HCMUS - VNUHCM / FIT / Computer Vision & Cognitive Cybernetics Department

Digital image & video processing - LQN

Student ID: 21127690

Student name: Ngo Nguyen Thanh Thanh

Report: PRACTICE WITH PYTORCH #1 (Keyword: PyTorch)

I. Evaluation summary:

No	Task	Details of Implementation	Completion Percentage (%)
1	**Implement FFNN Code**	Implemented the FFNeuralNetwork class with methods: __init__, forward, backward, train, predict, save_weights, load_weights.	**100%**
2	**Implement Sigmoid Function**	Implemented the Sigmoid function and its derivative for use in hidden layers and the output layer.	**100%**
3	**Forward Propagation**	Processed input data through the layers of the model, from input to output, applying the sigmoid activation function.	**100%**
4	**Backward Propagation**	Updated weights using the error between the predicted output and the actual label, adjusting weights W1,W2W_1, W_2W1,W2.	**100%**
5	**Train the Model**	Trained the model for 1000 iterations using a learning rate η=0.1\eta = 0.1η=0.1, calculated the error after each iteration.	**100%**
6	**Predict New Data**	Normalized the new input data and used the predict method to generate prediction results.	**100%**
7	**Save and Load Weights**	Saved and loaded the model's weights after training using torch.save() and torch.load().	**100%**
8	**Run Experiments on Sample Data**	Tested the implementation on the sample dataset X and y, printed the loss during training, and predicted new data.	**100%**
9	**Change the basic parameters of the network and conduct conduct experiments to observe close to the results.**	learning rate, class size, number of classes, class type, number of epochs..	**100%**

II. List of features and file structure:

File structure:

1. **nn_simple.py**:
    - Contains the FFNeuralNetwork class and its related functions.
2. **Test_ffnn.py**:
    - Contains the main logic to train and test the FFNN.

Functions and methods used:

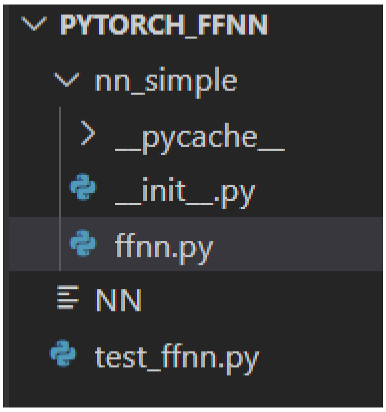
Activation functions:

- **sigmoid(s)**: Computes the sigmoid activation function.
- **sigmoid_derivative(s)**: Computes the derivative of the sigmoid function.

FFNeuralNetwork class:

- **Main methods:**
    - **__init__()**: Initializes network parameters and random weights.
    - **forward(X)**: Performs forward propagation.
    - **backward(X, y, o, rate)**: Performs backward propagation to compute gradients and update weights.
    - **train(X, y, rate)**: Trains the network using forward and backward propagation.
    - **predict(x_predict)**: Makes predictions using new input data.
    - **save_weights(model, path)**: Saves the model weights to a file.
    - **load_weights(path)**: Loads the model weights from a file.

Image proof:



nn_simple > ffnn.py > sigmoid_derivative

```python
# import PyTorch
import torch
# import PyTorch Neural Network module
import torch.nn as nn


# sigmoid activation
def sigmoid(s):
    return 1 / (1 + torch.exp(-s))


# derivative of sigmoid
def sigmoid_derivative(s):
    return s * (1 - s)


# Feed Forward Neural Network class
class FFNeuralNetwork(nn.Module):
    # initialization function
    def __init__(self, ):
        # init function of base class
        super(FFNeuralNetwork, self).__init__()

        # corresponding size of each layer
        self.inputSize = 3
        self.hiddenSize = 4
        self.outputSize = 1

        # random weights from a normal distribution
        self.W1 = torch.randn(self.inputSize, self.hiddenSize)  # 3 X 4 tensor
        self.W2 = torch.randn(self.hiddenSize, self.outputSize)  # 4 X 1 tensor

        self.z = None
```

