Final Project

Face and Digit Classification

Abid Azad (netID: aa2177, RUID: 202005452)

CS440, May 1st, 2024

Contents

1 Part 1 - Implement Two Classification Algorithms for Detecting Faces and Classifying Digits

1.1 Perceptron

* NOTE: Please Refer to the Perceptron Classifier Class with the faceDigitClassification.py script. *

1.1.1 Initialization:

The PerceptronClassifier class is initialized with parameters such as categories, dataType, and dataUsage.
Weights for each category are initialized using the Counter data structure.
Initializations also include setting up parameters like the learning rate and maximum number of iterations.

1.1.2 Training:

The train method is responsible for training the perceptron classifier.
It begins by checking if there are pre-learned weights available in a file.
If so, it loads these weights; otherwise, it initializes weights for each category and feature.
It then iterates through a maximum number of iterations, updating weights based on misclassifications.
For each training data point, it calculates the result for each category by taking the dot product of the weights and features and adding the bias term.
The predicted category is the one with the maximum result.

If the prediction is incorrect, it updates the weights.
It stops training if all data points are correctly classified or when it reaches the maximum number of iterations.

1.1.3 Classification:

The classify method is used to classify new data based on the learned weights.
It calculates the score for each category by taking the dot product of the weights and features of the input data and adding the bias term.
The category with the highest score is chosen as the predicted category for the input data.

1.1.4 Equations:

The calculation of the result for each category during training:

```
result_cat = weights_cat * data + weights_cat[0]
```

The update of weights when a misclassification occurs:

```
self.weights[predicted_cat] -= data
self.weights[predicted_cat][0] -= learn_rate
self.weights[int(train_labels[i])] += data
self.weights[predicted_cat][0] += learn_rate
```

`The calculation of the score for each category during classification:`

```
score_cat = self.weights_cat * pic + self.weights_cat[0]
```

This Python code implements a basic Perceptron algorithm for classification tasks, updating weights based on misclassifications until convergence or reaching a maximum number of iterations. The classification process then utilizes the learned weights to predict the category for new data points.

1.2 Two-layer Neural Network

1.2.1 Initialization

The `TwoLayerNeuralNetworkClassifier` class is initialized with the following parameters:

- `outClasses`: Number of output classes.
- `inSize`: Number of input features.
- `hidSize`: Number of neurons in the hidden layer.
- `outSize`: Number of neurons in the output layer.
- `dataSize`: Size of the dataset.
- `regularization_param`: Regularization parameter for preventing overfitting.
- `dataType`: Type of data being processed (e.g., faces or digits).
- `dataUsage`: Percentage of training data used for training.
- `useSavedLearnWeightData`: Flag indicating whether to use previously learned weights.

During initialization, various matrices and arrays are initialized:

- Activation matrices for input, hidden, and output layers.
- Weight matrices for input-to-hidden and hidden-to-output connections.
- Bias term for each data point.
- Weight change matrices to track changes during backpropagation.
- Random initialization of weights within a specified range.

1.2.2 Feed-forward Propagation

The `forward_propagate` method computes activations of the hidden and output layers:

$$\text{hidLayerInput} = \text{inWeights} \times \text{inActivation}$$

$$\text{hidActivation} = \text{sigmoid}(\text{hidLayerInput})$$

$$\text{outLayerInput} = \text{outWeights} \times \text{hidActivation}$$

$$\text{outActivation} = \text{sigmoid}(\text{outLayerInput})$$

The cost function and regularization term are computed based on the predicted output and ground truth labels.

1.2.3 Back-propagation

The `back_propagate` method computes gradients of weights during backpropagation:

$$\begin{aligned}
\text{outError} &= \text{outActivation} - \text{expected_output} \\
\text{hidError} &= \text{outWeights}[:, :-1]^T \cdot \text{outError} \times \text{dsigmoid}(\text{hidActivation}[:-1]) \\
\text{outChange} &= \frac{\text{outError} \cdot \text{hidActivation}^T}{\text{dataSize}} \\
\text{inChange} &= \frac{\text{hidError} \cdot \text{inActivation}^T}{\text{dataSize}}
\end{aligned}$$

1.2.4 Training

The `train` method trains the neural network using gradient descent optimization:

- Setting input activations and output truth values.
- Combining weights into a single vector.
- Applying gradient descent to minimize the cost function.
- Separating weights back into input and output weights.

