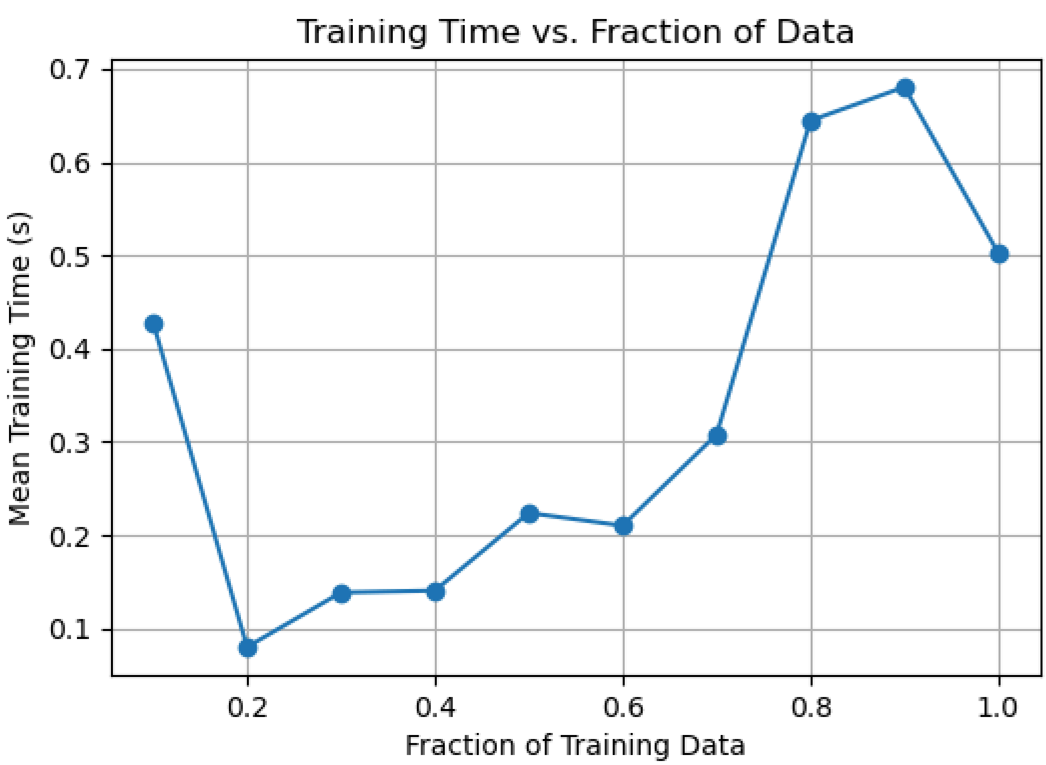
Training Time vs. Fraction of Data
Mean Training Time (s)
6
5
4
3
2
1
0
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data

nn_pytorch:


Accuracy vs. Fraction of Data
Mean Accuracy
0.78
0.80
0.82
0.84
0.86
0.88
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data


Error vs. Fraction of Data
Mean Error
0.12
0.14
0.16
0.18
0.20
0.22
0.2
0.4
0.6
0.8
1.0
Fraction of Training Data





Metric (@ 100 % train data)	**Perceptron (one-vs-all)**	**NN from scratch**	**NN (PyTorch)**
Accuracy	≈ 0.87	≈ 0.80	≈ 0.89 ▲ *best*
Std. dev. of accuracy	≈ 0.022 ▼ *best*	≈ 0.026	≈ 0.027
Training time (s)	≈ 0.06 s ▼ *fastest*	≈ 4.8 s	≈ 0.50 s

1 Accuracy (ACC)

- **PyTorch NN** again leads nearlt **0.90** once the model has access to the full training set!
- **Perceptron** is only ~2 pp lower (≈ 0.87) and shows a smooth upward trend as data grows.
- **Scratch NN** lags at ≈ 0.80, reflecting the heavier over-parameterization + noisier optimization on only two output classes.
  If we are interested in purely predictive power for face/non-face task, the PyTorch NN is best; the perceptron is surprisingly competitive; the NumPy scratch MLP lags behind

2 Stability (STD of accuracy across 3 runs)

- **Perceptron** shows the **least run-to-run variance** (~ 0.022) thanks to its simple deterministic update rule.
- **Scratch NN** hovers around 0.026.
- **PyTorch NN** is the noisiest (~ 0.027) — still small in absolute terms, but larger than the linear model because of Adam's stochastic updates and different mini-batch orders. This is expected.
  If our primary requirement is highly repeatable metrics, the perceptron classifer is safest. To treat PyTorch NN's noisy results perhaps we can experiment further with a fixed-seed.

3 Training time (TIME)

- **Perceptron** remains the fastest by orders of magnitude (~ 60 ms at full data).
- **PyTorch NN** is ~ 0.5 s — slower than the perceptron but *much* faster than the NumPy scratch network.
- **Scratch NN** costs ≈ 4–5 s per full training which is time inefficient when placed next to its competitors.
  Speed hierarchy is identical to the digits experiment: perceptron >> PyTorch_NN >> scratch_NN (order fastest >> slowest)

---

5. Conclude : Lessons Learned and Best Design Choices

After experimenting with all three implementations on both the digits and faces datasets, we can highlight some **lessons learned** and **best design decisions**:

1. **Activation Functions**
   - **ReLU** consistently outperforms sigmoid, mitigating vanishing gradients and allowing faster convergence.
2. **Weight Initialization**

- **He initialization** helps maintain stable forward/backward .
- Zero initialization (as in the perceptron) is simpler but may converge more slowly.

3. **Regularization**
    - **L2 weight decay** or `reg_lambda` helps avoid overfitting, especially in larger networks.
    - In PyTorch, setting `weight_decay` to a small value like 0.001 can be particularly beneficial for the faces dataset.
4. **Mini-Batch Training**
    - A batch size of 32 is a good *balance* of convergence speed and computational efficiency; very large batches can slow updates, and very small batches can cause higher variance in gradient estimates.
5. **Learning Rate**
    - Approx. lr=0.01 tends to work well.
    - Perceptron and scratch NN can be unstable with very large learning rates, whereas PyTorch's Adam optimizer is more robust.
6. **Choice of Algorithm**
    - The **One-vs-All Perceptron** is straightforward but often yields lower accuracy for complex tasks. It favorably gave best standard deviation across runs for the face classification task. Gave fastest time metrics for both classification tasks.
    - The **Scratch NN** offers full control. It yielded best standard deviation metrics for digits Multi-class classification task, impressively.
    - The **PyTorch NN** is more scalable, benefits from GPU acceleration, and usually converges faster. PyTorch gave the best accuracy metrics across both tasks.
7. **Scaling with data**:
    - all three models improve as the fraction of training data increases, but the perceptron's curve flattens earliest, confirming its limited capacity. That is, perceptron shows diminishing returns beyond a certain point.

Overall, **ReLU** + **He initialization** + **L2 regularization** + **mini-batch** + **Adam** together form a robust set of practices that I found were the best design choices when creating these neural networks. (I was shocked that the simple perceptron was successfully able to effectively complete both classification tasks.)