Character Recognition Project Report

Introduction to Artificial Intelligence

Constança Rocha (20241758)
Maiara Almada (20241723)
Neda Razavi (20241896)
Tiago Silveira (20241698)

Data Science Bachelor
NOVA Information Management School
June 2025

Contents

1 Introduction

Character recognition is a fundamental task in the field of pattern recognition and artificial intelligence (AI), with applications ranging from optical character recognition (OCR) to intelligent document processing and assistive technologies. This project explores the application of two classical machine learning algorithms, the **Perceptron** and the **Widrow-Hoff(δ) rule**, for the task of handwritten character classification.

Using a training set of white on black letter images, we implemented, trained and evaluated both models for their effectiveness in recognizing characters from pixel data. In addition to algorithmic development, we also built an interactive graphical user interface (GUI) using **Streamlit** to facilitate model training, testing and comparing without requiring coding knowledge.

This report details the theoretical background, implementation processes and practical considerations of each model, while also highlighting their strengths and limitations within the context of character classification.

2 Training Set

In order to train both **the Perceptron** and **the Widrow-Hoff** models, we used the data set available on (Patel, 2018) in CSV format. This data set was converted to `.png` using a python script made available by the author.

Since the data set had `10,000 +` images for each letter and our algorithms were unable to accommodate that amount of images in a reasonable time, we selected the first `50` images for each letter.

We then used an additional python script to convert the images to `NumPy` (Harris et al., 2020) 20x20 arrays while guaranteeing all pixels are either black or white. These arrays are then stored in an `.npz` file.

3 Perceptron Rule

3.1 Percepton

The perceptron is the simplest form of an artificial neural network It consists of a single layer of input nodes connected to one output, and it performs binary classification by computing a weighted sum of inputs followed by an activation function.

Originally introduced by Frank Rosenblatt, the perceptron forms the basis for more complex ANN architectures. In this project, it is used to classify character images by learning patterns in their pixel data.

3.2 Binary Perceptron Model

The `BinaryPerceptron` class is a simple neural model that performs binary classification, determining whether an input belongs to a specific class (e.g., "A") or not. It serves as the core component in the multiclass system by acting as a one-vs-all classifier.

The class is initialized with the input size (number of features per sample), a learning rate, and the number of training epochs. With key internal attributes as:

- `self.weights`: a `NumPy` array of weights, initialized with small random values for each input feature.
- `self.bias`: a single bias value, initialized to zero.
- `self.lr`: the learning rate that controls how much the model adjusts on each error.
- `self.epochs`: the number of times the model will iterate through the entire training dataset.

3.2.1 Methods Descriptions

- The `self.activation()` function implements a unit step function that returns 1 if the input is greater than or equal to zero, and -1 otherwise. This is used to make the final binary decision.
- The `raw_output()` function computes the linear combination of inputs and weights plus bias. It does not apply any activation — it's used to analyze the model's internal score before classification.
- The `predict()` function applies the activation function to the output from `self.raw_output(x)` to produce the final prediction of either 1 or -1.

The binary perceptron learns to distinguish one specific class from all others using a step-based activation and weight update rule.

3.3 Multiclass Perceptron Model

The `MultiClassPerceptron` class builds on the `BinaryPerceptron` to handle multiclass classification (e.g., 26 letters A–Z). It uses the one-vs-all approach, where a separate binary perceptron is trained for each class.

The class is initialized with:

- `n_classes`: the total number of classes (e.g., 26 for alphabet letters),
- `input_size`: the number of features per input,
- `learning_rate`: determines how much the model updates its weights during learning,
- `epochs`: specify how many times the model iterates over the training data

It also contains an internal list called self.models, which stores one `BinaryPerceptron` per class.

3.3.1 Method Descriptions

- The constructor initializes one binary perceptron for each class and stores them in the `self.models` list.