1.2.5 Classification

The `classify` method classifies test data using the trained neural network:

- Computing activations for hidden and output layers.
- Predicting class labels based on the highest activation.

1.2.6 Gradient Descent

Gradient descent is an optimization algorithm used to minimize the cost function by iteratively moving in the direction of the steepest descent of the cost function. The weight updates are calculated using the gradient of the cost function with respect to the weights.

The gradient descent algorithm involves the following steps:

1. Initialize weights randomly or with predefined values.
2. Compute the cost function and its gradient with respect to the weights.
3. Update the weights in the opposite direction of the gradient by a certain step size (learning rate).
4. Repeat steps 2 and 3 until convergence or a maximum number of iterations is reached.

The equations for gradient descent in this implementation are as follows:

$$\text{gradient} = \text{back\_propagate}(\textit{weight\_vector})$$

$$\text{weight\_vector} \leftarrow \text{weight\_vector} - \text{learning\_rate} \times \text{gradient}$$

.

2 Part 2 - Feature Extraction

2.1 Feature Design

For the given two problems of classifying digits and faces, feature design plays a crucial role in capturing relevant information from the images. The code primarily relies on custom

feature extraction to design features for each problem:

1. **Custom Feature Extraction:**

   The code defines specific feature extraction functions, namely `digitFeatureExtractor` and `faceFeatureExtractor`, for each problem. These functions convert the raw pixel data of the images into binary features. For digits, each pixel's intensity is thresholded to 0 or 1, resulting in a binary representation of the digit image. Similarly, for faces, the same process is applied to extract binary features from each pixel. These binary features capture the presence or absence of information at each pixel position.

2.2 Feature Extraction Program

The feature extraction program in the code is responsible for converting the raw image data into feature vectors suitable for training the classifiers. It consists of the following components:

1. **Image Loading:**

   The code provides functions to load image data from files. Two sets of loading functions are available: `load_data_randomly` and `load_data`, along with corresponding label loading functions. These functions allow loading data either randomly or sequentially, depending on the specific use case. Random loading is particularly useful when selecting a subset of training data randomly, as specified by the user.

2. **Feature Extraction:**

   The feature extraction process involves applying the appropriate feature extraction function (`digitFeatureExtractor` or `faceFeatureExtractor`) to each image in the

dataset. These functions convert the raw pixel data into binary feature representations, capturing relevant information from the images. The code primarily uses custom feature extraction rather than using raw pixels directly as features.

3. **Classifier Training and Evaluation:**

   After feature extraction, the program proceeds to train and evaluate the classifiers using the extracted features. This includes initializing the classifier based on user input (Perceptron or Two-Layer Neural Network), training the classifier on the extracted features, validating the trained model on a validation set, and finally, testing the model's performance on a separate test set. The results of each step are printed to provide feedback on the training and evaluation processes.

In summary, the feature design and extraction process in the code primarily rely on custom feature extraction, which converts raw pixel data into binary representations suitable for classification tasks.

3 Part 3 - Training

3.1 Classifier Initialization

- The main function allows the user to select the type of classifier (Perceptron or Neural Network) based on their input.
- Based on the selected classifier type, appropriate classes (`PerceptronClassifier` or `TwoLayerNeuralNetworkClassifier`) are initialized.

3.2 Data Loading

- The main function loads the training, validation, and test datasets along with their corresponding labels.
- The `load_data_randomly` and `load_labels_randomly` functions are used to load data and labels randomly for training, ensuring that each percentage of training data usage receives a random subset of the training dataset.
- The Picture class represents individual images, and functions are provided to convert characters to integers.

3.3 Feature Extraction

- Feature extraction is performed using specified feature extractor functions (`faceFeatureExtractor` or `digitFeatureExtractor`) to convert raw image data into feature vectors.
- Features are extracted from the raw data for both training and validation/test datasets.

