Perceptron: A method of updating knowledge through Gradient Descent

Yang Zhili
Department of Statistics and Data Science
Southern University of Science and Technology
12112711@mail.sustech.edu.cn

Abstract

This task involves modifying a python numpy-based simple/multi-layer perceptron model and its training function to allow for both batch gradient descent and stochastic gradient descent. The simple/multi-layer perceptron is trained on a synthetic two-dimensional dataset generated using $make\_moons$ from scikit-learn. The logs from Part_2 and Part_3 are stored in $training\_log.csv$ respectly. Code is available on `https://github.com/radicalyyyahaha/24DL/tree/main/Assigment1`

1 Introduction

The perceptron is one of the simplest types of artificial neural networks and forms the basis for more complex deep learning models. It was introduced by Frank Rosenblatt in 1957 as a binary classifier that could make decisions by learning from examples. The perceptron algorithm is a type of supervised learning, where the model learns from labeled data to make predictions. Before the perceptron, people always use statistical method to make prediction through existing data, the most famous algorithm must be Linear regression.

Linear regression is a technique used to predict a continuous output by finding the best-fitting linear equation that minimizes the error between the predicted and actual values. Its objective is to minimize the sum of squared errors between predictions and true values, which involves gradient descent or analytical solutions. Granted, this is a fairly straightforward idea, but unfortunately, everything in the world is not a simple linear relationship.

Then, generalized linear model was proposed. GLMs extend linear models to situations where the dependent variable does not necessarily follow a normal distribution. Instead, they allow the output to follow any distribution from the exponential family (e.g., binomial, Poisson, Gaussian). Generalized linear models, such as logistic regression, offer a more robust and flexible framework for classification and regression tasks, making them more broadly applicable in real-world scenarios where the data may not be perfectly linearly separable. In essence, the perceptron is a special case of linear classifiers, with its simplicity and limited flexibility.

The perceptron, on the other hand, is used for binary classification, where the goal is to classify data points into one of two classes. Instead of minimizing a continuous error function, the perceptron makes hard decisions by determining which side of a hyperplane the data points lie on, and it only updates the weights when a misclassification occurs.

A perceptron has several key characteristics that define its operation: The first is a binary classifier, meaning it can only classify inputs into two categories. It uses a linear decision boundary (a hyperplane) to separate the data into these categories. The second is Threshold Activation Function, the perceptron uses a step function as the activation function. If the weighted sum of inputs exceeds a certain threshold, the perceptron outputs one class; otherwise, it outputs the other class. The third is

Online Learning, the perceptron algorithm processes data points one at a time. After each data point, it updates its weights based on whether the current prediction is correct or incorrect. The fourth is Linearly Separable Data, the perceptron can only find a solution if the data is linearly separablei.e., if a linear hyperplane can completely separate the two classes. If the data is not linearly separable, the perceptron will not converge.

The perceptron is defined by a set of weights $\mathbf{w} = (w_1, w_2, , w_n)$, which correspond to the input features, and a bias term b . The output of the perceptron is determined as follows:

1. **Input Representation:** The input consists of feature values $\mathbf{x} = (x_1, x_2, , x_n)$, where each x_i represents a feature of the input data.

2. **Weighted Sum of Inputs:** The perceptron computes the weighted sum of the inputs as:

$$z = \mathbf{w} \cdot \mathbf{x} + b = \sum_{i=1}^{n} w_i x_i + b$$

Here, z represents the linear combination of the input features and weights.

3. **Activation Function:** The step function (or Heaviside function) is used to produce the binary output:

$$\hat{y} = 1, z > 0$$
$$\hat{y} = 0, z \leq 0$$

This function maps the weighted sum to either class 1 or 0 based on whether the sum is positive or negative.

4. **Learning Algorithm:** The perceptron adjusts its weights during the training process using the following rule:

$$w_i \leftarrow w_i + \Delta w_i$$

where $\Delta w_i = \eta(y - \hat{y})\ x_i$ is the update for the weight w_i , η is the learning rate, y is the actual label, and $\hat{y}$ is the predicted label. The weights are only updated when the perceptron makes an incorrect prediction. If the prediction is correct, no change is made to the weights. If the prediction is incorrect, the weights are updated to reduce the error for that specific data point.

2 Method

2.1 Part_1

In this problem, we aim to train a single-layer Perceptron to classify inputs into one of two categories, labeled as 1 and -1. The Perceptron is trained using a simple algorithm that adjusts its weights based on classification errors. This process continues until the model successfully classifies the entire training set or reaches a pre-set number of iterations.

