Report for SDM274 Assgn05 MLP

Author: Ziyuan Li, 12211225

This document is created as a report for SDM274, Sustech.

Background

Multiple Layer Perceptron(MLP) is a type of neural network, primarily used in supervised learning. It's made up of multiple layers of nodes, or neurons, organized into an input layer, one or more hidden layers, and an output layer. Each neuron is connected to every neuron in the previous and following layer (fully connected), and each connection has an associated weight, which is learned during training.

It was initially invented in the 1980s, aimed at addressing problems where linear models were insufficient. By introducing non-linear activation functions and organizing neurons into multiple fully connected layers, MLPs allowed for more sophisticated feature extraction and data representation. As deep learning evolved, MLPs were largely surpassed by more specialized architectures like CNN for image and spatial data, and RNNs for sequential tasks such as language processing. Despite this, MLPs are still widely used in scenarios where simplicity and computational efficiency are prioritized, or as subcomponents in complex architectures.

Furthermore, the concept of the fully connected layer remains fundamental across various modern deep learning models, including CNNs, RNNs, and transformers, where they play a crucial role in transforming high-dimensional data, interpreting features, and synthesizing final outputs, illustrating their enduring importance in deep learning.

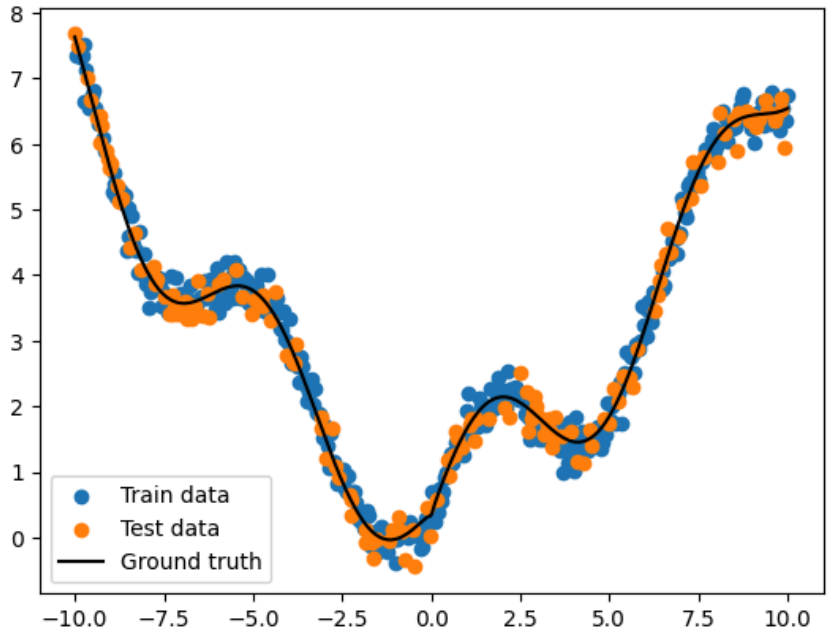
Code Design

Designing the nonlinear function

```
def complex_nonlin_func(x):
    return (
        np.sin(x) +
        0.3 * np.exp( np.abs(x)**0.5 )
    )
```

Here I introduced a nonlinear function composed of a sine and an exponential part. It is nonlinear, and not symmetrical, which is hard to approximate using a linear model.

I generated 500 points on the line, spanning from -10 to 10, and also split the data into (0.7, 0.3) for training and testing.




```
y = y + 0.2 * np.random.normal(size=x.size)
```

I added a random noise with the amplitude of 0.2 to the signal.

Modules.py resembling torch.nn

I designed a family of class that **resembles the Pytorch Torch.nn library**. This library is well designed for its modularity, ease of use, and flexibility. User can design a network by simply declaring and stacking layers in a list. An **example code snippet of torch.nn code** is provided below:

```
net = nn.Sequential()
net.add(nn.Dense(256, activation='relu'),
        nn.Dense(10))
net.initialize(init.Normal(sigma=0.01))
```

modularity

In my designed class, I implement the modularity with **a base class** `Module`. The `Module` class contains shared attributes and methods, like `parameters` (to manage learnable parameters) and caches (`output_cache`, `input_cache`) for forward and backward passes.

```
class Module:

    def __init__(self):
        self.parameters = []
        self.output_cache = None
        self.input_cache = None
        self.isInference = False

    def forward(self):
        raise NotImplementedError

    def backward(self):
        raise NotImplementedError

    def parameters(self):
        return self.parameters
```

Adding new types of layers or operations is straightforward—I can inherit from `Module` and override the `forward` and `backward` methods as needed.

caching intermediate values

Implemented layers **caches intermediate values** needed for backpropagation.

For example, `Linear` caches `input_cache` to store inputs during the forward pass. This allows the layer to reuse these inputs during the backward pass for gradient calculations.

forward and backward methods

Each module implements its **own `forward` and `backward` methods**, essential for differentiable operations in neural networks.