3.4 Training and Evaluation

- For each percentage of training data usage (from 10% to 100%):
  - The main function initializes the classifier.
  - It determines the size of the training dataset and generates a random order of indices corresponding to the data points in the training set, effectively shuffling the training data.
  - The training, validation, and test datasets are loaded based on the specified percentage, with the training data shuffled according to the random order to prevent

the model from learning patterns based on the order of the data.

- Features are extracted from the raw data.
- The classifier is trained using the training data.
- The trained model is validated on the validation dataset, and its performance is evaluated.
- Finally, the trained model is tested on the test dataset, and its performance is evaluated.

3.5 Weight Saving and Loading

- The trained weights of the classifier are saved to files after training to avoid retraining.
- If trained weights are available, they are loaded from files to avoid unnecessary retraining.

3.6 Output

- The main function prints relevant information such as training time, the number of correct predictions on the validation and test sets, and the corresponding accuracy for each percentage of training data usage.
- Results are presented as a function of the number of data points used for training, allowing for easy comparison across different percentages.

Note, by shuffling the training data before each training iteration, the main function ensures that the model encounters a diverse range of examples during training, which can

help prevent it from memorizing patterns based on the order of the data, leading to better generalization and performance of the trained model.

4 Part 4 - Comparison and Analysis

When executing the code and specifying both the classifier and data type (without employing pre-trained weights), we begin to observe outputs resembling the following:

```
Current Classifier Type: Perceptron
Training Data Used: 10.0%
Training Set Size: 45
Validation Set Size: 301
Test Set Size: 150
        Extracting features...Feature extraction complete.
        Training...Training complete.
        Training Time: 2.90 s
        Validating the Model...Validation complete.
                Correct Predictions on Validation Set: 181/301 (60.13%).
        Testing the Model...Testing complete.
                Correct Predictions on Test Set: 92/150 (61.33%).
```

Figure 1: Output of Perception Training and Classifying Face Data

We'll iterate through each training data usage set 10 times, generating plots illustrating the average training time against the number of data points used, and the average prediction error (with standard deviation) relative to the training data size. These visualizations will cover both classifiers and their respective data types, leveraging Matplotlib for the plotting process.