- The `self.train()` function loops over all classes. For each class, it extracts the corresponding column from the one-hot encoded label matrix and uses it to train the corresponding binary perceptron. Each model learns to answer the question: "Is this sample class X?"
- The `self.predict()` function returns the final classification for each input. For each sample, it computes the raw output score from all trained perceptrons. The index of the perceptron with the highest score is returned as the predicted class.
- The `self.save()` function stores the trained weights and biases of all models into a `.npz` file. This allows the model to be reloaded and used without retraining.
- The `self.load()` class method reconstructs a previously saved model. It reads the weights and biases from file, rebuilds the binary perceptrons, and restores them to their trained state.

3.4 Conclusion

The perceptron-based system developed in this project highlights key artificial neural network principles using a simple and interpretable structure. The binary perceptron serves as a core unit, applying a linear decision rule with step activation to separate two classes. This approach was extended into a multiclass model using a one-vs-all strategy, where each perceptron detects a specific letter. Training was influenced by parameters like `learning rate` and `epochs`, and letters with similar pixel patterns were more challenging to classify. Input normalization proved essential for stable learning. Despite its simplicity, the model effectively demonstrated how artificial neural networks can learn through iterative weight updates and lays a foundation for more advanced architectures.

4 Widrow-Hoff Rule (δ rule)

The Widrow-Hoff learning rule, also known as delta rule (Stone, 1986) is a linear (Widrow and Hoff, 1960) supervised learning method (Lytton, 2002) That from a set of inputs and targets can distinguish between different classes. In This project it was used for multiclass character recognition using a dataset of black and white handwritten letters.

4.1 Theoretical Background

This algorithm is trained by minimizing the difference between the predicted outputs and the actual target (error), and updates the weights to reduce it over time. According to the following formula:

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} + \alpha \cdot (\mathbf{t} - \mathbf{y}) \cdot \mathbf{x}^T$$

where:

- α: Learning rate
- $\mathbf{t}$: Target output vector

- $\mathbf{y}$: Predicted output vector ($\mathbf{y} = \mathbf{x} \cdot \mathbf{w}$)
- $\mathbf{x}$: Input vector with appended bias

The prediction is given by the following formula:

$$\vec{y}_j = \sum x_i w_{\text{ij}}$$

where:

- $\vec{y}_j$: Output Vector
- x: Input vector
- w: Weight matrix

4.2 Implementation Details

The Widrow-Hoff rule is implemented in a Python class `WidrowHoff`, with the following attributes:

- `self.lr`: Corresponds to the learning rate(α) at which the algorithm will be trained;
- `self.variable`: Boolean variable that activates or not the learning rate decay (`True` $\rightarrow$ Active; `False` $\rightarrow$ Inactive);
- `self.epochs`: Sets the number of training iterations a bigger number of epochs, normally, corresponds to a better accuracy but a bigger time cost.
- `self.X`: Input vector with bias term added to increase model flexibility
- `self.T`: Target vector, a 26 bit vector where only the bit corresponding to the letter represented in the input is '1';
- `self.input_size` and `self.output_size`: Are the size of, respectively, the input and output vector;
- `self.weights`: Weight matrix. It is initialized on random weights and then is adjusted during the training cycle

To implement and run the model, the following methods were implemented:

- `train`: Iterates over epochs and samples, updating weights using the outer product. If `self.variable` = True, updates decreases the learning rate by 5% every 1/10 of the total epochs.
- `predict`: Returns the dot product of the weights matrix and the input vector(with bias).
- **`save` and `load`:** Allows saving and loading the trained model parameters via `np.savez()` to a `.npz` file, allowing the user not to have to re-train the model, with already ran configurations.

4.3 Advantages

- Simplicity of Implementation
- Simple and intuitive update rule: The algorithm updates weights using a simple, linear formula
- Supports Multi-Output learning: Can handle multi-class problems by outputting a vector of class scores, to which `np.argmax` must be applied to select the highest value.

4.4 Limitations

- Can only learn linear relationships between inputs and outputs.
- Too sensitive to learning rate:

  Too high a learning rate α. $\rightarrow$ can cause divergence.

  Too low $\rightarrow$ very slow convergence.
- Requires a normalized input (input features must be scaled similarly).