- In `forward`, each layer computes its output based on the input, caching intermediate results if necessary.
- In `backward`, the layers calculate the gradients with respect to their inputs and parameters. For example, `Linear` calculates gradients for weights (`dW`), biases (`db`), and the input (`dX`) to propagate gradients to previous layers.

How parameters are stored

Layers with learnable parameters, such as `Linear`, define and manage their parameters (weights `W` and biases `b`). These parameters are stored in a dictionary (`self.parameters`) .

> However this design may lead to unwanted re-train if lack proper management

managing parameters

- The library introduces an `isInference` flag to distinguish between **training and inference modes**. When `isInference` is set to `True`, the forward pass bypasses caching and specific training-only operations.
- The `reset` method in `Linear` layers reinitializes the weights and biases. This can be useful when experimenting with different initializations and hyperparameters.

The source code is provided in the appendix

Nonlinear Regression with MLP

Design of the Network

I defined a MLP class which inherits from the Module class (which might not be the best choice but it works).

Initialization

First, initialize the `MLP` class with the desired layers, number of epochs, learning rate, input shape, and output shape. Here is the structure used for training.

```
layers = [
    Linear(input_size=1, output_size=64),
    ReLU(),
    Linear(input_size=64, output_size=128),
    ReLU(),
    Linear(input_size=128, output_size=256),
    ReLU(),
    Linear(input_size=256, output_size=1)
]

mlp = MLP(layers=layers, epochs=1000, lr=0.01, input_shape=1, output_shape=1)
```

To train the model, call the `train_BGD` method with the input data X and target data `Y`.

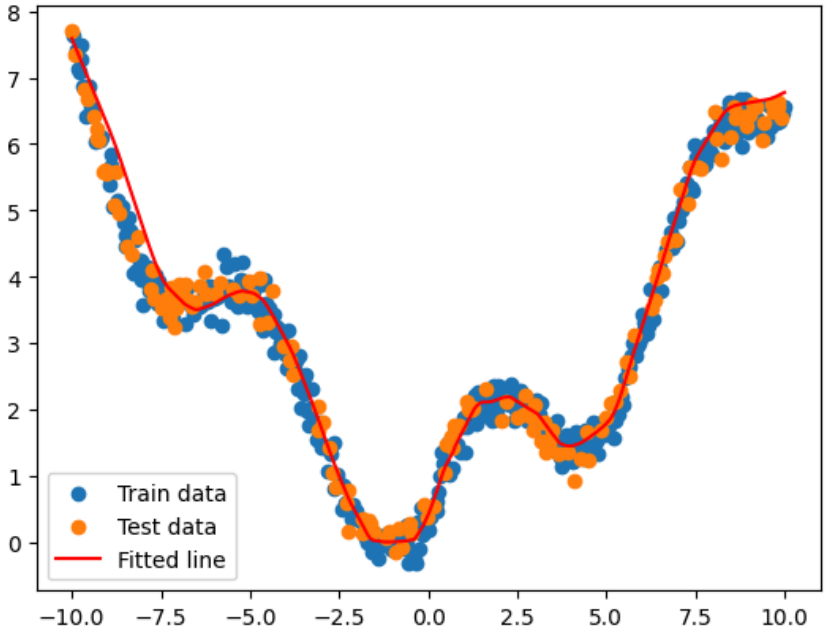
```
X = np.array([...])  # Input data
Y = np.array([...])  # Target data

mlp.train_BGD(X, Y)
```

Training Steps

1. **Forward Pass**: The `forward` method is called to compute the output of the model for the given input X.
2. **Loss Calculation**: The loss is computed using Mean Squared Error (MSE) between the predicted output and the target `Y`.
3. **Backward Pass**: The `backward` method is called to compute the gradients of the loss with respect to the model parameters.
4. **Parameter Update**: The `update_params` method updates the model parameters using the computed gradients and the learning rate.
5. **Epoch Loop**: Steps 1-4 are repeated for the specified number of epochs.