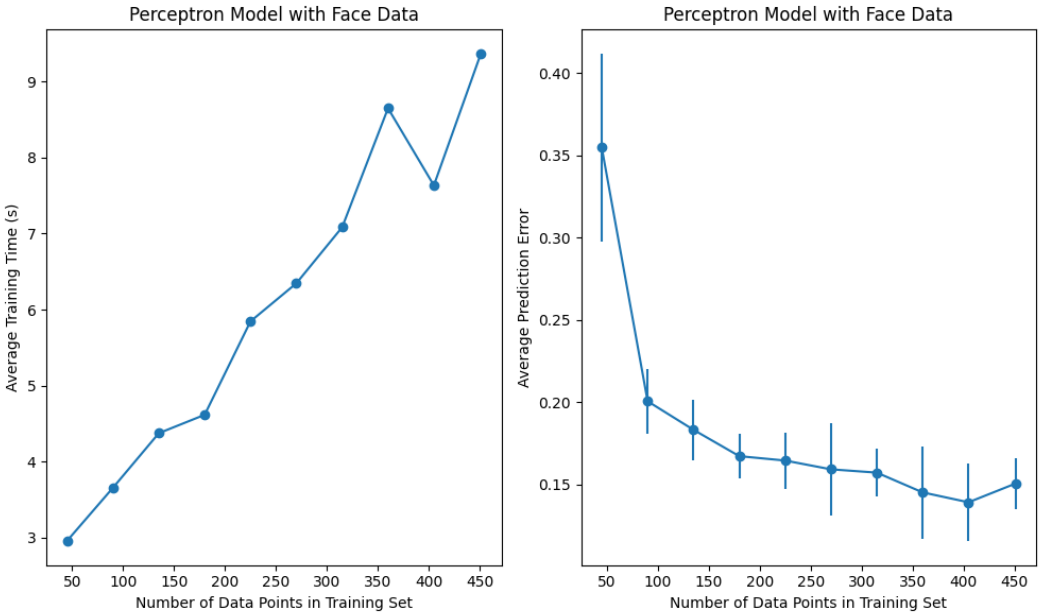

Figure 2: Accuracy/Training Time of Perception when using Face Data

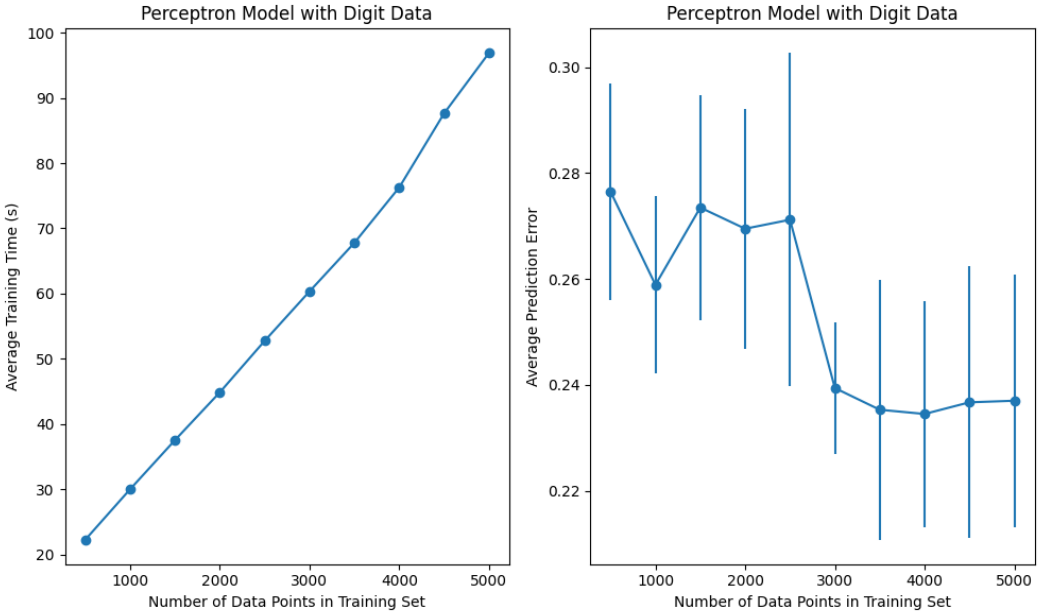

Figure 3: Accuracy/Training Time of Perception when using Digit Data

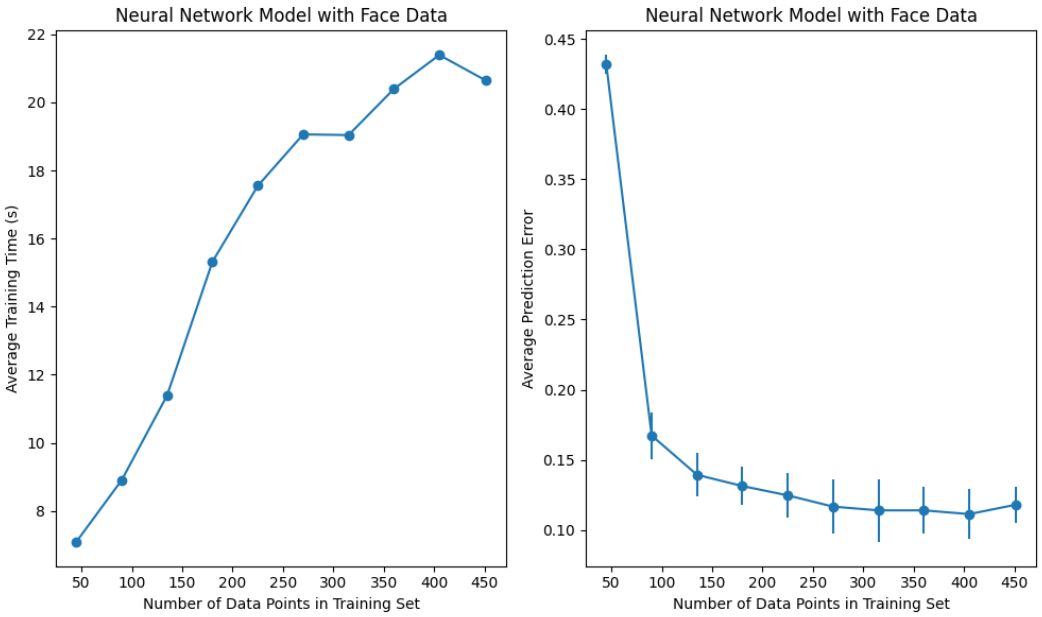

Figure 4: Accuracy/Training Time of Neural Network when using Face Data

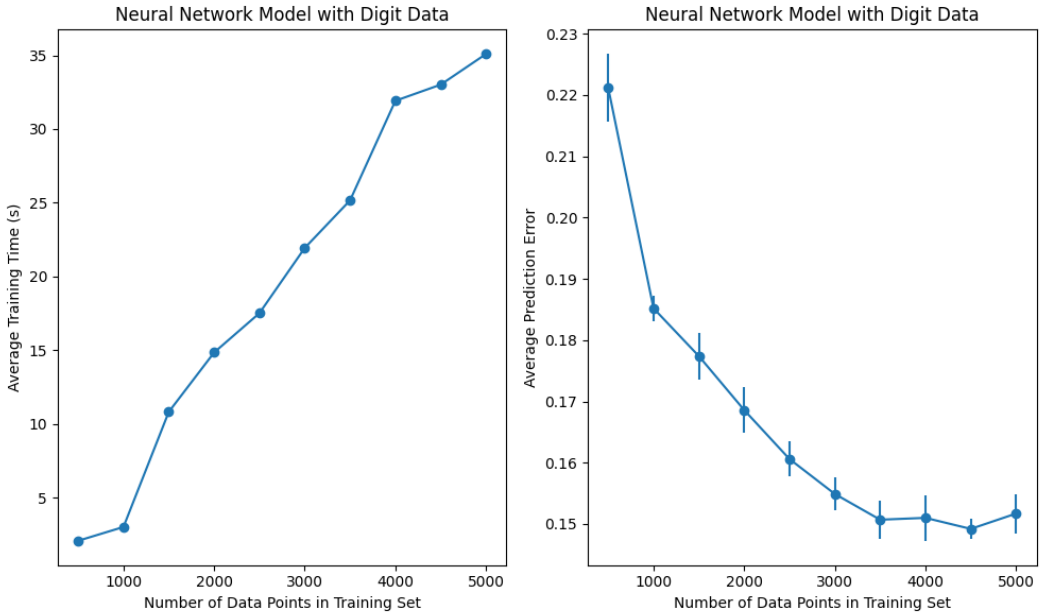

Figure 5: Accuracy/Training Time of Neural Network when using Digit Data

From the plots, several observations can be made:

- The Perceptron Model took a significant amount of time when training and classifying Digit Data, especially compared to the neural network model.
- The Perceptron Model did not perform as well with Digit Data compared to Face Data, with slightly higher prediction errors and even higher standard deviations.
- Conversely, the neural network model performs well with both Face Data and Digit Data, exhibiting consistently low prediction errors and standard deviations for both types of data.
- Although the training time of the neural network model for both data types is longer than the perceptron model with Face Data, they are still significantly shorter than the perceptron model's training time on Digit Data.

- Notably, for both models, the increase in training data size results in longer training times but leads to reduced prediction errors, demonstrating the effectiveness of more extensive training data.

5 Part 5 - Conclusion

5.1 Performance Comparison Between Classifiers and Other Observations:

1. **Performance Comparison Between Classifiers**:
    - The Perceptron model took significantly longer when training and classifying digit data compared to the neural network model.
    - The Perceptron model showed inferior performance with digit data compared to face data, as indicated by higher prediction errors and standard deviations.
    - Conversely, the neural network model performs well with both Face Data and Digit Data, exhibiting consistently low prediction errors and standard deviations for both types of data.
    - Although the training time of the neural network model is longer than the perceptron model with Face Data, it is still significantly shorter than the perceptron model's training time on Digit Data.
    - Overall, the neural network model tended to outperform the Perceptron model, especially when dealing with complex datasets like digit recognition.
2. **Effect of Training Data Size**:

- Increasing the size of the training data generally led to longer training times for both classifiers.
- With larger training datasets, the prediction errors tended to decrease for both classifiers, indicating improved performance with more data.