We start by initializing the Perceptron with a set of inputs $\mathbf{x} = [x_1, x_2, \ldots, x_n]$, where each input is a vector with n features (e.g., x_1 and x_2). The Perceptron is initialized with random small weights $mathbfw = [w_0, w_1, w_2, \ldots, w_n]$, where w_0 corresponds to the bias term. Bias can be treated as an extra input $x_0 = 1$, allowing us to incorporate it into the weight vector.

During each epoch, the Perceptron makes predictions for each training example based on the current weights. For a given input vector $\mathbf{x}$, the Perceptron computes the weighted sum z of the input features:

$$z = \mathbf{w} \cdot \mathbf{x} + b$$

This can be rewritten as:

$$z = \mathbf{w} \cdot \mathbf{x\prime} = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$$

where $\mathbf{x\prime} = [1, x_1, x_2, \ldots, x_n]$ and w_0 is the bias term.

The output of the Perceptron is determined by applying a step activation function, then the Perceptron will output 1 if the weighted sum z is non-negative, and -1 otherwise.

For each training input, we compare the predicted output $\hat{y}$ with the actual label y (either 1 or -1):

$$e = y - \hat{y}$$

If the prediction is correct, e = 0 , meaning no error, and the weights are not updated, and if the prediction is incorrect, $e \neq 0$, meaning there is an error, and we need to adjust the weights.

If an error occurs, we update the weights according to the following rule:

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \eta \cdot e \cdot \mathbf{x}\prime$$

where: η is the learning rate, controlling how large the weight updates should be. $e = y - \hat{y}$ is the error, indicating the direction and magnitude of the adjustment. $\mathbf{x}\prime$ is the input vector (including the bias term).

This update rule modifies the weights in such a way that the model is more likely to predict correctly for the current example in future iterations.

The Perceptron algorithm converges if the data is linearly separable, meaning that after a certain number of weight updates, the model will correctly classify all training examples. When this happens: The error e = 0 for all training examples, and the weights stop changing. Training can be halted early since the model has reached its goal of perfect classification on the training data.

If the data is not linearly separable, the Perceptron may never converge, and the training will continue for the maximum number of epochs without finding a perfect solution. However, the weights will still adjust to reduce classification errors as much as possible.

2.2 Part_2 & Part_3

The second problem focused on finalizing two Python files, $modules.py$ and $mlp\_numpy.py$, and then using them to run a training procedure called $train_mlp_numpy.py$ for an MLP.

1. Completion of $modules.py$:

This file contains the implementation of core neural network modules. For the Linear Layer, we implements a fully connected layer with weights and biases initialized. The forward pass computes $output = x * weight + bias$, while the backward pass computes gradients for backpropagation. For the ReLU Activation, we implements the ReLU activation function, with the forward pass applying the element-wise maximum of 0 and input, and the backward pass computing gradients. For the SoftMax, we implements the softmax function to convert logits into probabilities. This is useful for classification tasks. For the crossEntropy Loss, we implements the cross-entropy loss function, which is combined with softmax for the forward pass and computes gradients during the backward pass.

2. Completion of $mlp\_numpy.py$:

This file implements the MLP architecture using the modules from modules.py. For the initialization, we initializes multiple linear layers, depending on the specified number of input features, hidden units, and output classes. ReLU activations are applied between layers, and softmax is used for the output. For the Forward Pass, we Sequentially applies the linear layers and activations to the input data to compute the output.Then for the Backward Pass, we implements backpropagation through all layers, updating weights using computed gradients.

3. Running the Training:

After completing the Python files, a train function was implemented that: First loads a synthetic dataset created using $make_moons$ from scikit-learn. Then initializes the MLP model and cross-entropy loss function. Then performs training with gradient descent by iteratively updating weights. Finally tracks and evaluates model performance by logging metrics like loss and accuracy during training.

The third problem involved enhancing the training function to allow the user to choose between Batch Gradient Descent (BGD) or Stochastic Gradient Descent (SGD) by adding a parameter for the batch size, which is based on Part_2.

First of all, the difference between Batch Gradient Descent and Stochastic Gradient Descent, should be when the user specifies a batch size equal to the full training dataset, BGD is used. This method

computes gradients using all training samples before performing an update. When the batch size is set to 1, the model processes one training sample at a time, updating weights after each sample, SGD is used.

Then we made changes to the train file, a new argument batch_size was introduced to control whether to use full-batch or mini-batch training. The training loop was modified to shuffle the dataset and split it into batches, depending on the batch_size value. For each batch, the model: 1. Performs a forward pass to compute the output. 2. Computes the loss using cross-entropy. 3. Backpropagates the gradients through the model. 4. Updates the weights of the linear layers based on the computed gradients. 5. This flexibility allows the model to be trained using either BGD or SGD, depending on the task or dataset size.