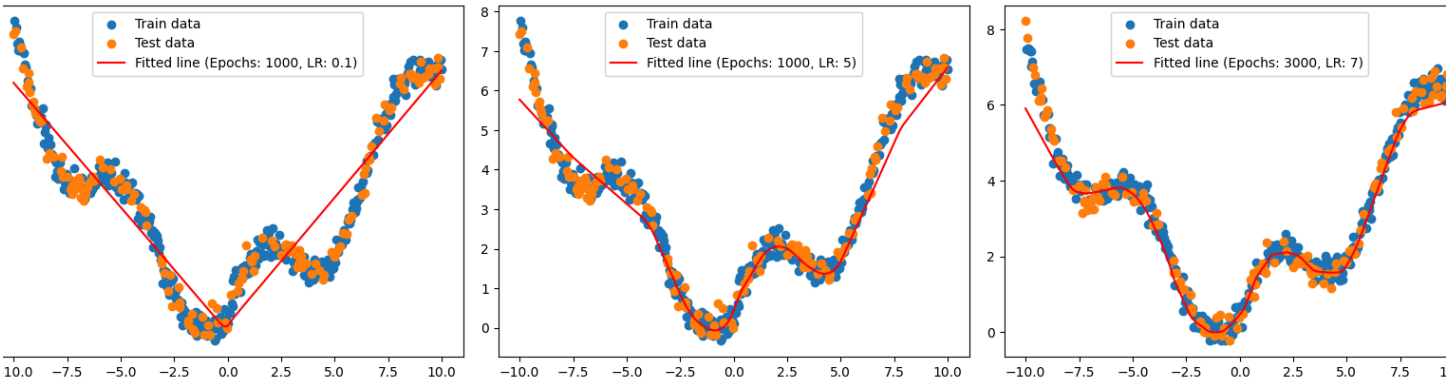
Code provided in the appendix.

Fitting




```
# Creates the model
mlp_model = MLP(layers, epochs=5000, lr=7)
if not trained:
    # Train the model with the generated data
    mlp_model.train_BGD(X_train, y_train)
    trained = True
elif trained:
    mlp_model.reset()
    mlp_model.train_BGD(X_train, y_train)
```

This image is the result of successful fitting. For the initial version of the Multi-Layer Perceptron (MLP), which lacked any form of regularization, I had to set hyperparameters to rather extreme values in order to achieve satisfactory training performance. This process significantly increased the training time, requiring approximately 2 minutes to complete, which is notably longer compared to the simpler single-neuron models used previously.



Some comparations. The MLPs need significantly more resources to train.

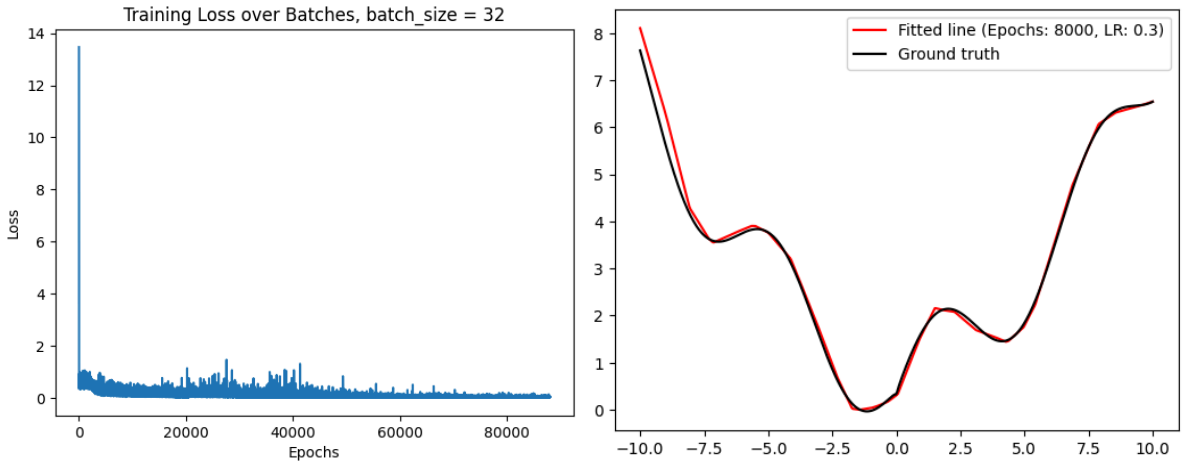
Optimization of structure and training strategy

Empirically, I discovered that a shallower layer can achieve approximately the same result, with lesser demand on training resources.

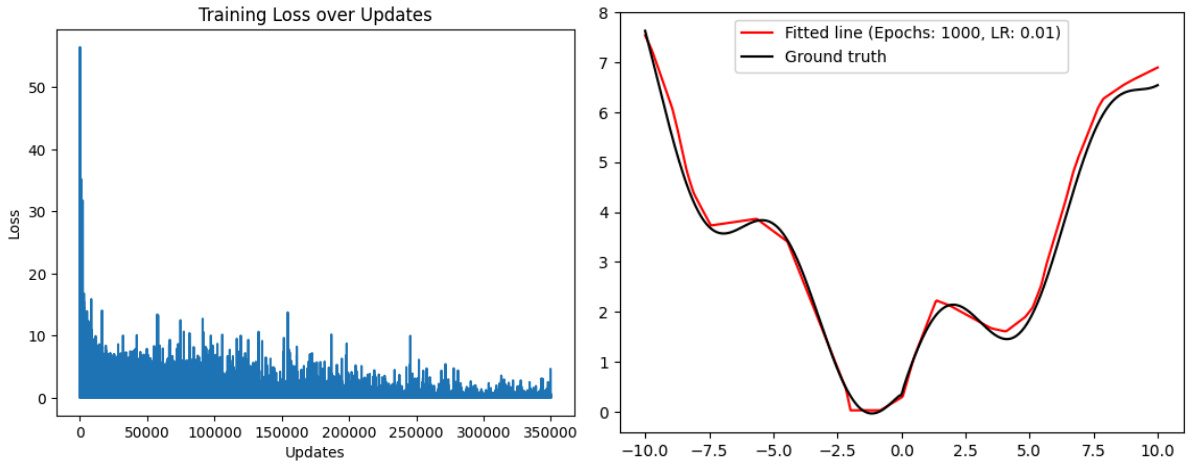
```
layers = [
    Linear(input_size=1, output_size=64),
    ReLU(),
    Linear(input_size=64, output_size=128),
    ReLU(),
    Linear(input_size=128, output_size=1)
]
```

