CS324 · Deep Learning

Assignment 1 Report

Sicheng Zhou / 12110644

October 17, 2024

1 Files Description

Under directory Part_1:

- generate.py: define two Gaussian distributions and sample 100 points from each.
- perceptron.py: implement the perceptron.
- train.py: invoke the generate function and train the perceptron.
- watch.py: visualize the dataset.
- results.ipynb: run train.py with different parameters and display the results.

Under directory Part_2:

- generate.py: Using scikit-learn and the make moons method, create a dataset of 1000 two-dimensional points.
- mlp_numpy.py and modules.py: implement the multi-layer perceptron.
- train_mlp_numpy.py: invode mlp_numpy.py to train the multi-layers perceptron model.
- results.ipynb: run train_mlp_numpy.py with different parameters and display the results.

2 The Perceptron

A single perceptron consists of:

- **Input layer**: Takes a vector of inputs $x \in \mathbb{R}^n$.
- **Weights and bias**: Each input is associated with a weight w_i, and the perceptron also includes a bias term b. Those are adjusted during training.
- **Activation function**: The perceptron uses a sign function as its activation

$$f(x) = \begin{cases} 1 & \text{if } w^T x + b > 0 \\ -1 & \text{otherwise} \end{cases} \tag{1}$$

The learning process includes:

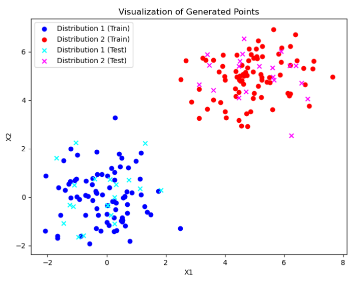
- **Initialization**: Weights are initialized to zeros.
- **Prediction**: For each input, the perceptron computes the dot product of the weights and input features, adds the bias, and applies the activation function to predict the class label.
- **Error computation**: The perceptron calculates the error as the difference between the true label y_i and the predicted label $\hat{y}_i$.
- **Weight update**: For each misclassified point, the weights and bias are updated using the gradient descent rule:

$$\begin{aligned} w &\leftarrow w + \eta(y_i - \hat{y}_i)x_i \\ b &\leftarrow b + \eta(y_i - \hat{y}_i) \end{aligned} \tag{2}$$

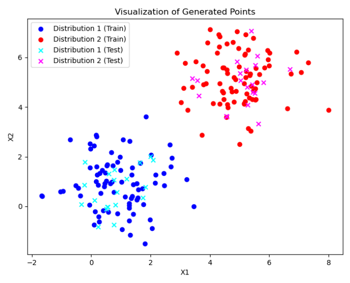
If the data is linearly separable, the perceptron learning algorithm will converge to a solution in a finite number of iterations. However, if the data is not linearly separable, the algorithm will never converge and may continue to oscillate.

In Task 4, we tried to adjust the difference between the means and the variances of the two Gaussian distributions. We can figure out that:

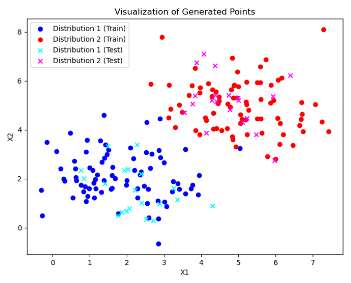
- If the means are too close, the training accuracy will drop. Because the data points are too close to each other so they are no longer linear separable. Experiment results see Figure 1.
- If the variance of the distribution is too high, the training accuracy will also drop because there will be more outliers. Experiment results see Figure 2.