5 Comparison Between Perceptron and Widrow-Hoff rule

The Widrow-Hoff and Perceptron algorithms are both linear learning rules used for classification, but they differ in several key aspects:

Learning Objective

- Widrow-Hoff minimizes the mean squared error (MSE) between the predicted output and the target.
- Perceptron focuses on classification accuracy, updating weights only on misclassifications.

Output Type

- Widrow-Hoff produces continuous-valued outputs (can be negative).
- Perceptron produces binary outputs after applying a step activation function.

Convergence Behavior

- Widrow-Hoff converges toward the solution that minimizes error, even if data is not linearly separable.
- Perceptron converges only if the data is linearly separable; otherwise, it may never stabilize.

Multi-Class Strategy

- Widrow-Hoff directly outputs a vector (one dimension per class).

- Perceptron requires one binary classifier per class (one-vs-all setup).

Sensitivity to Outliers

- Widrow-Hoff is more sensitive to large errors or outliers, which can dominate weight updates.
- Perceptron is more robust to outliers, since it only reacts to misclassifications, not the error magnitude.

Learning Signal

- Widrow-Hoff uses a proportional update: the greater the error, the larger the update.
- Perceptron uses a fixed step based only on correct or incorrect prediction.

6 Graphical User Interface (GUI)

We developed a web-based graphical user interface (GUI) using **Streamlit** to provide an intuitive and interactive user experience. **Streamlit** is an open-source Python library that facilitates the creation of web applications. The GUI is essential to allow users to experiment with the Perceptron and Widrow-Hoff learning algorithms for character classification without needing to modify the code manually. The interface supports model training, input drawing, drawing classification, and model management in an interactive manner.

6.1 Interface Overview

The interface is organized into two main areas:

- **Sidebar**: For configuration and model control.
- **Main Panel**: For drawing input, triggering classification, training, and displaying results.

This layout guarantees ease of visualization and functionality, improving the interaction between the user and the models.

6.2 Sidebar: Configuration and Model Management

The sidebar offers several interactive widgets for configuring the ANN training and model behavior:

- **Algorithm Selection** (`selectbox`):
  - Allows users to choose between two implemented algorithms:
    - **Perceptron**: A classic binary classifier extended to multiple classes.
    - **Widrow-Hoff**: Also known as the delta rule, which updates weights based on the error signal.

- **Learning Rate (`slider`)**:
  - The user can select a learning rate from 0.001 to 0.01, with a default value of 0.005 and a precision step of 0.001.
  - For the Widrow-Hoff algorithm, this value directly affects how quickly the model adapts during training.
- **Variable Learning Rate (`toggle`)**:
  - If enabled, the learning rate is adapted during training (specific adaptation logic is defined in the `WidrowHoff` class).
  - This allows for finer convergence behavior in long training sessions.
- **Training Epochs (`slider`)**:
  - Allows selection of training epochs in a logarithmic scale from 1 to 5 with a step size of 1. The default is 2.
  - A higher number of epochs generally results in better convergence, assuming the learning rate is properly set.
- **Model Management Buttons**:
  - *Load Saved Model*:
    - Attempts to load a saved model using a consistent naming convention based on selected algorithm, learning rate, epochs, and the variable rate flag.
    - Feedback is provided through success or error messages depending on the outcome.
  - *Save Model*:
    - Enables saving the trained model into a file using NumPy's `.npz` format.
    - File naming includes key configuration metadata to facilitate reusability.

These sidebar components allow rapid reconfiguration of training conditions and switching between pre-trained models, enhancing experimentation and learning.

6.3 Main Panel: Drawing Input, Training, and Classification

The central panel handles all user interaction related to input, model execution, and results display:

- **Model Training (`Train Model` Button)**:
  - When clicked, this button triggers model training using the selected algorithm and parameters.
  - The model is initialized and trained using either the Perceptron or Widrow-Hoff implementation from the backend module (`main.py`).
  - The trained model is stored in `st.session_state` to persist across user interactions and button presses.

