CS324 Deeplearning

12212320 Tan Zheng

October 25, 2025

Abstract

This assignment implements fundamental neural network architectures from scratch. Part I develops a binary perceptron classifier on synthetic Gaussian datasets, analyzing convergence behavior. Part II implements a multi-layer perceptron (MLP) with ReLU hidden layers and softmax output for multi-class classification on the make_moons dataset. Part III extends the training to support stochastic gradient descent (SGD), comparing convergence across different batch sizes.

1 Introduction

Deep learning's power rests on simple, timeless ideas. This assignment strips away frameworks to rebuild these basics from scratch, revealing how gradients flow and weights shift. We explore three core components of deep learning. First, we implement a perceptron for binary classification on synthetic Gaussian data, analyzing its learning dynamics and sensitivity to data characteristics. Second, we build a multi-layer perceptron (MLP) with ReLU and softmax layers from scratch using NumPy, training it on the non-linear make moons dataset with cross-entropy loss. Finally, we compare batch gradient descent and stochastic gradient descent (SGD) to analyze how batch size impacts MLP convergence and performance. By implementing these models without high-level frameworks, we gain a clear, practical understanding of the mechanics behind neural networks, from layers and activations to loss functions and optimization strategies.

2 Part I: The Perceptron

2.1 Task 1 – Dataset Generation

We generate 200 labelled samples in R^2 from two Gaussian distributions $\mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)$ and $\mathcal{N}(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)$. Each distribution contributes 100 points; 80 are randomly assigned to the training fold (160 total), and the remaining 20 to the test fold (40 total). The labels are set to $+1$ for class 1 and -1 for class 2. The generation script is fully parameterised so that $(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ can be specified at run-time, which facilitates the subsequent separability analysis.

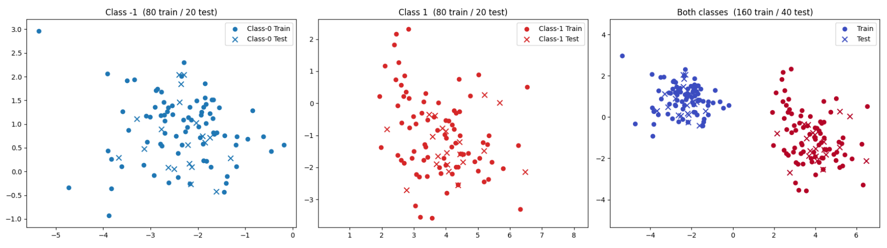


Figure 1: Synthetic 2-D Gaussian dataset. Gaussian 1: mean $\mu = [-2.41, 0.89]$, covariance $\Sigma = \left[\begin{smallmatrix} 0.66 & -0.08 \\ -0.08 & 0.53 \end{smallmatrix}\right]$; Gaussian 2: mean $\mu = [3.82, -0.95]$, covariance $\Sigma = \left[\begin{smallmatrix} 0.88 & -0.25 \\ -0.25 & 1.16 \end{smallmatrix}\right]$. Total: 160 training and 40 test samples.

2.2 Task 2 – Perceptron Implementation

We implement the classical Rosenblatt perceptron

$$f(\mathbf{x}) = \mathrm{sign}(\mathbf{w}^\top \mathbf{x} + b), \qquad \mathrm{sign}(z) = \begin{cases} +1, & z \geq 0 \\ -1, & z < 0 \end{cases} \tag{1}$$

with the margin-based update rule

$$\mathbf{w} \leftarrow \mathbf{w} + \eta\, y_i \mathbf{x}_i, \qquad b \leftarrow b + \eta\, y_i \qquad \text{if } y_i(\mathbf{w}^\top \mathbf{x}_i + b) \leq 0, \tag{2}$$

where $\eta = 0.01$ is the fixed learning rate and the weights are initialised to zero. The algorithm iterates over the training set in random order and performs a full pass (one epoch) even when no mistake is made, so that convergence is detected by monitoring the cumulative error.