On my system with a AMD Ryzen 7 7840HS@25W, it takes only 9.7 seconds to train, and achieved fairly good results on the regression task.

Results Implemented by the Mini-Batch Gradient Descent (MBGD) Method



Results Implemented by Stochastic Gradient Descent (SGD) Method



Empirically discovered, MBGD with bs=32, epochs=8000, lr = 0.3 is a good set of hyperparameters. We will do the cross validation part on it.

Cross Validation

I implemented a k-fold cross validation method to evaluate the overall model.

```
def train_k_fold(self, X, Y, k=5, epochs=None, lr=None):
        if epochs is None:
            epochs = self.epochs
        if lr is None:
            lr = self.lr

        m = X.shape[0]
        fold_size = m // k # integer division, i.e. 103 // 5 = 20
        indices = np.arange(m) # 0, 1, 2, ..., m-1
        np.random.shuffle(indices) # shuffle the indices, train randomly

        fold = 1
        for i in range(k): # iterate over the folds, totally we train for k times
            val_start = i * fold_size
            val_end = val_start + fold_size if i < k - 1 else m # if it's the last fold, use the remaining data

            val_indices = indices[val_start:val_end] # takes the validation indices
            train_indices = np.concatenate([indices[:val_start], indices[val_end:]]) # takes all the indices except the validation indices

            X_train = X[train_indices]
            Y_train = Y[train_indices]
            X_val = X[val_indices]
            Y_val = Y[val_indices]

            self.reset()
            self.train_MBGD(X_train, Y_train, epochs=epochs, lr=lr) # we still use the mini-batch gradient descent, with a default bs of 32.

            A_val = self.inference(X_val)
            loss = mse_loss(A_val, Y_val)
```

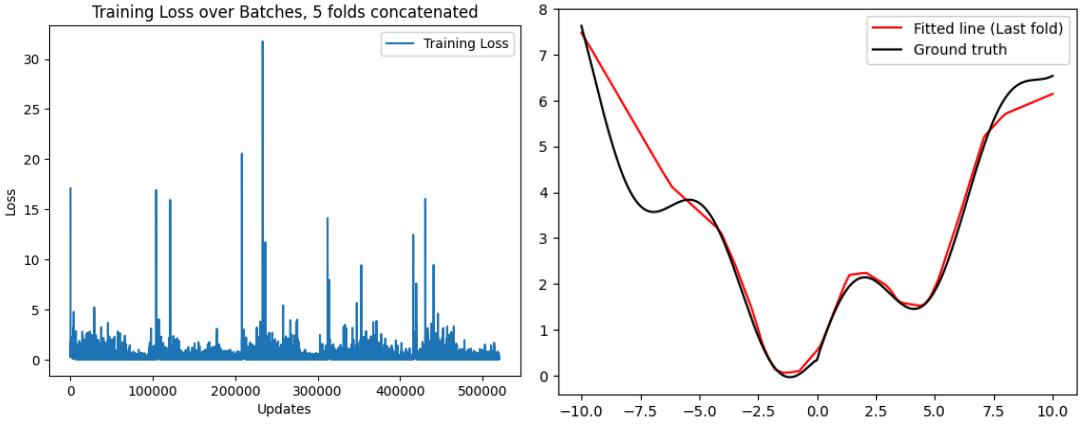
```
            self.validation_loss.append(loss)

            print(f'Fold {fold}, Loss: {loss}')
            fold += 1

```

It shuffles the x and y data and split them into 5 parts. The method then iterates over the number of folds(k). For each fold, it determines the start and end indices for the validation set. The remaining indices are used for the training set. This ensures that each fold has a unique validation set and the rest of the data is used for training.