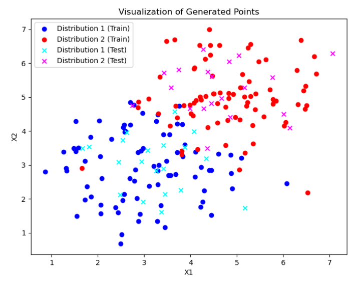
(a) mean1: [0, 0], Accuracy: 0.975



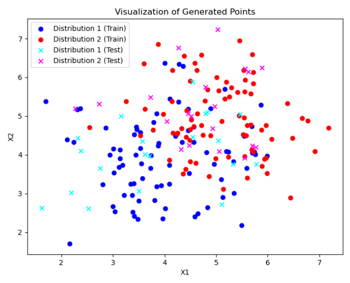
(b) mean1: [1, 1], Accuracy: 1.0



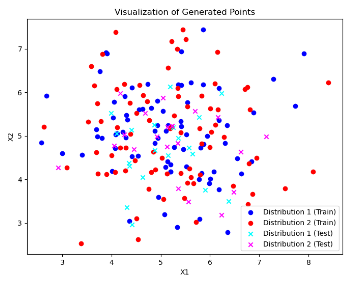
(c) mean1: [2, 2], Accuracy: 0.5



(d) mean1: [3, 3], Accuracy: 0.5



(e) mean1: [4, 4], Accuracy: 0.5



(f) mean1: [5, 5], Accuracy: 0.5

Figure 1: The test accuracy of different mean1. mean2 is fixed to [5, 5], both cov = [[1, 0], [0, 1]], max epochs = 15, learning rate = 0.1. The accuracy will drop when the means of two distributions are too close to each other.

3 The Multi-layer Perceptron

3.1 Forward Propagation

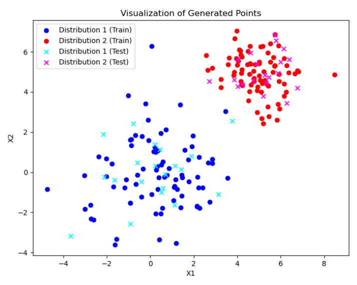
Forward propagation involves passing input data through the network, layer by layer, to compute the final output. Each layer in the MLP performs two key operations: a *linear transformation* followed by a *non-linear activation*.

For a given input $\mathbf{x}^{(l)}$ to layer l, the output $\mathbf{z}^{(l+1)}$ is computed as:

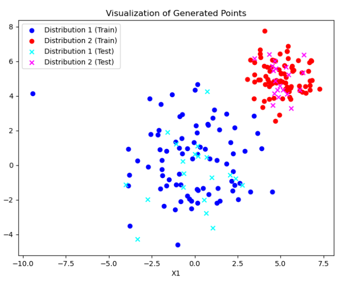
$$\mathbf{z}^{(l+1)} = \mathbf{W}^{(l)}\mathbf{x}^{(l)} + \mathbf{b}^{(l)}$$

Where:

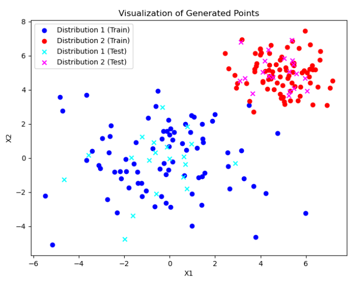
- $\mathbf{W}^{(l)} \in \mathbb{R}^{m \times n}$ is the weight matrix of layer l (with m neurons and n inputs).
- $\mathbf{b}^{(l)} \in \mathbb{R}^{m}$ is the bias vector of layer l.
- $\mathbf{x}^{(l)} \in \mathbb{R}^{n}$ is the input vector to layer l (output of the previous layer or input data for the first layer).



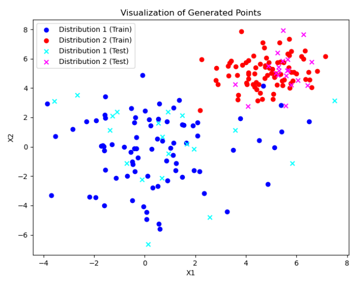
(a) cov1: [[3, 0], [0, 3]], Accuracy: 0.98



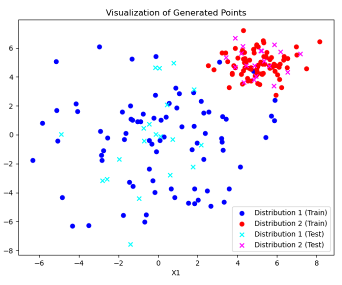
(b) cov1: [[4, 0], [0, 4]], Accuracy: 1.0