- **Drawing Canvas (`st_canvas`):**
  - Provided by the `streamlit-drawable-canvas` plugin, it offers a freehand drawing area for users to input characters.
  - Canvas specifications:
    * Size: 280×280 pixels
    * Stroke Color: White
    * Background: Black
    * Stroke Width: 15 pixels
  - The canvas returns a NumPy image array, which is preprocessed before classification.
- **Image Preprocessing and Classification (`Classify Drawing` Button):**
  - When the user submits a drawing:
    1. The image is converted to white on black using OpenCV.
    2. It is resized to 20×20 pixels to match the training data format.
    3. The flattened image vector is normalized (values in the range $[0, 1]$).
  - This processed input is then passed to the trained model's `predict()` function.
  - The model outputs a score vector representing confidence for each class.
  - The class with the highest score is selected as the prediction.
  - The corresponding character label is retrieved from `main.letters_list` and displayed.
- **Output Vector (`Output Vector` Button):**
  - When pressed, it shows the full output vector (prediction scores for all possible character classes) in a tabular format using a `Pandas` (pandas development team, 2020) Data Frame.
  - This is only available for the Widrow-Hoff model.
  - Each value is rounded to three decimal places to improve readability.
  - A snowfall animation (`st.snow()`) is triggered to make the result visualization more engaging.

6.4 User Experience and Error Handling

- The interface actively checks whether a model is trained before allowing classification.
- Clear feedback messages (warnings, success, or errors) are used to guide user actions.
- Model states are preserved using `st.session_state`, allowing smooth interaction across different stages (e.g., training, classifying, saving).
- A consistent naming convention for model files simplifies saving and loading models.

6.5 Design Rationale

This GUI was designed with the following goals in mind:

- **Modularity**: Separate training, configuration and classification concerns to allow focused interaction.
- **Ease of Use**: Enable non-experts to explore ANN behavior through drawing and sliders without needing to understand the codebase.
- **Responsiveness**: Provide immediate feedback on actions such as training, loading models or predicting input.
- **Scalability**: Easily extensible to support additional algorithms, characters or preprocessing techniques.

6.6 Conclusion

The developed graphical user interface (GUI) creates a harmonic balance between complex neural network algorithms and user-friendly interaction. It enables users to experiment with Perceptron and Widrow-Hoff learning models without requiring any kind of programming knowledge. With Streamlit's interactive components such as sliders, buttons and a drawable canvas, the interface allows for intuitive configuration, training and classification of handwritten characters. Error handling and state management ensure a satisfactory user experience, while the scalable design provides a solid foundation for future enhancements.

Overall, this GUI demonstrates how thoughtful interface design can make machine learning techniques accessible and engaging for a large audience.

7 Conclusion

This project demonstrated the practical application of foundational artificial neural network models, the **Perceptron** and the **Widrow-Hoff(δ) rule**, for handwritten character recognition. Through the development and comparison of both algorithms, we gained insights into their learning behaviors, capabilities and trade-offs.

While the **Perceptron** offers simplicity and robustness in binary settings, its extension to multi-class problems requires additional complexity. The **Widrow-Hoff rule**, with its continuous output and mean squared error minimization, provided a more flexible alternative with greater sensitivity to learning rate and input scaling. The GUI enabled user-friendly experimentation and visualization, making AI techniques more accessible and engaging.

Overall, this project served as a comprehensive learning experience in building, training, and deploying neural-based models for real-world classification tasks.

References

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., del Río, J. F., Wiebe, M., Peterson, P., . . . Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*(7825), 357–362. https://doi.org/10.1038/s41586-020-2649-2

Lytton, W. W. (2002). Supervised learning: Delta rule and back-propagation. In *From computer to brain: Foundations of computational neuroscience* (pp. 141–162). Springer New York. https://doi.org/10.1007/0-387-22733-4_9

pandas development team, T. (2020, February). *Pandas-dev/pandas: Pandas* (Version latest). Zenodo. https://doi.org/10.5281/zenodo.3509134

Patel, S. (2018). A-z handwritten alphabets in .csv format [visited on 2025-06-01]. https://www.kaggle.com/datasets/sachinpatel21/az-handwritten-alphabets-in-csv-format

Stone, G. O. (1986). An analysis of the delta rule and the learning of statistical associations.

Widrow, B., & Hoff, M. E. (1960). " adaptative switching circuits. *IRE WESCON Convention Record*, 96–104.