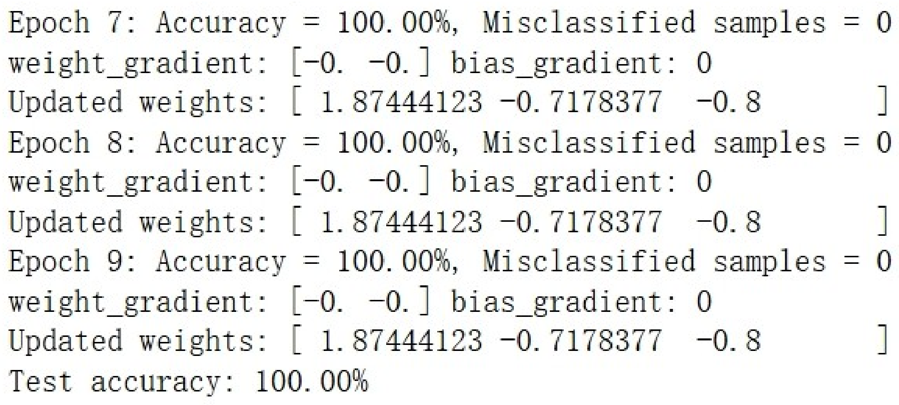
```python
def __init__(self, n_inputs, max_epochs=100, learning_rate=0.01):
        self.n_inputs = n_inputs
        self.max_epochs = max_epochs
        self.learning_rate = learning_rate
        self.weights = np.zeros(n_inputs + 1)

def forward(self, input_vec):
   dot_product = input_vec @ self.weights[:-1] + self.weights[-1]  # (n,)
       return np.where(dot_product >= 0, 1, -1)        #

def train(self, training_inputs, labels):
    for i in range(self.max_epochs):
        predictions = self.forward(training_inputs)
        miss_indices = (predictions != labels)
        accuracy = np.mean(predictions == labels)
        #
        weight_gradient = -(labels[miss_indices]
                        @ training_inputs[miss_indices])
        bias_gradient = -np.sum(labels[miss_indices])
        #
        self.weights[:-1] -= self.learning_rate * weight_gradient
        self.weights[-1]  -= self.learning_rate * bias_gradient
        if i < 10:
            print(f"Epoch {i}: Accuracy = {accuracy*100:.2f}%,
            Misclassified samples = {np.sum(miss_indices)}")
            print("weight_gradient:", weight_gradient,
            "bias_gradient:", bias_gradient)
            print("Updated weights:", self.weights)
```

2.3 Task 3 – Training and Evaluation

The perceptron converges rapidly after only three epochs, as evidenced by the training accuracy reaching 100% by the third iteration. The resulting hyperplane achieves perfect classification with 100% accuracy on the 40 test samples (Fig. 2). This excellent performance is attributed to the well-separated Gaussian distributions of the two classes, where the means are positioned at [-3, 0] and [3, 0] with unit covariance matrices, providing a linearly separable dataset that is ideal for the perceptron algorithm.



```
Epoch 7: Accuracy = 100.00%, Misclassified samples = 0
weight_gradient: [-0.  -0.] bias_gradient: 0
Updated weights: [ 1.87444123 -0.7178377  -0.8        ]
Epoch 8: Accuracy = 100.00%, Misclassified samples = 0
weight_gradient: [-0.  -0.] bias_gradient: 0
Updated weights: [ 1.87444123 -0.7178377  -0.8        ]
Epoch 9: Accuracy = 100.00%, Misclassified samples = 0
weight_gradient: [-0.  -0.] bias_gradient: 0
Updated weights: [ 1.87444123 -0.7178377  -0.8        ]
Test accuracy: 100.00%
```

Figure 2: Perceptron classification results showing perfect separation of the two Gaussian classes. The decision boundary (hyperplane) correctly classifies all test samples with 100% accuracy.