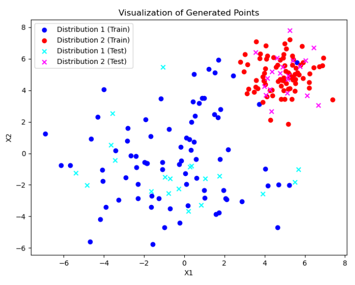
(c) cov1: [[5, 0], [0, 5]], Accuracy: 1.0



(d) cov1: [[6, 0], [0, 6]], Accuracy: 0.95



(e) cov1: [[7, 0], [0, 7]], Accuracy: 0.85



(f) cov1: [[8, 0], [0, 8]], Accuracy: 0.93

Figure 2: The test accuracy of different cov1. cov2 is fixed to [[1, 0], [0, 1]], mean1 = [0, 0], mean2 = [5, 5], max epochs = 15, learning rate = 0.1. The accuracy will drop when the variance is too high.

- $\mathbf{z}^{(l+1)} \in \mathbb{R}^m$ is the pre-activation output (the weighted sum before applying the activation function).

After applying the linear transformation, the result passes through a non-linear activation function $f(\cdot)$ to introduce non-linearity to the network:

$$\mathbf{a}^{(l+1)} = f(\mathbf{z}^{(l+1)})$$

Where:

- $f(\cdot)$ is typically a non-linear function like *ReLU*, *sigmoid*, or *tanh*.
- $\mathbf{a}^{(l+1)}$ is the final output (activation) of layer $l + 1$.

This process repeats for each layer until reaching the final output layer.

3.2 Backward Propagation

Backward propagation is the process of computing the gradient of the loss function with respect to each parameter (weights and biases) in the network, enabling the model to update these parameters using gradient-based optimization methods (e.g., stochastic gradient descent). The key idea is to apply the *chain rule* to propagate errors from the output layer back through the network.

3.2.1 Backpropagation through the Output Layer

1. **Error at the Output Layer**: The gradient of the loss with respect to the pre-activation output $\mathbf{z}^{(L)}$ at the output layer is:

$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(L)}} \odot f'(\mathbf{z}^{(L)})$$

   Where:

   - $\delta^{(L)}$ is the error signal (gradient) at the output layer.
   - $\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(L)}}$ is the gradient of the loss with respect to the output activations.

- $f'(\mathbf{z}^{(L)})$ is the derivative of the activation function at the output layer.

2. **Gradients with Respect to Weights and Biases**: The gradients for the weights and biases in the output layer are computed as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L-1)}} = \delta^{(L)} \cdot \mathbf{a}^{(L-1)^\top}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(L-1)}} = \delta^{(L)}$$

Where $\mathbf{a}^{(L-1)}$ is the activations from the previous layer.

3.2.2 Backpropagation through Hidden Layers

For each hidden layer l, the error signal is backpropagated to compute the gradients for that layer. The error at layer l depends on the error at layer $l+1$ as follows:

1. **Error at Layer** l: The error at layer l is computed using the error at layer $l+1$ and the weights between layers l and $l+1$:

$$\delta^{(l)} = \left(\mathbf{W}^{(l)}\right)^\top \delta^{(l+1)} \odot f'(\mathbf{z}^{(l)})$$

Where $\delta^{(l)}$ is the error at layer l, and $f'(\mathbf{z}^{(l)})$ is the derivative of the activation function at layer l.

2. **Gradients with Respect to Weights and Biases**: The gradients for the weights and biases in layer l are then computed as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l-1)}} = \delta^{(l)} \cdot \mathbf{a}^{(l-1)^\top}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l-1)}} = \delta^{(l)}$$

This process continues layer by layer until the gradients are computed for all layers.

3.3 Batch Gradient Descent and Stochastic Gradient Descent

Batch Gradient Descent compute the gradient of the loss function using the entire training dataset. For a model with N training examples, the gradient at each step is calculated as:

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta L(\theta, x^{(i)}, y^{(i)})$$

This process provides a more stable and smoother convergence path toward the global minimum because it averages over all training examples. But it also is computationally expensive as it requires summing over all the training examples to calculate a single gradient update.