3. **Utilization of Saved Learned Weights**:
   - The option to utilize saved learned weights from previous training sessions was implemented, reducing the need for retraining the model from scratch.
   - Using saved weights significantly reduced training time for subsequent runs, especially useful for iterative experimentation and testing different configurations.
4. **Hardware Impact on Performance**:
   - Performance could have been impacted by the hardware specifications of the desktop where the code was executed.
   - Factors such as CPU speed, memory bandwidth, and GPU capabilities (if applicable) can affect the execution time of machine learning algorithms.
   - Limited computational resources may result in longer training times, especially for complex models or large datasets.

5.2 Understanding Differences in Performance Between Classifiers on Different Data

The differences in performance between the Perceptron and Neural Network classifiers on digit and face data can be attributed to several factors:

1. **Complexity of Data Representation**:

    - Digit data typically consists of grayscale images of handwritten digits, which have clear and distinct patterns that can be effectively learned by neural networks. The features in digit data are well-defined, such as edges, corners, and strokes, making them suitable for deep learning models like neural networks.
    - On the other hand, face data is more complex due to variations in lighting conditions, facial expressions, poses, and occlusions. Extracting meaningful features from face data requires more sophisticated techniques, and the patterns may not be as easily separable as in digit data.

2. **Model Capacity**:

    - Neural networks, especially deep neural networks with multiple layers, have a higher capacity to learn complex patterns and representations from data compared to the simple perceptron model. The multi-layer architecture of neural networks allows them to capture hierarchical features and relationships, which is beneficial for tasks involving intricate data like digit recognition.
    - Perceptrons are simple linear classifiers with limited expressive power. They can only learn linear decision boundaries and are less effective at capturing non-linear relationships present in complex data like face images.

3. **Feature Extraction**:

    - Neural networks are capable of automatically learning relevant features from raw data during training, alleviating the need for handcrafted feature engineer-

ing. This ability allows them to adapt to the unique characteristics of different datasets, including digit and face data.

- Perceptrons, on the other hand, rely on manually engineered features or simple feature extraction techniques. They may struggle to extract discriminative features from complex datasets like face images, leading to inferior performance compared to neural networks.

4. **Representation Power**:

   - Neural networks can represent highly nonlinear and intricate decision boundaries, enabling them to model the complex relationships present in digit data more effectively. They can capture both global and local patterns in the data, making them well-suited for tasks with diverse and intricate patterns.
   - Perceptrons, being linear classifiers, can only represent linear decision boundaries. They may struggle to separate the overlapping classes and capture the subtle variations present in face data, resulting in lower accuracy compared to neural networks.

5.3 Possible Improvements and Concluding Remarks:

- Utilizing more powerful hardware, such as GPUs or cloud computing services, can significantly improve performance by speeding up training and inference times.
- Implementing parallel processing techniques or distributed computing frameworks can further enhance scalability and reduce training times.

- Optimizing the code for efficiency, such as vectorization and parallelization, can lead to faster execution and improved resource utilization.
- Experimenting with different machine learning algorithms, hyperparameters, and feature engineering techniques can help achieve better performance and accuracy.
- While the results provide valuable insights into the performance of the classifiers on different datasets, optimizing hardware resources and code efficiency can further enhance the capabilities and scalability of the machine learning system. Additionally, understanding the characteristics of the data and selecting appropriate algorithms based on their strengths and weaknesses is crucial for achieving optimal performance.

6 References

1. Project 5: Classification. *inst.eecs.berkeley.edu/ ~cs188/sp11/projects/classification/classification.html.*
2. "Lecture 7: Two Layer Artificial Neural Networks."*www.sfu.ca/iat813/lectures/lecture7.html.*
3. Miranda, LJ. "Implementing a Two-Layer Neural Network from Scratch." *ljvmiranda921.github.io/ notebook/2017/02/17/artificial-neural-networks/*
4. Perceptrons. *www.w3schools.com/ai/ai_perceptrons.asp.*
5. Aflak, Omar. "Neural Network from Scratch in Python." *Medium, Towards Data Science*, 24 May 2021. *towardsdatascience.com/math-neural-network-from-scratch-in-python-d6da9f29ce65.*

6. Contribution of ChatGPT in assisting the development of the user input for model/data selection as well as plot generation.