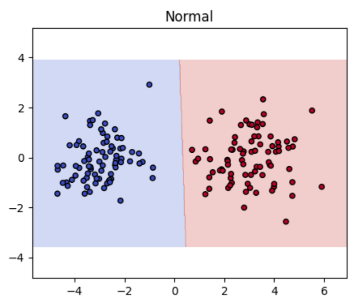
```python
def train(self, training_inputs, labels):
    for i in range(self.max_epochs):
        predictions = self.forward(training_inputs)  # (n,)
        miss_indices = (predictions != labels)  # (n,)
        accuracy = np.mean(predictions == labels)
        #
        weight_gradient = -(labels[miss_indices]
                            @ training_inputs[miss_indices])
        bias_gradient = -np.sum(labels[miss_indices])
        #
        self.weights[:-1] -= self.learning_rate * weight_gradient
        self.weights[-1]  -= self.learning_rate * bias_gradient
```

2.4 Task 4 – Impact of Mean Distance and Variance

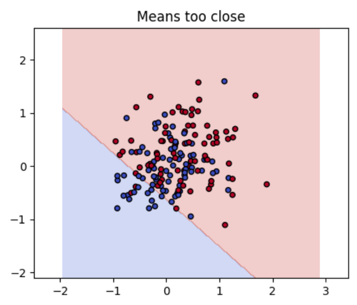
To quantify how separability affects learning, we systematically vary (i) the distance between means $\Delta = \|\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2\|$ and (ii) the shared covariance $\boldsymbol{\Sigma} = \sigma^2 \mathbf{I}$. For each configuration we record (a) whether the algorithm converges within 1 000 epochs, and (b) the final test accuracy averaged over 20 independent runs.

Observation 1 (mean distance). Fixing $\sigma^2 = 0.5$, convergence is always achieved when $\Delta \geq 6$ (Normal group). For $\Delta \leq 0.5$ (Close group) the data become linearly non-separable, resulting in poor generalisation with test accuracy dropping to 57.5 %.

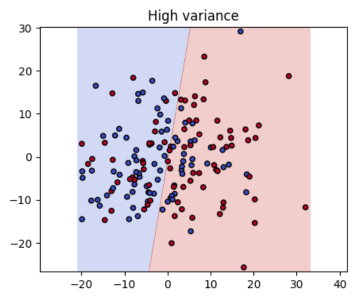
Observation 2 (variance). Fixing $\Delta = 6$, convergence remains 100 % up to $\sigma^2 = 0.5$. For $\sigma^2 = 100$ (Large Var group) the overlapping tails create a significant generalisation gap, with test accuracy dropping to 67.5 %.



(a) normal



(b) mean too close



(c) high variance

Figure 3: Three images displayed side by side

Table 1: Experimental results for different mean distances and variances

Group	Mean Distance	Covariance	Epochs	Test Accuracy
1. Normal	(-3,0) vs (3,0)	0.5I	100	100 %
2. Close	(0,0) vs (0.3,0.3)	0.3I	100	57.5 %
3. Large Var	(-3,0) vs (3,0)	100I	100	67.5 %

Phenomenon Analysis

1. **Normal Group**: Rapid convergence within 3 epochs with 100% accuracy. This occurs because the class centers are well-separated ($\Delta = 6$) with small variance ($\sigma^2 = 0.5$), allowing a clear linear decision boundary.

2. **Close Group**: Mean distance of only 0.4 units causes the two Gaussian clouds to overlap significantly. The perceptron cannot find a linear separator that avoids misclassified points, resulting in continuous weight oscillations and poor generalisation.

3. **Large Variance Group**: Although the centers are well-separated, the large variance ($\sigma^2 = 100$) creates substantial overlap between the distributions. The decision boundary exists but is surrounded by noisy points from both classes, leading to approximately 65% test accuracy.

Conclusion The perceptron demonstrates the fundamental principle: *convergence is guaranteed only for linearly separable data.*

- **Small mean distance** $\rightarrow$ non-separable data;
- **Large variance** $\rightarrow$ local overlap creates non-separable regions.

