CS324 Assignment 1

Li Xuanran
12312110

October 26, 2025

1 Part I: The perceptron (20 pts)

1.1 Task 1

Simply use `numpy`. Setting `np.random.seed(1)` is for reproducing the same random numbers, and the parameter can be any number. Labels distinguish the different 2D Gaussian distributions.

```
def generate(mean=2, cov=1):
    np.random.seed(11)
    mean1 = [mean, mean]
    cov1 = [[cov, cov], [cov, cov]]
    mean2 = [-mean, -mean]
    cov2 = [[cov, -cov], [-cov, cov]]
    data1 = np.random.multivariate_normal(mean1, cov1, 100)
    data2 = np.random.multivariate_normal(mean2, cov2, 100)
    labels1 = np.zeros(100)
    labels2 = np.ones(100)
    train_data = np.vstack((data1[:80], data2[:80]))
    train_labels = np.hstack((labels1[:80], labels2[:80]))
    test_data = np.vstack((data1[80:], data2[80:]))
    test_labels = np.hstack((labels1[80:], labels2[80:]))
```

Listing 1: Generate a dataset

1.2 Task 2

We compute the sign function `y`, which determines the mask operation.

```
def forward(self, input_vec):
    label_0 = np.ones([input_vec.shape[0], 1])
    matrix = np.hstack((input_vec, label_0))
    y = np.dot(self.weights, matrix.T)
    y = np.array([1 if i >= 0 else -1 for i in y])
    return y
```

Listing 2: `forward` function

According to **perceptron_tutorial.pdf**, we should only keep the sample i where `pred[i] * labels[i] < 0` to calculate the gradient, So we define `mask` matrix, and `True` means we pick up the sample.

```
def train(self, training_inputs, labels):
    for _ in range(self.max_epochs):
        p = self.forward(training_inputs)
        mask = p * labels < 0
        X, y = training_inputs[mask], labels[mask]
        grad = -np.sum(X * y.reshape(y.shape[0], 1)) / training_inputs.
    shape[0]
        self.weights = self.weights - self.learning_rate * grad
```

Listing 3: `train` function

1.3 Task 3

We train the model with epoch $= 20$ and $\alpha = 0.01$, setting mean $= 2$ and cov $= 1$ by default. For accuracy, we can directly invoke `accuracy_score` function in `sklearn.metrics`. **Listing 4** only shows the main part.

```
if __name__ == '__main__':
    model = Perceptron(n_inputs=2, max_epochs=20, learning_rate=0.01)
    X_train, X_test, y_train, y_test = generate(2, 1)
    model.train(X_train, y_train)
    p = model.forward(X_test)
    print(f'mean={2}, cov={1}, Accuracy: {accuracy_score(p, y_test)}')
```

Listing 4: Training and accuracy

1.4 Task 4

We took $\boldsymbol{\mu} = \{20, 2, 0.2, 0.02\}$ and $\boldsymbol{\sigma} = \{10, 1, 0.1, 0.01\}$.

```
means, covs = [20, 2, 0.2, 0.02], [10, 1, 0.1, 0.01]
for mean in means:
    for cov in covs:
        model = Perceptron(n_inputs=2, max_epochs=20, learning_rate=0.01)
        X_train, X_test, y_train, y_test = generate(mean, cov)
        model.train(X_train, y_train)
        p = model.forward(X_test)
        print(f'mean={mean}, cov={cov}, Accuracy: {accuracy_score(p,
    y_test)}')
```

Listing 5: Experiment with different sets of points

The result shows in **Table 1**.

We found if the means are too close (which means μ is too small), or their variances is too high (which means σ is too big), the accuracy score will be much lower.

μ \ σ	**10**	**1**	**0.1**	**0.01**
20	1.0	1.0	1.0	1.0
2	0.875	1.0	1.0	1.0
0.2	0.7	0.35	0.5	0.5
0.02	0.275	0.65	0.45	0.5

Table 1: Accuracy of different means and variances

1.5 Theoretical Analysis

The perceptron is a binary classifier with:

1. Input: $\boldsymbol{x} \in \mathbb{R}^n$
2. Weighed vector: $\boldsymbol{w} \in \mathbb{R}^n$
3. Bias: b
4. Output:

$$\hat{\boldsymbol{y}}_i = \begin{cases} 1, & \text{if } \boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b \geq 0, \\ -1, & \text{otherwise.} \end{cases}$$

For each training example, we have forward and backward process:

1. Forward:
$$\hat{\boldsymbol{y}}_i = \operatorname{sign}(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{x} + b).$$
2. Backward: Update $\boldsymbol{w}$ and $\boldsymbol{b}$ by
$$\begin{aligned} \boldsymbol{w} &\leftarrow \boldsymbol{w} + \eta(\boldsymbol{y}_i - \hat{\boldsymbol{y}}_i)\boldsymbol{x}_i, \\ b &\leftarrow b + \eta(y_i - \hat{y}_i). \end{aligned}$$