Within each fold, the method resets the model parameters to ensure that the training starts from scratch. It then trains the model using MBGD on the training set. After training, the model's performance is evaluated on the validation set using the inference method. The MSE loss is calculated for the validation set and appended to the [validation_loss list. The loss for each fold is printed to provide feedback on the training process.

```
Training MBGD: 100%|██████████| 8000/8000 [00:11<00:00, 679.42it/s]
Fold 1, Loss: 0.135129258476114
Training MBGD: 100%|██████████| 8000/8000 [00:11<00:00, 704.24it/s]
Fold 2, Loss: 0.37937692480483626
Training MBGD: 100%|██████████| 8000/8000 [00:11<00:00, 703.15it/s]
Fold 3, Loss: 0.05067628846152356
Training MBGD: 100%|██████████| 8000/8000 [00:11<00:00, 716.51it/s]
Fold 4, Loss: 0.7495206586337292
Training MBGD: 100%|██████████| 8000/8000 [00:11<00:00, 704.32it/s]
Fold 5, Loss: 0.3724279561516599
```

Training of 5 folds



Trying different Hyperparameters

Test 1

```
Average validation loss: 0.3374262173055726
Hyperparameters are: Epochs: 8000, LR: 0.4
Model structure: [<modules.Linear object at 0x7fd263aa6670>, <modules.ReLU object
at 0x7fd263aa68b0>, <modules.Linear object at 0x7fd263aa6190>, <modules.ReLU object
at 0x7fd263aa6070>, <modules.Linear object at 0x7fd263aa64c0>]
```

Test 2 - Simpler Nets

```
Average validation loss: 0.46817570040655837
Hyperparameters are: Epochs: 8000, LR: 0.4
Model structure:
Layer 1: Linear
Input size: 1, Output size: 128
Layer 2: ReLU
Layer 3: Linear
Input size: 128, Output size: 1
```

Test 2 - Simpler Nets, extensive training

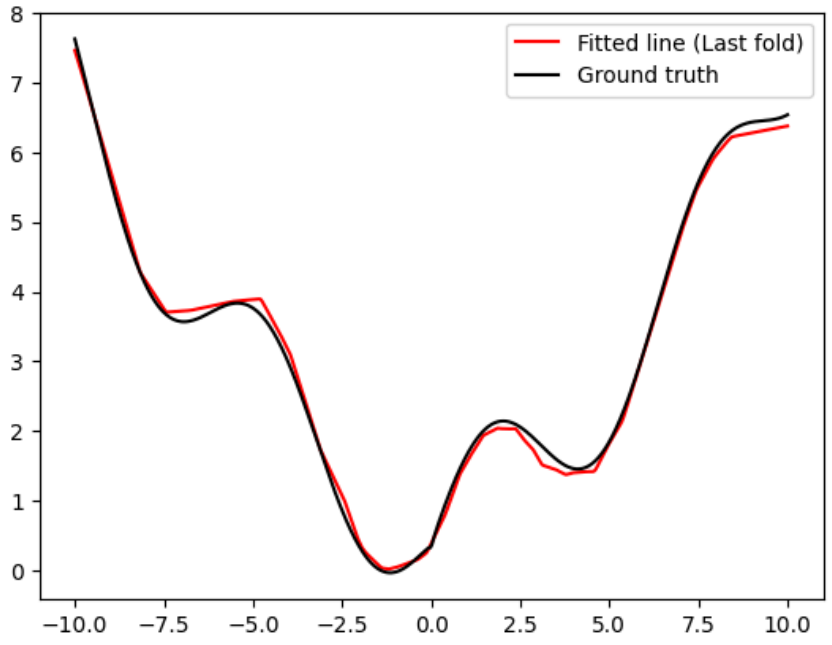
```
Average validation loss: 0.3243354289653034
Hyperparameters are: Epochs: 15000, LR: 0.4
Model structure:
Layer 1: LinearInput size: 1, Output size: 128
Layer 2: ReLU
Layer 3: LinearInput size: 128, Output size: 1
```

Test 3 - Deeper Nets

```
Average validation loss: 0.09484475132123227
Hyperparameters are: Epochs: 8000, LR: 0.4
Model structure:
Layer 1: LinearInput size: 1, Output size: 128
Layer 2: ReLU
Layer 3: LinearInput size: 128, Output size: 512
Layer 4: ReLU
Layer 5: LinearInput size: 512, Output size: 1