Finally we logging training progress, the modified train method also logs the training progress, including the loss and accuracy for both training and test sets, at regular intervals. The logs are saved in a CSV file for later analysis or plotting of learning curves.

3 Experiment

We will introduce the dataset used in the experiment, the variables created by the experiment and the main results obtained.

3.1 Datasets

In Part_1, the dataset mainly generated through function def $generate\_data()$ in $train.py$, the function generates two sets of points, each coming from a multivariate normal distribution. The first set (points1) has a mean of $[2, 3]$ and a covariance matrix of $[[1, 0], [0, 1]]$, producing 100 samples; The second set (points2) has a mean of $[7, 8]$ and the same covariance matrix, generating 100 samples. Each point set is assigned labels: points1 receives a label of 0; points2 receives a label of 1. The random state is set to 42 to ensure reproducibility. The points are combined into one dataset, and the corresponding labels are concatenated. The combined data is shuffled, and the dataset is split into training and testing sets. 80 samples of each label (0 and 1) are used for the training set; 20 samples of each label are used for the test set. Both the training and test sets are reshuffled to ensure randomness, and the function returns these two sets.

As for Part_2 and Part_3, the dataset mainly generated through function def $generate\_data()$ in $train\_mlp\_numpy.py$. This function generates a dataset using the $make\_moons$ function from $sklearn$, which creates a two-dimensional dataset with two interleaving half-moon shapes. The dataset consists of 1,000 samples with a small noise level of 0.1 to introduce variation; The random state is set to 42 to ensure reproducibility. The generated data is stored in a pandas DataFrame with columns 'x' and 'y', and the labels (0 or 1) are assigned to the 'label' column. The entire dataset is shuffled using a random seed (42) to maintain randomness. The dataset is split such that: The first 800 samples are used as the training set; The remaining 200 samples are used as the test set.

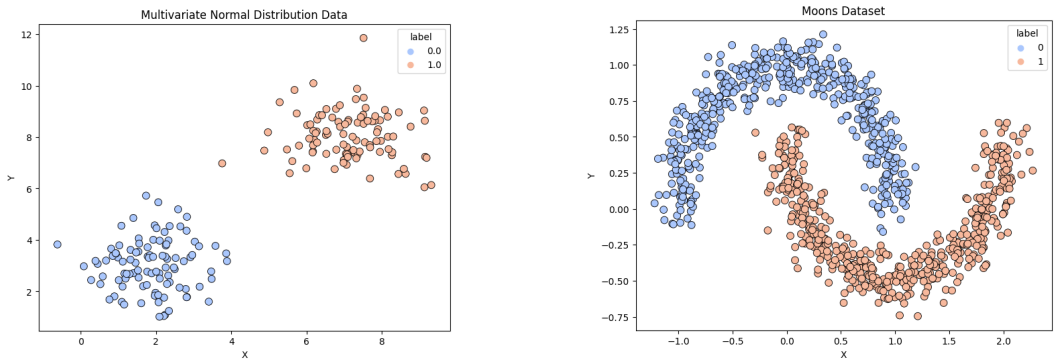
We represent our dataset in Figure 1 on next page.

3.2 Variables

In Part_1, we made change to bias and threshold of activation function, result shows in Table 1 and Table 2.

Bias	Threshold	Convergence Epochs	Accuracy
0	0	1000	50%
0.1	0	13	100%
1	0	7	100%

Table 1: Bias Change



(a) Part_1

(b) Part_2 & Part_3

Figure 1: Datastes

Bias	Threshold	Convergence Epochs	Accuracy
1	0	7	100%
1	0.1	4	100%
1	10	3	100%

Table 2: Threshold Change

The results can be analyzed through the interaction between the bias term and the threshold in the activation function, particularly in terms of how these parameters affect the decision boundary and model convergence.

The perceptron learns by updating its weights using the rule:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \eta(y - \hat{y})\mathbf{x}_{\mathbf{bias}}$$

η is the learning rate, y is the true label, $\hat{y}$ is the predicted label, $\mathbf{x}_{\mathbf{bias}}$ is the input vector with an appended bias term.