2 Part II: The mutli-layer perceptron (60 pts)

2.1 Task 1 & Theoretical Analysis

1. Hidden Layers (Layer 1 to $N-1$): **ReLU**
$$x^{(l)} = \max(0, \tilde{x}^{(l)}), \qquad \frac{\partial x^{(l)}}{\partial \tilde{x}^{(l)}} = \mathbf{1}_{\tilde{x}^{(l)} > 0}.$$
2. Output Layer (Layer N): **Softmax**
$$x^{(N)} = \frac{\exp(\tilde{x}^{(N)})}{\sum_{i=1}^{d_N} \exp(\tilde{x}^{(N)})_i}, \qquad \frac{\partial x_i^{(N)}}{\partial \tilde{x}_j^{(N)}} = x_i^{(N)}(\delta_{ij} - x_j^{(N)}).$$

3. Loss function: **Cross-Entropy**

$$L(x^{(N)}, t) = -\sum_i t_i \log x_i^{(N)}, \quad \frac{\partial L}{\partial \tilde{x}^{(N)}} = x^{(N)} - t.$$

4. **Backpropagation**: For the last linear layer L, the gradient of the loss with respect to the output $\hat{y}$ is:

$$\frac{\partial L}{\partial \hat{y}} = \frac{\partial L}{\partial x^{(L)}}$$

Then, the gradient w.r.t. weights is:

$$\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial \tilde{x}^{(l)}} (x^{(l-1)})^{\mathrm{T}}.$$

The gradient w.r.t. bias is:

$$\frac{\partial L}{\partial b^{(l)}} = \sum_i \frac{\partial L}{\partial \tilde{x}_i^{(l)}}.$$

Finally, we have the gradient w.r.t. the previous layer:

$$\frac{\partial L}{\partial x^{(l-1)}} = (W^{(l)})^{\mathrm{T}} \frac{\partial L}{\partial \tilde{x}^{(l)}}.$$

For hidden layers with ReLU:

$$\frac{\partial L}{\partial \tilde{x}^{(l)}} = \frac{\partial L}{\partial x^{(l)}} \odot \mathbf{1}_{\tilde{x}^{(l)} > 0}.$$

```python
class ReLU(object):
    def forward(self, x):
        self.x = x
        return np.maximum(0, x)

    def backward(self, dout):
        dx = dout * (self.x > 0).astype(float)
        return dx
```

Listing 6: ReLU function

```python
class SoftMax(object):
    def forward(self, x):
        shift_x = x - np.max(x, axis=1, keepdims=True)
        exp_x = np.exp(shift_x)
        return exp_x / np.sum(exp_x, axis=1, keepdims=True)

    def backward(self, dout):
        return dout
```

Listing 7: Softmax function

```python
class CrossEntropy(object):
    def forward(self, x, y):
        pred_clipped = np.clip(x, 1e-12, 1.-1e-12)
        loss = -np.sum(y * np.log(pred_clipped), axis=1)
        return np.mean(loss)

    def backward(self, x, y):
        return x-y
```

Listing 8: Cross-Entropy function

2.2 Task 2

We use `make_moons()` to generate the training and testing data.

```python
dnn_hidden_units = [int(x) for x in dnn_hidden_units.split(',')]
X, y = make_moons(n_samples=1000, shuffle=True)
n_values = np.max(y) + 1
y = np.eye(n_values)[y]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    shuffle=True)

model = MLP(n_inputs=2, n_hidden=dnn_hidden_units, n_classes=2)
loss = CrossEntropy()
```

Listing 9: Load data and initialize the MLP model

Line 3 and line 4 is to transform the 0/1 classification to one-hot encoding possibility.

```python
    for step in range(max_steps):
        y = model.forward(X_train)
        l = loss.forward(y, y_train)
        dout = loss.backward(y, y_train)
        model.backward(dout)
        model.head.params['weight'] = model.head.params['weight'] -
    learning_rate * model.head.grads['weight']
        model.head.params['bias'] = model.head.params['bias'] -
    learning_rate * model.head.grads['bias']
        for linear in model.linears:
            linear.params['weight'] = linear.params['weight'] -
    learning_rate * linear.grads['weight']
            linear.params['bias'] = linear.params['bias'] - learning_rate
    * linear.grads['bias']

        if step % eval_freq == 0 or step == max_steps - 1:
            y = model.forward(X_test)
            print(f"Step: {step}, Loss: {l}, Accuracy: {accuracy(y, y_test
    )}")
```

Listing 10: Implement the training loop

2.3 Task 3

Simply run `mlp.ipynb` in `Part_2` folder. Here we set `MAX_EPOCHS_DEFAULT = 1500`. The visualized result is as belows.