Stochastic Gradient Descent computes the gradient using only one training example at a time. For each training example $(x^{(i)}, y^{(i)})$, the gradient is calculated as:

$$\nabla_\theta J(\theta) = \nabla_\theta L(\theta, x^{(i)}, y^{(i)})$$

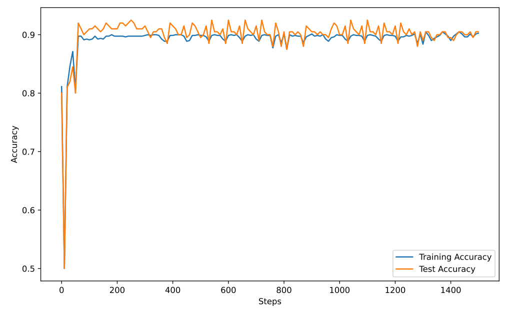
This method introduces more noise in the gradient updates because it relies on one example for each update, which can cause fluctuations in the path to the minimum. The convergence may be faster, but it is often less stable and may oscillate around the minimum rather than settle precisely at it.

Mini-Batch Gradient Descent is a compromise between BGD and SGD. It calculates the gradient using a small random subset of the training data (mini-batch) in each iteration. It combines the benefits of both BGD and SGD: faster convergence than BGD and more stable updates than SGD.

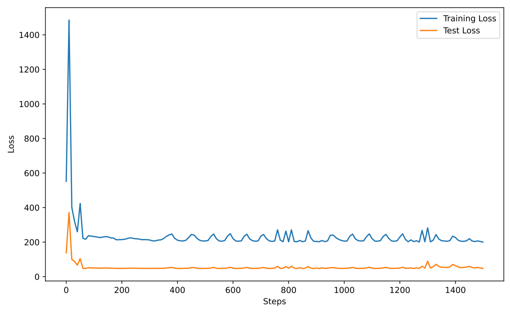
In the experiments, the performances of these methods are compared in two ways.

1. If one epoch means going through all the samples, the results are shown in Figure 3. Batch gradient descent provides a more stable and smoother convergence path, while mini-batch gradient descent converges faster to the global optimal solution.

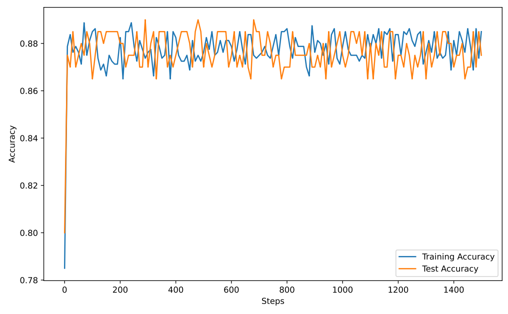
2. If one epoch means going through a batch of samples, the results are shown in Figure 4. Stochastic GD and mini-batch GD took only half of the time to finish training.