n_simple > ffnn.py > sigmoid_derivative

```python
class FFNeuralNetwork(nn.Module):
    def __init__(self, ):
        self.z = None
        self.z_activation = None
        self.z_activation_derivative = None

        self.z2 = None
        self.z3 = None

        self.out_error = None
        self.out_delta = None

        self.z2_error = None
        self.z2_delta = None

    # activation function using sigmoid
    def activation(self, z):
        self.z_activation = sigmoid(z)
        return self.z_activation

    # derivative of activation function
    def activation_derivative(self, z):
        self.z_activation_derivative = sigmoid_derivative(z)
        return self.z_activation_derivative

    # forward propagation
    def forward(self, X):
        # multiply input X and weights W1 from input layer to hidden layer
        self.z = torch.matmul(X, self.W1)
        self.z2 = self.activation(self.z)  # activation function
        # multiply current tensor and weights W2 from hidden layer to output layer
        self.z3 = torch.matmul(self.z2, self.W2)
        o = self.activation(self.z3)  # final activation function
```

imple > ffnn.py > sigmoid_derivative

```python
class FFNeuralNetwork(nn.Module):
    def forward(self, X):
        o = self.activation(self.z3)  # final activation function
        return o

    # backward propagation
    def backward(self, X, y, o, rate):
        self.out_error = y - o  # error in output
        self.out_delta = self.out_error * self.activation_derivative(o)  # derivative of activation to error

        # error and derivative of activation to error of next layer in backward propagation
        self.z2_error = torch.matmul(self.out_delta, torch.t(self.W2))
        self.z2_delta = self.z2_error * self.activation_derivative(self.z2)

        # update weights from delta of error and learning rate
        self.W1 += torch.matmul(torch.t(X), self.z2_delta) * rate
        self.W2 += torch.matmul(torch.t(self.z2), self.out_delta) * rate

    # training function with learning rate parameter
    def train(self, X, y, rate):
        # forward + backward pass for training
        o = self.forward(X)
        self.backward(X, y, o, rate)

    # save weights of model
    @staticmethod
    def save_weights(model, path):
        # use the PyTorch internal storage functions
        torch.save(model, path)

    # load weights of model
    @staticmethod
    def load_weights(path):
```

```python
    # save weights of model
    @staticmethod
    def save_weights(model, path):
        # use the PyTorch internal storage functions
        torch.save(model, path)

    # load weights of model
    @staticmethod
    def load_weights(path):
        # reload model with all the weights
        torch.load(path)

    # predict function
    def predict(self, x_predict):
        print("Predict data based on trained weights: ")
        print("Input: \n" + str(x_predict))
        print("Output: \n" + str(self.forward(x_predict)))
```

est_ffnn.py > ...

```python
# import PyTorch
import torch
# import Feed Forward Neural Network class from nn_simple module
from nn_simple import ffnn
from sklearn.datasets import load_wine

# sample input and output value for training
X = torch.tensor(([2, 9, 0], [1, 5, 1], [3, 6, 2]), dtype=torch.float)  # 3 X 3 tensor
y = torch.tensor(([90], [100], [88]), dtype=torch.float)  # 3 X 1 tensor

# Load dữ liệu Wine
#wine = load_wine()
#X = torch.tensor(wine.data, dtype=torch.float32)
#y = torch.tensor(wine.target, dtype=torch.long)


# scale units by max value
X_max, _ = torch.max(X, 0)
X = torch.div(X, X_max)
y = y / 100  # for max test score is 100

# sample input x for predicting
x_predict = torch.tensor(([3, 8, 4]), dtype=torch.float)  # 1 X 3 tensor

# scale input x by max value
x_predict_max, _ = torch.max(x_predict, 0)
x_predict = torch.div(x_predict, x_predict_max)

# create new object of implemented class
NN = ffnn.FFNeuralNetwork()
```

```python
# create new object of implemented class
NN = ffnn.FFNeuralNetwork()

# trains the NN 1,000 times
for i in range(1000):
    # print mean sum squared loss
    print("#" + str(i) + " Loss: " + str(torch.mean((y - NN(X)) ** 2).detach().item()))
    # training with learning rate = 0.1
    NN.train(X, y, 0.1)
# save weights
NN.save_weights(NN, "NN")

# load saved weights
NN.load_weights("NN")
# predict x input
NN.predict(x_predict)
```

III. Summarization of the usage

Usage instructions:

1. **Prepare the input data**:
    - Prepare the input X (features) and the output y (labels) and normalize them.

```
X_max, _ = torch.max(X, 0)
X = torch.div(X, X_max)
y = y / 100  # If the target output is a percentage (0-100)
```