```

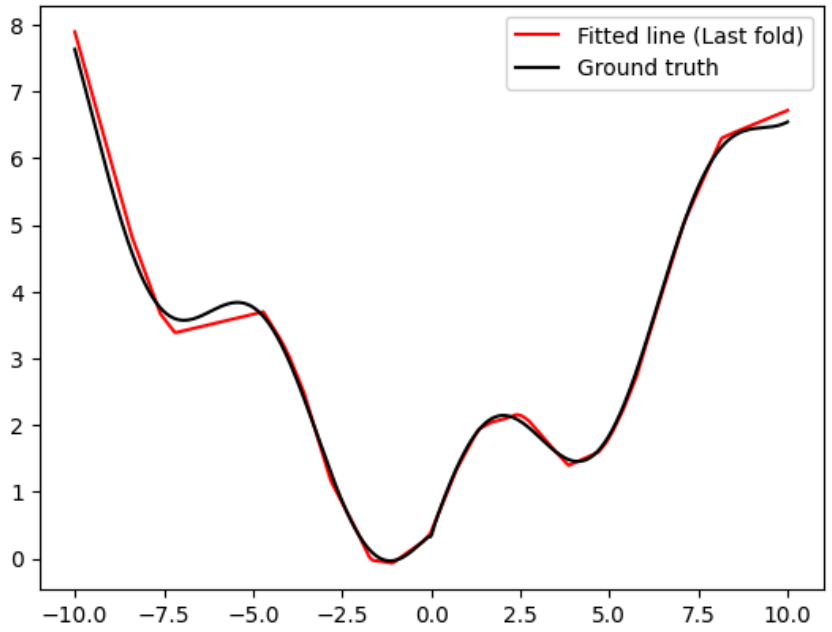
Note: this network took 23m39.7s to finish the 5-fold cross validation.

The result looks pretty good though.



Test 4 - Simply Better Hyperparams

```
Average validation loss: 0.06755265032590395
Hyperparameters are: Epochs: 8000, LR: 0.1
Model structure:
Layer 1: LinearInput size: 1, Output size: 64
Layer 2: ReLU
Layer 3: LinearInput size: 64, Output size: 128
Layer 4: ReLU
Layer 5: LinearInput size: 128, Output size: 1
```



Lots easier to train.

Summary

- Deeper and wider nets are more difficult to train.

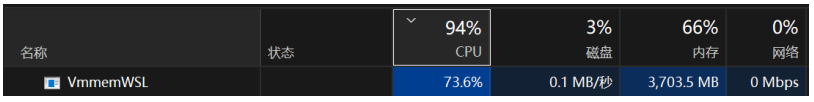
```
eval_layers = [
        Linear(input_size=1, output_size=128),
        ReLU(),
        Linear(input_size=128, output_size=256),
        ReLU(),
        Linear(input_size=256, output_size=512),
        ReLU(),
        Linear(input_size=512, output_size=1)
    ]
```

with a network like this, the speed is about **10~20 it/s**, while using a simpler net in simpler nets, the speed can go up to **~1700it/s**.

```
Training MBGD:   0%|          | 0/8000 [00:00<?, ?it/s]
Training MBGD: 100%|██████████| 8000/8000 [08:42<00:00, 15.32it/s]
Fold 1, Loss: 0.09874567588570843
Training MBGD:   0%|          | 0/8000 [00:00<?, ?
it/s]/tmp/ipykernel_29773/4177186514.py:4: RuntimeWarning: overflow encountered in
square
  return np.mean((predicted - target) ** 2)
/root/sdm274/assgn05_MLP/modules.py:45: RuntimeWarning: overflow encountered in
matmul
  dW = (dZ.T @ X) / m # dZ.T ( output_size, m) @ X (m, input_size) = dW
(output_size, input_size)
/root/sdm274/assgn05_MLP/modules.py:90: RuntimeWarning: invalid value encountered
in multiply
  dZ = dA * (self.output_cache > 0).astype(float)
/root/sdm274/assgn05_MLP/modules.py:45: RuntimeWarning: invalid value encountered
in matmul
  dW = (dZ.T @ X) / m # dZ.T ( output_size, m) @ X (m, input_size) = dW
(output_size, input_size)
Training MBGD:  55%|█████▌     | 4396/8000 [04:29<03:41, 16.29it/s]
```

It is so resource-hungry that it might trigger **overflows**.

- Deep nets capture nonlinearities better, when properly trained.
- Shallow networks do not have enough layers to learn complex hierarchical features, making them less capable of capturing intricate nonlinear relationships. However, If you train them enough, you might have it yield a small validation loss, with the danger of overfitting.
- The improvement of accuracy is not proportionately big when you go from a normal model to a large model. The more accuracy you want, the more resources it takes to train and run a model.
- Hardware problem:

  What the program does is do tons of matrix multiplications for the training. It can cause the CPU to be fully loaded. But those matrix multiplications can be well accelerated by a GPU, which is suitable for computing simple calculations for millions of times.