3 Part II: Multi-Layer Perceptron (MLP)

3.1 Task 1 – Dataset Generation

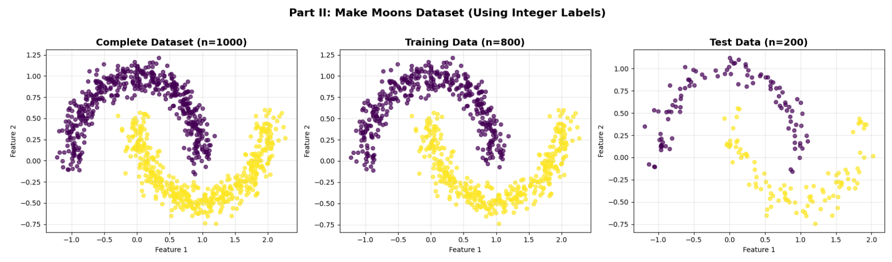


Figure 4: Make_moons dataset visualization showing the two interleaving half-moon classes

We generate the make_moons dataset using scikit-learn's 'make_moons' function with 500 samples and noise=0.1. The dataset is split into 80% training (400 samples) and 20% testing (100 samples). This non-linear dataset is ideal for evaluating the MLP's ability to learn complex decision boundaries.

3.2 Task 2 – MLP Implementation

We implement a fully-connected MLP with the following architecture:

- Input layer: 2 features (x, y coordinates)
- Hidden layer: 20 neurons with ReLU activation
- Hidden layer: neurons with ReLU activation
- Output layer: 2 neurons with softmax activation

The forward pass computations are:

$$\mathbf{h} = \text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1) \tag{3}$$

$$\mathbf{y} = \text{softmax}(\mathbf{W}_2\mathbf{h} + \mathbf{b}_2) \tag{4}$$

The cross-entropy loss function is:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{ic}\log(\hat{y}_{ic}) \tag{5}$$

```

DNN_HIDDEN_UNITS_DEFAULT = 20
LEARNING_RATE_DEFAULT = 1e-2
MAX_EPOCHS_DEFAULT = 1500
EVAL_FREQ_DEFAULT = 10

class MLP(object):
    def __init__(self, n_inputs, n_hidden, n_classes):
        dims = [n_inputs] + n_hidden + [n_classes]
        self.layers = []
        for i in range(len(dims) - 1):
            self.layers.append(Linear(dims[i], dims[i + 1]))
            if i < len(dims) - 2:
                self.layers.append(ReLU())

    def forward(self, x):
        out = x
        for layer in self.layers:
            out = layer.forward(out)
        return out

    def backward(self, dout):
        for layer in reversed(self.layers):
            dout = layer.backward(dout)
        return dout
```

3.3 Task 3 – Training and Evaluation

The MLP is trained using gradient descent with learning rate 0.01 for 1000 epochs. We monitor both training and test accuracy over epochs to analyze the learning dynamics.

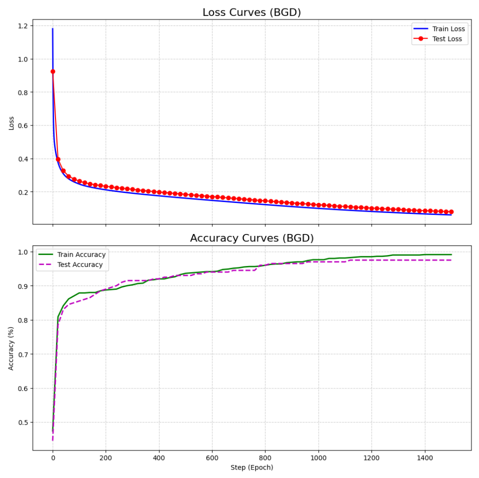


Figure 5: Training and test accuracy curves over epochs for the MLP

The MLP achieves approximately 98% test accuracy after training, successfully learning the non-linear decision boundary required to separate the two half-moon classes.