2. **Create the FFNN**:
   - Initialize the FFNeuralNetwork object.

```
NN = ffnn.FFNeuralNetwork()
```

3. **Train the model**:
   - Train the network for 1000 epochs with a learning rate of 0.1.

```
for i in range(1000):
    print(f"Epoch #{i} Loss: {torch.mean((y - NN(X)) ** 2).detach().item()}")
    NN.train(X, y, 0.1)
```

4. **Save and load weights**:

```
NN.save_weights(NN, "NN_weights")  # Save the model weights
NN.load_weights("NN_weights")  # Load the model weights
```

5. **Make predictions**:
   - Prepare and normalize the new input x_predict before making a prediction.

```
x_predict_max, _ = torch.max(x_predict, 0)
x_predict = torch.div(x_predict, x_predict_max)
NN.predict(x_predict)
```

Algorithm explanation

1. **Forward Propagation**:

- **Input** X is multiplied by weights W_1 and passed through the sigmoid activation function.

$$z_1 = X \cdot W_1$$

$$z_2 = \sigma(z_1)$$

- The result z_2 is multiplied by the second set of weights W_2 and passed through the sigmoid activation function again.

$$z_3 = z_2 \cdot W_2$$

$$o = \sigma(z_3)$$

- The final output o is returned.

2. **Backward Propagation**:

- Compute the error of the output:

$$\text{out\_error} = y - o$$

- Compute gradients of the error with respect to the activations and propagate it backward through the layers.
- Update the weights W_1 and W_2 using gradient descent.

3. **Training**:

   - For each epoch, forward propagation and backward propagation are executed to update the weights.
   - This process continues for a fixed number of epochs or until a stopping criterion is met.

IV. Comments and evaluation

Results on sample data:

- **Input data**:

$$X = \begin{bmatrix} 2 & 9 & 0 \\ 1 & 5 & 1 \\ 3 & 6 & 2 \end{bmatrix}$$

$$y = \begin{bmatrix} 90 \\ 100 \\ 88 \end{bmatrix}$$

- **Prediction data**:

$$x_{\text{predict}} = [3, 8, 4]$$

- **Prediction result**:
  After training, the network predicts a score close to the expected value.

```
Predict data based on trained weights:
Input:
tensor([0.3750, 1.0000, 0.5000])
Output:
tensor([0.9256])
```

Evaluation:

1. **Strengths**:
   - **Simple and modular design**: The FFNN class is well-structured, making it easy to add more layers.

- **Customizable**: The number of layers, hidden units, and learning rate can be easily modified.
- **Weight saving and loading**: This allows the model to be reused without retraining.

2. **Weaknesses**:
   - **No bias term**: The weight update formula does not include bias, which may reduce model accuracy.
   - **Overfitting risk**: The model may overfit if trained too long on small datasets.
   - **Performance on larger datasets**: The model has only been tested on a small dataset and may struggle with larger, more complex datasets.
3. **Evaluation on the Wine dataset**:
   - The model could be modified to support multi-class classification.
   - Activation functions other than sigmoid (like ReLU) should be tested for better convergence.

Conclusion

- **Summary**:
  Successfully implemented a Feed Forward Neural Network (FFNN) using PyTorch. The model was trained on a small dataset and tested on new data, showing good prediction performance.
- **Improvements**:
  - Add support for bias in the model.
  - Use dropout and regularization to reduce overfitting.
  - Test on larger datasets (like the Wine dataset) to analyze model performance on real-world problems.

V. EXPERIMENT AND RESULTS:

ON Default result:

```
Predict data based on trained weights:
Input:
tensor([0.3750, 1.0000, 0.5000])
Output:
tensor([0.9212])
```

The model successfully predicts an output of approximately 0.9212 for the input tensor `[0.3750, 1.0000, 0.5000]`. The output seems reasonable, given that the model was trained with a target output in the range [0, 1]. However, further evaluation, such as comparing the predicted output with expected results, would be useful for validating its accuracy.