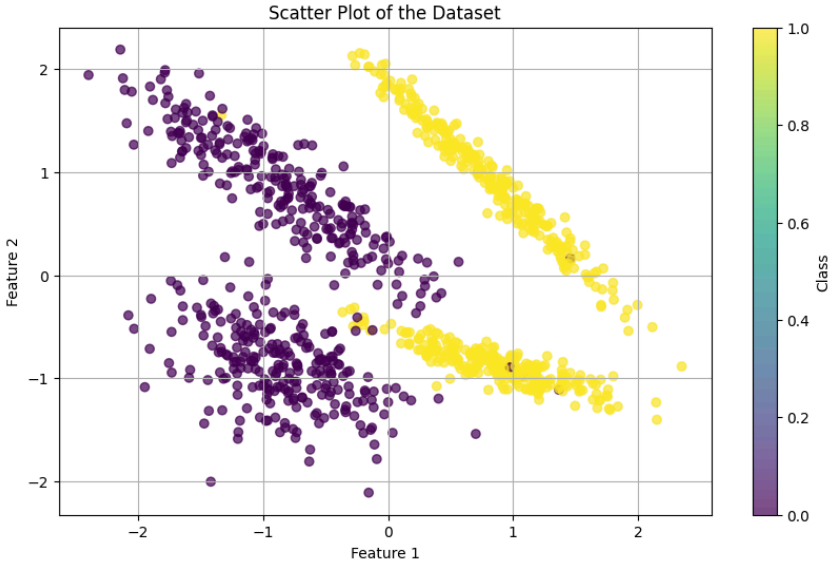
Classification Using MLP

Data gathering

We generated a set of 2-feature binary classification data using sklearn.

```
from sklearn.datasets import  make_classification

X,y = make_classification(n_samples=1000, n_features=2,
n_informative=2,n_redundant=0, n_repeated=0, n_classes=2,
n_clusters_per_class=2,class_sep=1.5, random_state=17)


```

```
Training data shape: (800, 2)
Testing data shape: (200, 2)

```



I defined an MLPClassifier class that inherits from the wrapped MLP class. This structure ensures reusability.

```python
class MLPClassifier(MLP):

    def __init__(self, layers, epochs=1000, lr=0.01, input_shape=1, output_shape=1):
        super().__init__(layers, epochs, lr, input_shape, output_shape)
        self.threshold = 0.5

    def get_loss(self, y_true, y_pred):
        y_pred = y_pred.reshape(-1, 1)
        # Clip predictions to prevent log(0)
        y_pred = np.clip(y_pred, 1e-12, 1. - 1e-12)
        # Compute cross-entropy loss
        return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))

    def get_loss_grad(self, y_pred, y_true):
        # The loss here is the cross-entropy loss
        # It accepts matrix with the shape (x, n_classes)
        # where x is the number of samples and n_classes is the number of classes

        y_pred = y_pred.reshape(-1, 1)
        # Clip predictions to prevent division by zero
        y_pred = np.clip(y_pred, 1e-12, 1. - 1e-12)
        # Compute gradient of cross-entropy loss
        return (y_pred - y_true) / (y_pred * (1 - y_pred))

    def predict(self, X):
        # Forward pass
        y_pred = self.forward(X)

        # Convert the output to binary
        return (y_pred > self.threshold).astype(int)