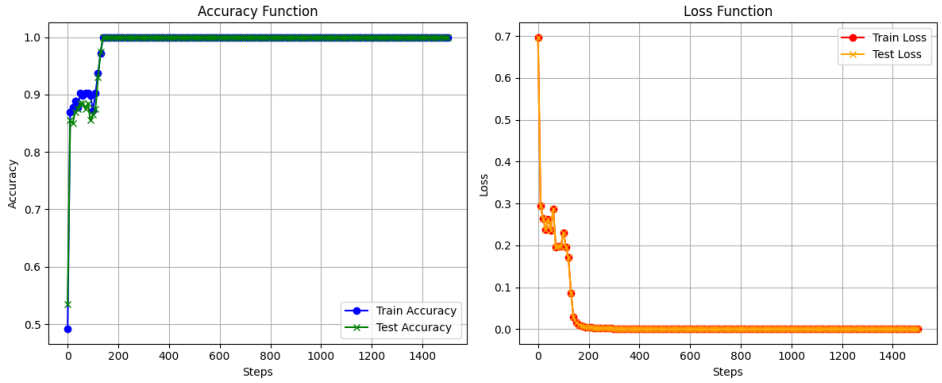

Figure 1: Accuracy function and loss function

From **Figure 1**, we know that after about **180 steps**, the accuracy and loss both converge.

3 Part III: Stochastic gradient descent (15 pts)

3.1 Task 1

Notion: There are two more parsers added in `train_mlp_numpy.py` in `Part_3`, so the way to run `train_mlp_numpy.py` becomes

```
python train_mlp_numpy.py --dnn_hidden_units 20 --learning_rate 0.01 --
    max_steps 1500 --eval_freq 10 --sgd True --batch_size 1
```

Listing 11: How to run Part 3

where the default `sgd` is `True`, the default `batch_size` is `1`.

When using SGD, the training loop becomes

```
batches = shuffle_and_batch(X_train, y_train, batch_size=batch_size)
for X, y in batches:
    X = X.reshape((1, -1))
    y_hat = model.forward(X)
    l = loss.forward(y_hat, y)
    dout = loss.backward(y_hat, y)
    model.backward(dout)
    model.head.params['weight'] = model.head.params['weight'] -
    learning_rate * model.head.grads['weight']
    model.head.params['bias'] = model.head.params['bias'] - learning_rate
    * model.head.grads['bias']
    for linear in model.linears:
        linear.params['weight'] = linear.params['weight'] - learning_rate
    * linear.grads['weight']
```

```
12          linear.params['bias'] = linear.params['bias'] - learning_rate *
        linear.grads['bias']
```

Listing 12: sgd

3.2 Task 2

We took `batch_size` $= \{1, 10, 50, 200, 500\}$. Simply run `train_mlp.ipynb` in `Part_3` folder, and the visualized result is as belows:

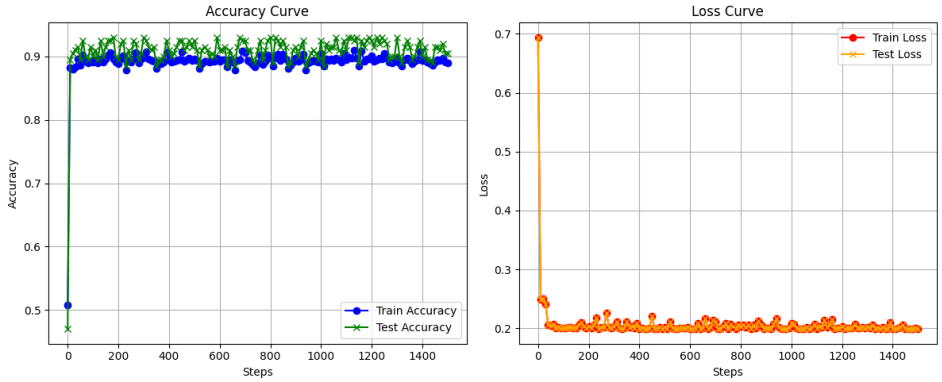

Figure 2: Accuracy and loss function when batch size is 1

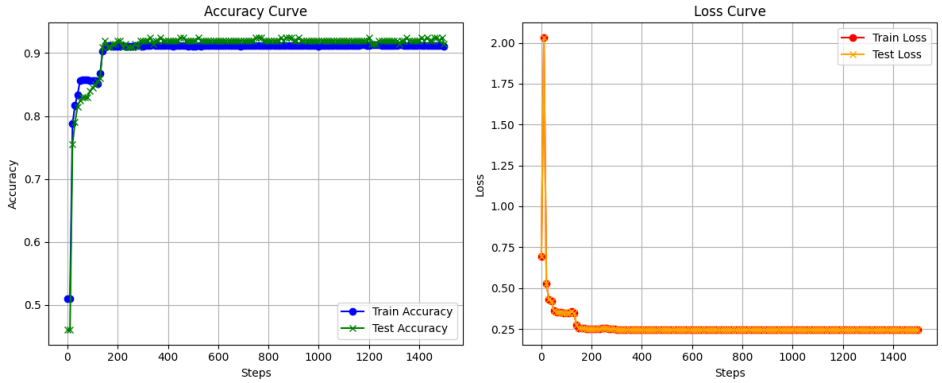

Figure 3: Accuracy and loss function when batch size is 10

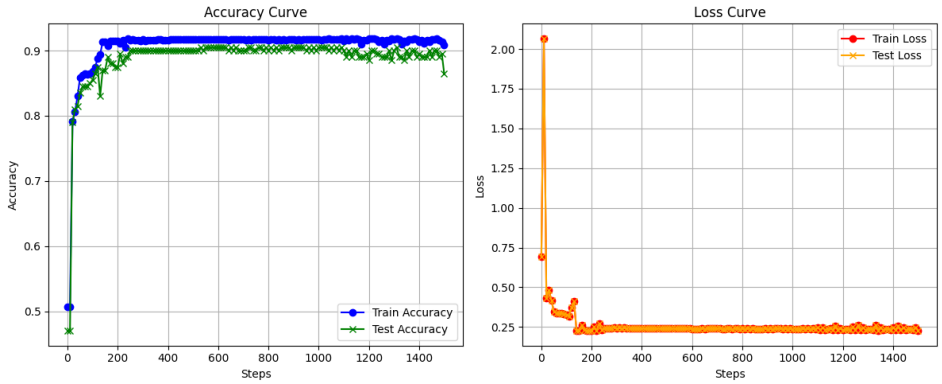

Figure 4: Accuracy and loss function when batch size is 50

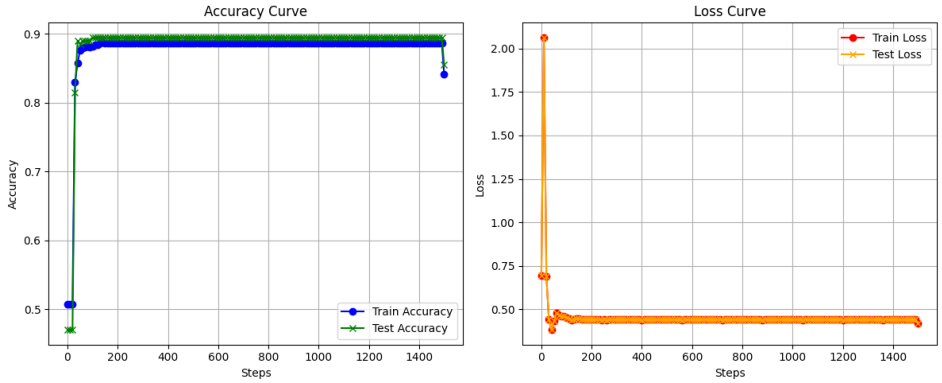

Figure 5: Accuracy and loss function when batch size is 200

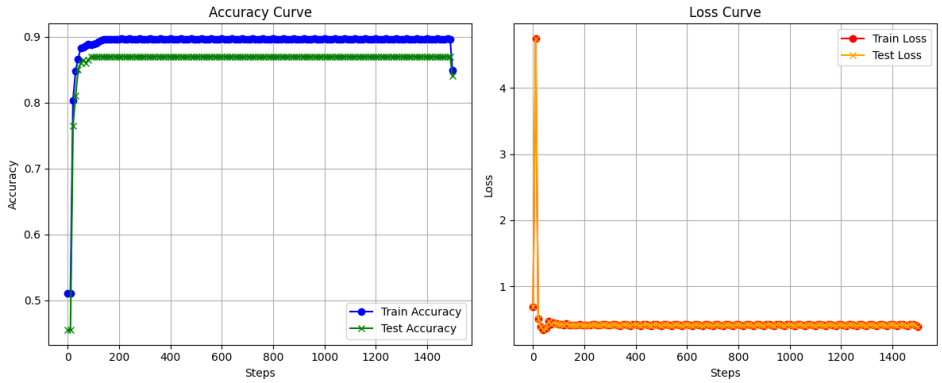

Figure 6: Accuracy and loss function when batch size is 500

We found that when the batch size gets larger, the oscillation of accuracy and loss will decrease. As a result, setting batch_size= 200 is relatively the best choice.