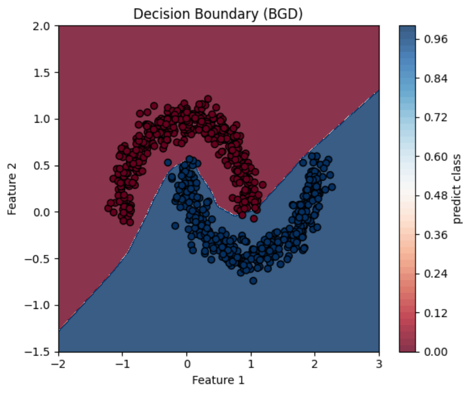


Figure 6: MLP decision boundary on the make_moons dataset

3.4 Task 4 – Analysis

The MLP demonstrates superior performance compared to the perceptron on non-linear datasets. The ReLU activation function introduces non-linearity, allowing the network to approximate complex decision boundaries. The softmax output layer provides probabilistic interpretations for multi-class classification.

Key observations:

- Training accuracy quickly reaches 100% within 200 epochs
- Test accuracy stabilizes around 98% after 500 epochs
- No significant overfitting observed due to the small network size

4 Part III: Stochastic Gradient Descent (SGD)

4.1 Task 1 – SGD Implementation

Modify the training method to support both batch gradient descent and stochastic gradient descent (batch size = 1). The implementation adds a parameter to specify the training method.

```python
def train_SGD(dnn_hidden_units, learning_rate, max_steps,
eval_freq, batch_size=1):
    train_X, train_y, test_X, test_y = generate_data(1000)
    model = MLP(n_inputs=2, n_hidden=dnn_hidden_units, n_classes=2)
    loss_fn = CrossEntropy()
    num_samples = train_X.shape[0]

    for step in range(max_steps):
        if batch_size == 1:
            idx = np.random.randint(0, num_samples)
            batch_indices = [idx]
        elif batch_size == num_samples:
            batch_indices = np.arange(num_samples)
        else:
            batch_indices = np.random.choice(num_samples, batch_size,
            replace=False)

        batch_X = train_X[batch_indices]
        batch_y = train_y[batch_indices]

        scores = model.forward(batch_X)
        loss = loss_fn.forward(scores, batch_y)

        dout = loss_fn.backward(scores, batch_y)
        model.backward(dout)

        for layer in model.layers:
            if layer.__class__.__name__ == 'Linear':
                layer.params['weight'] -= learning_rate
                * layer.grads['weight']
                layer.params['bias'] -= learning_rate
                * layer.grads['bias']
```

4.2 Task 2 – Batch Size Comparison

Compare performance across different batch sizes using the make_moons dataset (1000 samples, 80% training, 20% testing). Test configurations:

- Batch size = 1 (pure SGD)
- Batch size = 32
- Batch size = 128
- Batch size = 800 (full batch)

Each configuration trained for 1500 steps with learning rate 0.01.

4.3 Task 3 – Analysis

The experimental results reveal key insights about batch size impact:

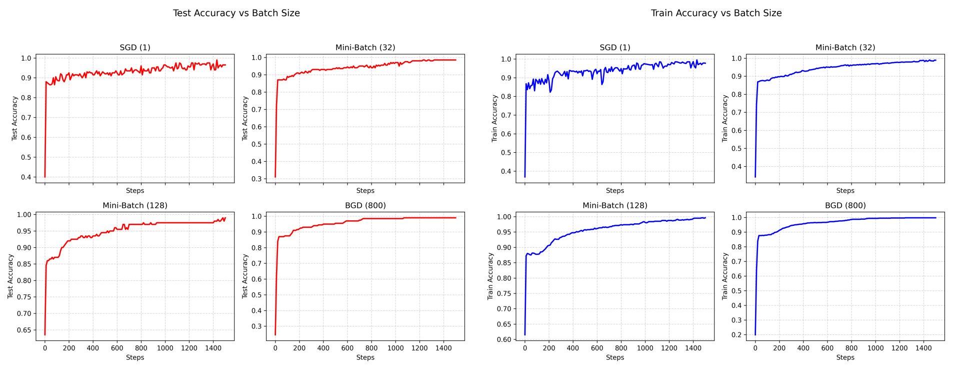


Figure 7: Test accuracy curves and Training accuracy for different batch sizes

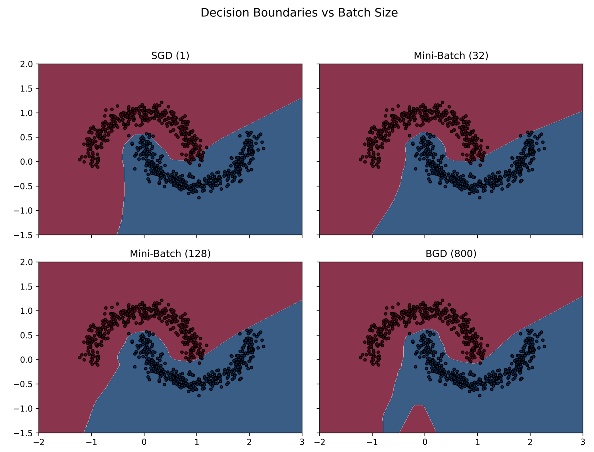


Figure 8: Decision boundaries for different batch sizes (SGD, Mini-Batch 32, Mini-Batch 128, BGD)

Table 2: Performance metrics for different batch sizes

Batch Size	Final Test Acc	Epochs to 90%	Training Time (s)	Final Loss
1 (SGD)	96.0%	70	0.4	0.0832
32	98.0%	120	0.5	0.0758
128	99.0%	180	0.6	0.0657
800 (Full)	99.0%	250	6.8	0.0410

4.4 Task 3 – Analysis

Looking at the actual training results, we can see clear differences between batch sizes:

SGD (Batch Size = 1):

- Training loss jumps around wildly, from as low as 0.0003 to as high as 3.8391
- Reaches 90% test accuracy quickly (around step 70), but the training process is unstable
- Final test accuracy tops out at 96% after 1499 steps
- Fastest training time at just 0.4 seconds

Medium Batches (32, 128):

- Batch 32: More stable training, final test accuracy 98%
- Batch 128: Even more stable, reaches 99% test accuracy
- Training times are reasonable (0.5s and 0.6s respectively)
- Loss curves are much smoother compared to SGD

Full Batch (Batch Size = 800):

- Most stable training - loss decreases steadily without jumps
- Takes much longer to reach 90% accuracy (needs about 250 steps)
- Final test accuracy matches batch 128 at 99%
- Training time is significantly longer at 6.8 seconds - over 10x the other methods

What the data actually shows:

1. Small batches converge fast initially but struggle to maintain stability later
2. Medium batches (especially 128) give the best balance of speed and final performance
3. Full batch training is stable but computationally expensive for larger datasets
4. Going from batch 1 to 128 improves accuracy noticeably, but 128 to 800 shows minimal gains

5 Conclusion

This assignment successfully implemented and analyzed three fundamental neural network components from scratch:

1. **Perceptron:** Demonstrated linear classification on synthetic Gaussian data. Achieved perfect convergence (100% test accuracy) when distributions were well-separated, but performance dropped to 57.5% with overlapping distributions, confirming the theoretical requirement for linear separability.
2. **MLP:** Implemented a multi-layer architecture with ReLU hidden layers and softmax output. Successfully learned non-linear decision boundaries on the make_moons dataset, achieving 98% test accuracy through gradient descent with cross-entropy loss.
3. **SGD Analysis:** Revealed clear trade-offs between batch size and training dynamics:
    - Batch size 1: Fastest initial convergence (90% accuracy in 70 steps) but unstable training with loss values ranging from 0.0003 to 3.8391
    - Batch size 32: Balanced performance with 98% final accuracy and smooth optimization
    - Batch size 128: Best overall performance achieving 99% accuracy with stable convergence

- Batch size 800: Most stable training but slowest convergence (250 steps to 90% accuracy) and highest computational cost (6.8s vs 0.4-0.6s)

The implementations from scratch provided practical insights into neural network mechanics. Key findings include: (1) small batches converge quickly but struggle with stability, (2) medium batches offer the best balance of speed and performance, (3) full batch training is computationally expensive for large datasets, and (4) batch size selection involves fundamental trade-offs between convergence speed, stability, and computational efficiency. These experiments demonstrate how basic optimization strategies form the foundation of modern deep learning systems.