```

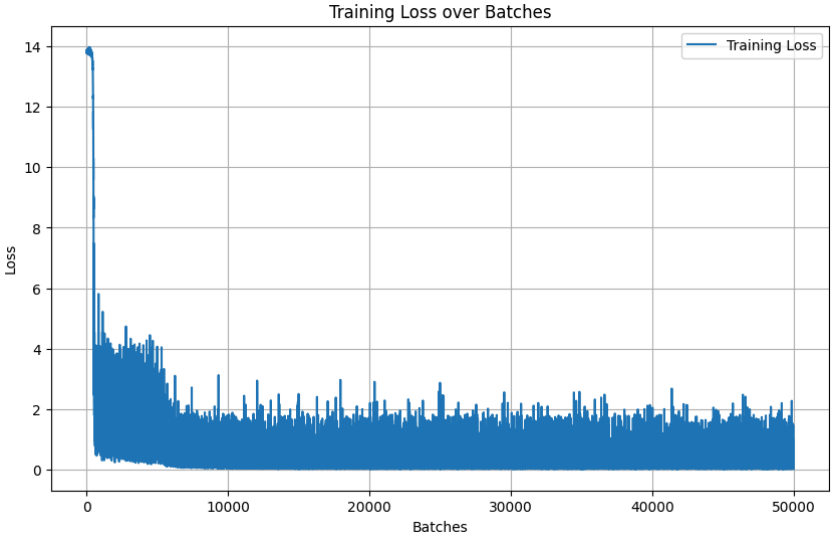
```
33
```

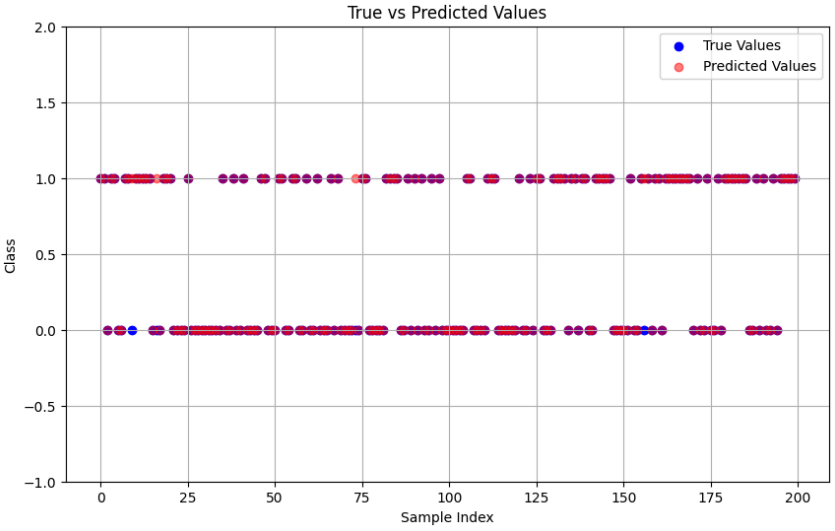
I reused the model structure for the nonlinear function training. It balances performance and resource.

```
layers = [
        Linear(input_size=2, output_size=64),
        ReLU(),
        Linear(input_size=64, output_size=128),
        ReLU(),
        Linear(input_size=128, output_size=1),
        Sigmoid()
    ]

```

I used MBGD in training, reaching about 400it/s with a good accuracy.

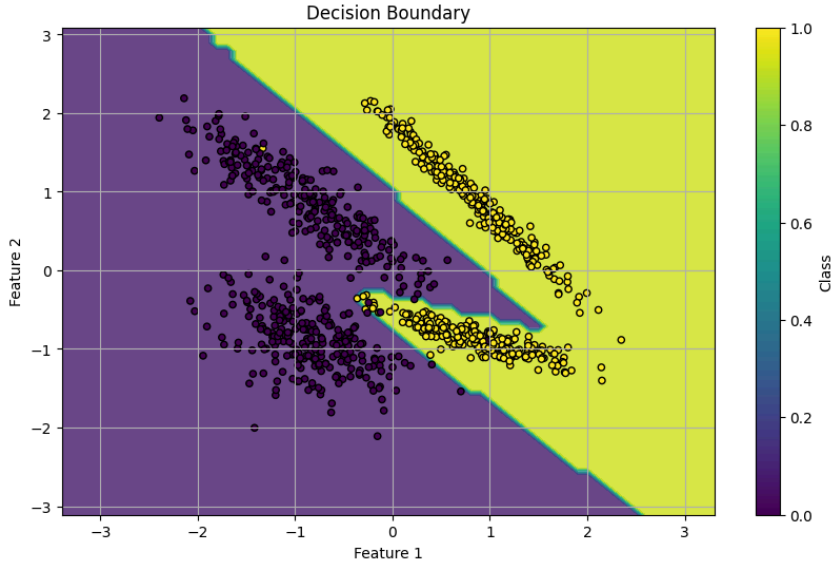
```
Training MBGD: 100%|██████████| 2000/2000 [00:07<00:00, 284.20it/s]
```





The model achieved high F1 score on this dataset.

```
Accuracy: 0.975
Recall: 1.0
Precision: 0.9484536082474226
F1 Score: 0.9735449735449735
```



Above is the decision boundary of the trained MLP. As shown in the graph the boundary is **a folded line** (like a lightning bolt) instead of a straight line. This indicates that the MLP model has learned a **nonlinear decision boundary**. Unlike linear classifiers, which can only separate classes with straight lines or planes, an MLP with at least one hidden layer can model complex, nonlinear relationships between features.

Shown below is some sets cross-validation result.

```
Average validation loss: 0.8235056154333691
Hyperparameters are: Epochs: 3000, LR: 0.1
Model structure:
Layer 1: LinearInput size: 30, Output size: 64
Layer 2: ReLU
Layer 3: LinearInput size: 64, Output size: 128
Layer 4: ReLU
Layer 5: LinearInput size: 128, Output size: 1
Layer 6: Sigmoid
```

```
Average validation loss: 0.7517043588174628
Hyperparameters are: Epochs: 2000, LR: 0.1
Model structure:
Layer 1: LinearInput size: 30, Output size: 64
Layer 2: ReLU
Layer 3: LinearInput size: 64, Output size: 128
Layer 4: ReLU
Layer 5: LinearInput size: 128, Output size: 1
Layer 6: Sigmoid

F1 Score: 0.9859154929577465
```

```
Average validation loss: 0.8006151770380503
Hyperparameters are: Epochs: 2000, LR: 0.01
Model structure:
Layer 1: LinearInput size: 30, Output size: 64
Layer 2: ReLU
Layer 3: LinearInput size: 64, Output size: 128
Layer 4: ReLU
Layer 5: LinearInput size: 128, Output size: 1
Layer 6: Sigmoid

F1 Score: 0.9929078014184397
```

We have almost come to the best of this structure, achieving very high F1 scores.

Summary

The results and findings provide insights into the MLP's representational power, demonstrating the ability of them to approximate nonlinear functions and capture complex patterns in data.