Effect of Bias Term The bias term helps shift the decision boundary without relying on input features alone. Mathematically, the bias term b is added as a constant to the input, influencing the weighted sum before the activation. With zero bias, the model struggles to correctly classify points unless they are perfectly linearly separable and centered at the origin. The decision boundary relies solely on the inputs. Thus, convergence is very slow and accuracy is poor since the algorithm needs many epochs to adjust weights. A small bias term begins to shift the decision boundary slightly, helping the model converge faster. The decision boundary is no longer forced to pass through the origin, improving the models ability to separate classes and leading to full accuracy. A bias term of 1 has a larger impact, effectively moving the decision boundary even further, reducing the number of epochs to converge. Since the bias helps accommodate the inherent offset in the data distributions, the model converges more quickly.

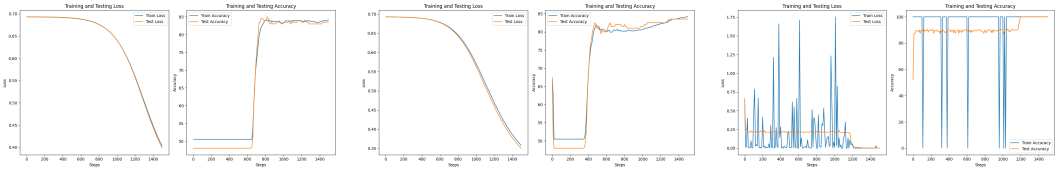
Effect of Threshold The threshold affects the activation function. When the weighted sum reaches the threshold, the activation fires. This controls how confident the perceptron must be to classify inputs. With zero threshold, the model requires fewer epochs to converge since it fires as soon as the weighted sum becomes non-negative. However, this may result in more small updates, which still work in this case. By introducing a small positive threshold, the model demands a slightly stronger activation to classify a sample as positive. This leads to faster convergence, as the model fine-tunes the weights more effectively to reach the higher threshold. A high threshold forces the perceptron to achieve a very large weighted sum before making a positive classification. This accelerates convergence since the model quickly adjusts to create a strong margin between classes, though it might rely heavily on extreme weight adjustments early in training.

Mathematical Explanation For any input $\mathbf{x}$, the perceptron computes a weighted sum $s = \mathbf{w}^T\mathbf{x}+b$. With large bias, the term b dominates, meaning the model shifts the decision boundary far away from the origin. This can quickly separate linearly separable data. With increased threshold, the model

becomes stricter in classification, requiring the weighted sum to reach a higher value. This means that smaller updates are no longer enough to change the output, leading to more decisive updates and faster convergence.

3.3 Main results

Shows in Figure 2.



(a) Part_2 (b) BGD (c) SGD

Figure 2: Training process

In batch gradient descent, the model updates its parameters using the gradients calculated over the entire dataset. This leads to a smooth and relatively stable trajectory for the loss and accuracy. Initially, as the model learns, the loss and accuracy improve steadily as it captures the general trends in the data. After the initial improvements, the loss and accuracy often hit a plateau. This happens because the model might be stuck in a region where the gradients are small, or the model is approaching a local minimum, but the learning rate is not fine-tuned enough to make steady progress. The sudden improvement later could be due to one of two reasons: 1. Learning Rate Impact: If learning rate is relatively large, the model might overshoot the optimal region a few times. At a certain point, it might jump out of a suboptimal region and reach a better configuration, leading to a sudden improvement in performance. 2. Epoch-Specific Data Distribution: As BGD looks at the entire dataset, it could have been in a situation where improvements were gradual, but a more significant improvement occurred once the weights moved closer to a better solution.

In SGD, the model updates its weights after every single training example. Since each individual example contributes noisy gradients, the model tends to have highly fluctuating loss and accuracy values. This noise is due to the fact that each example might pull the model's parameters in a different direction, which can be very different from the true gradient direction that batch gradient descent would calculate using the entire dataset. Unlike BGD, SGD has high variability and fluctuating loss/accuracy because the updates are based on a small, often noisy subset of the data (just one sample in the case of standard SGD). This means that while the model may eventually reach a better solution, the path there is much less smooth, and it may take longer to converge fully. Despite the noise, SGD is known for its ability to avoid getting stuck in poor local minima, which can help improve performance in the long run.

Both behaviors are typical and reflect the trade-offs between the more stable updates of BGD and the faster but noisier updates of SGD. If we want more stability in SGD, using a mini-batch gradient descent (processing smaller batches, e.g., 16, 32, or 64 samples at a time) can provide a good compromise between stability and speed.

4 Conclusion

In this task, we used perceptron to solve some linearly separable and indivisible problems. Back propagation and gradient updating are used to enable perceptron to learn new knowledge for classification tasks. In the future, we can further change the structure of the model so that it can capture more complex data feature structures to complete more difficult classification tasks.