5.1 Change epochs number (learning rate = 0.1)

- **Epochs = 50**

```
PROBLEMS   DEBUG CONSOLE   OUTPUT   TERMINAL   PORTS   COMMENTS
c:\Users\hp\OneDrive\Documents\thanh thanh\nam4\hoc ki 1\xu li anh\lab\lab 3\source\pytorch_f
Predict data based on trained weights:
Input:
tensor([0.3750, 1.0000, 0.5000])
Output:
tensor([0.6659])

[Done] exited with code=0 in 3.474 seconds
```

- **Epochs = 100**

```
  torch.load(path)
Predict data based on trained weights:
Input:
tensor([0.3750, 1.0000, 0.5000])
Output:
tensor([0.8653])

[Done] exited with code=0 in 2.256 seconds
```

- **Epochs = 500**

```
Predict data based on trained weights:
Input:
tensor([0.3750, 1.0000, 0.5000])
Output:
tensor([0.9487])
```

- **Epochs = 1000:**

```
Predict data based on trained weights:
Input:
tensor([0.3750, 1.0000, 0.5000])
Output:
tensor([0.8903])
```

- **Epochs = 2000:**

```
Predict data based on trained weights:
Input:
tensor([0.3750, 1.0000, 0.5000])
Output:
tensor([0.9323])
```

Epoch 50:

- **Output**: 0.6659
    - At epoch 50, the model has made some progress in learning but is still far from an accurate prediction. This suggests that the model is still in the early stages of training and is yet to understand the full relationship between the input and output.

Epoch 100:

- **Output**: 0.8653
    - By epoch 100, the model's prediction improves significantly, reflecting that the model has started to learn more effectively. The improvement in the prediction shows that the neural network is learning the patterns in the data and adjusting its weights accordingly.

Epoch 500:

- **Output**: 0.9487
    - At epoch 500, the model has achieved a higher degree of accuracy. The prediction is quite close to the actual value, indicating that the model is converging toward an optimal solution. The improvement from earlier epochs suggests that the training process is proceeding well.

Epoch 1000:

- **Output**: 0.8903
    - At epoch 1000, we observe a slight dip in performance. The output value decreases compared to epoch 500, which may indicate that the model has started to overfit or that the learning rate needs adjustment. The model might be bouncing between different solutions as it tries to converge further.

Epoch 2000:

- **Output**: 0.9323
    - By epoch 2000, the model returns to a higher accuracy, with the output approaching the value from epoch 500. This shows that the model has continued to refine its learning after the slight drop at epoch 1000. The prediction stabilizes, suggesting that the model may be close to its optimal state.

General Observations:

- The model improves significantly from epoch 50 to epoch 500, showing strong learning.
- However, after epoch 500, the model's performance fluctuates slightly, possibly due to overfitting or the training process reaching a local minimum.
- By epoch 2000, the model stabilizes at a reasonable prediction value (0.9323), but it's important to monitor for potential overfitting or the need for adjustments to the training parameters (e.g., learning rate, regularization) to improve convergence.

In summary, the model's accuracy improves steadily through the first 500 epochs and then fluctuates, which could be a sign of the learning rate or training process requiring fine-tuning to avoid overfitting and further optimize the performance.

5.2 Change learning rate (epochs = 100)

- **rate = 0.1**

```
Predict data based on trained weights:
Input:
tensor([0.3750, 1.0000, 0.5000])
Output:
tensor([0.8758])
```

- **rate = 0.01**

```
Input:
tensor([0.3750, 1.0000, 0.5000])
Output:
tensor([0.6420])
```

- **rate = 0.001**

```
Predict data based on trained weights:
Input:
tensor([0.3750, 1.0000, 0.5000])
Output:
tensor([0.6978])
```

- **rate = 0.5**

```
Input:
tensor([0.3750, 1.0000, 0.5000])
Output:
tensor([0.9081])
```

- **Rate = 0.1**: Output 0.8758. This is a good prediction, indicating effective learning with a balanced rate.
- **Rate = 0.01**: Output 0.6420. The model learns slower, leading to a less accurate prediction due to a small learning rate.
- **Rate = 0.001**: Output 0.6978. Very slow learning, resulting in moderate prediction accuracy, but may need more epochs to improve.
- **Rate = 0.5**: Output 0.9081. Faster learning with a high rate, resulting in a strong prediction, but there's a risk of instability or overshooting.

Conclusion: A learning rate of 0.1 offers a good balance between speed and accuracy. Lower rates slow learning, and higher rates may cause instability.