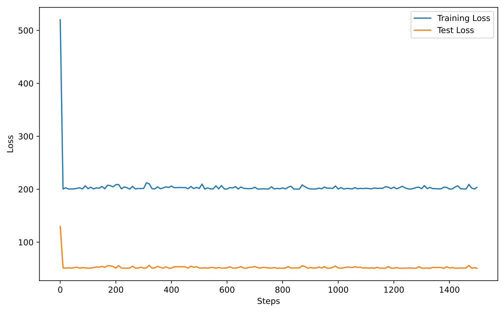
(a) Batch Gradient Descent Accuracy



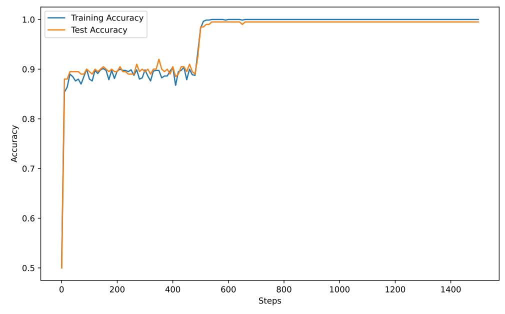
(b) Batch Gradient Descent Loss



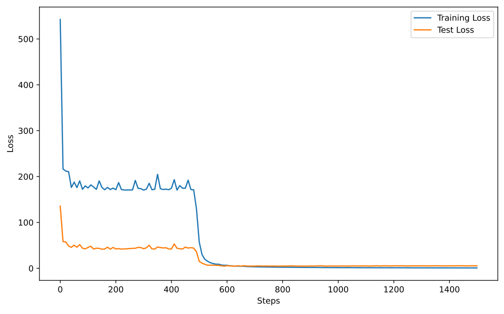
(c) Stochastic Gradient Descent Accuracy



(d) Stochastic Gradient Descent Loss

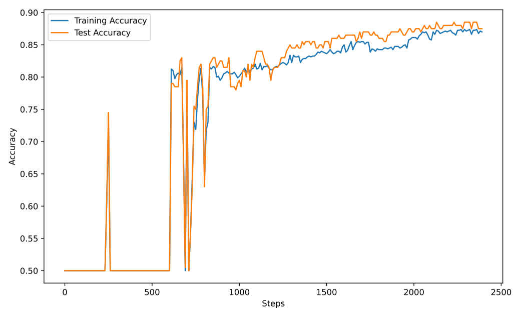


(e) Mini-Batch GD Accuracy, batch size = 100

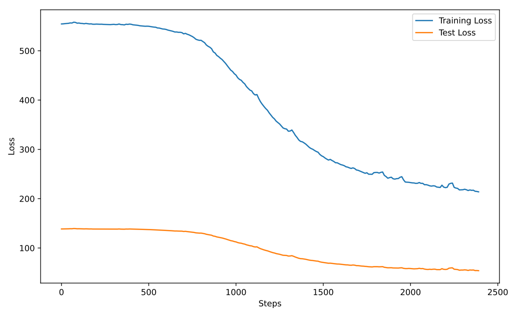


(f) Mini-Batch GD Loss, batch size = 100

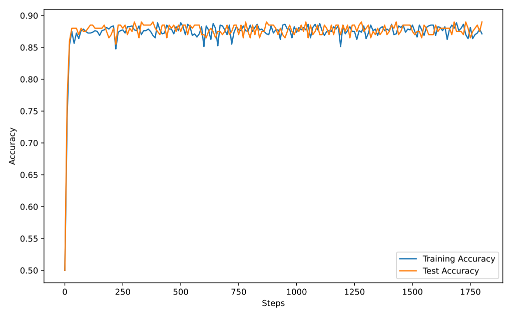
Figure 3: If one epoch means going through all the samples. The number of hidden units = 20, learning rate = 0.01, max epoch = 1500. Batch gradient descent provides a more stable and smoother convergence path, while mini-batch gradient descent converges faster.



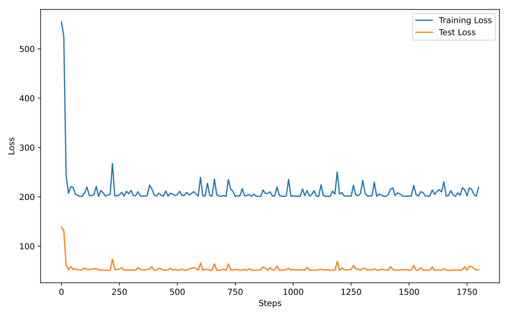
(a) Stochastic Gradient Descent Accuracy



(b) Stochastic Gradient Descent Loss



(c) Mini-Batch GD Accuracy, batch size = 100



(d) Mini-Batch GD Loss, batch size = 100

Figure 4: If one epoch means going through a batch of samples The number of hidden units = 20, learning rate = 0.01, max epoch = 1500. Batch gradient descent took 0.38s to finish, while stochastic gradient descent took only 0.20s, and mini-batch gradient descent took 0.26s.