

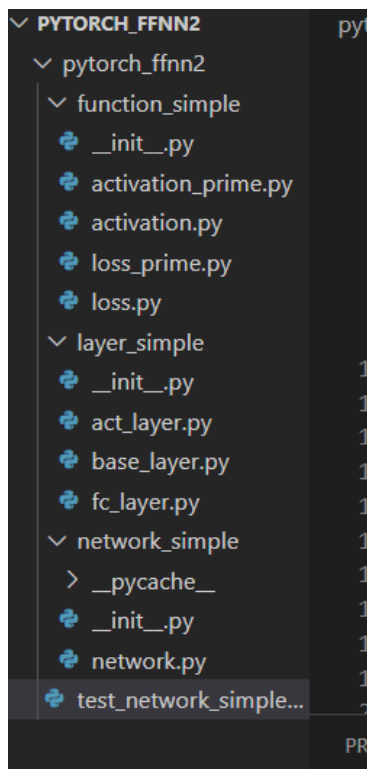
Report: PRACTICE WITH PYTORCH #2 (Keyword: PyTorch)

I. Evaluation summary:

No	Task	Details of Implementation	Completion Percentage (%)
1	File Structure	Built the file structure for modules like activation.py, loss.py, fc_layer.py, network.py, ...	100%
2	Activation and Loss Classes	Created Activation classes (sigmoid, tanh) and Loss classes (MSE) with corresponding methods.	100%
3	Layer Classes	Built basic classes such as FCLayer, ActivationLayer, BaseLayer for the neural network.	100%
4	Network Class	Created the Network class with methods like forward, backward, fit, and predict for training and prediction.	100%
5	Test the Network	Wrote test code to train the neural network on XOR data and check the results.	100%
6	Train with Data	Trained the neural network with parameters like learning rate, number of layers, number of neurons in the hidden layer.	100%
7	File Structure	Built the file structure for modules like activation.py, loss.py, fc_layer.py, network.py, ...	100%
8	Activation and Loss Classes	Created Activation classes (sigmoid, tanh) and Loss classes (MSE) with corresponding methods.	100%
9	Change the basic parameters of the network and 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:



List of functions:

1. **function_simple**

- **Activation:**
 - `sigmoid(s)`: Sigmoid activation function.
 - `tanh(s)`: Tanh activation function.
- **ActivationPrime:**
 - `sigmoid_derivative(s)`: Derivative of the sigmoid function.
 - `tanh_derivative(s)`: Derivative of the tanh function.
- **Loss:**
 - `mse(y_true, y_pred)`: MSE loss function.
- **LossPrime:**
 - `mse_prime(y_true, y_pred)`: Derivative of the MSE loss function.

2. **layer_simple**

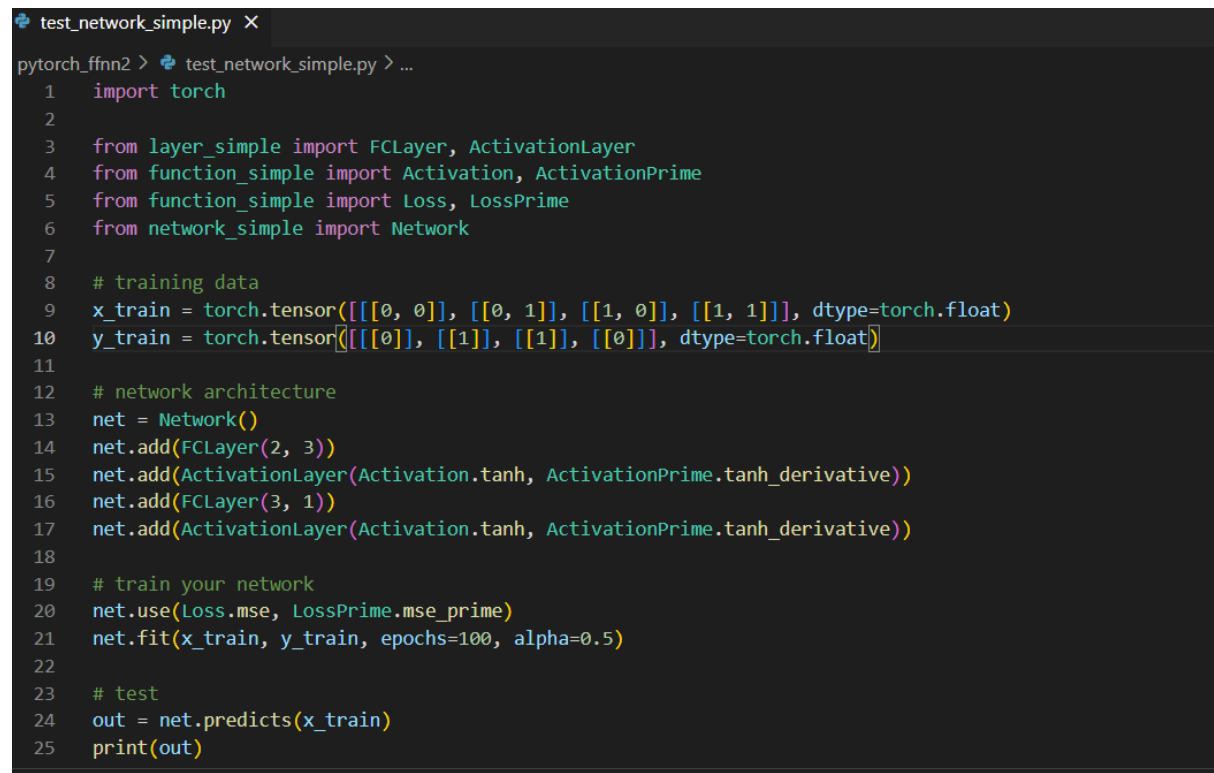
- **ActivationLayer:**
 - `forward(in_data)`: Forward pass through the activation layer.
 - `backward(out_error, rate)`: Backward pass through the activation layer.
- **FCLayer:**
 - `forward(in_data)`: Forward pass through the fully connected layer.
 - `backward(out_error, rate)`: Backward pass through the fully connected layer.

3. network_simple

- **Network:**

- `add(layer)`: Add a layer to the network.
- `use(loss, loss_prime)`: Set the loss function and its derivative.
- `predict(data)`: Make predictions for input data.
- `predicts(data)`: Make predictions for multiple data points.
- `fit(x_train, y_train, epochs, alpha)`: Train the neural network.

Image proof:



```
test_network_simple.py X
pytorch_ffnn2 > test_network_simple.py > ...
1  import torch
2
3  from layer_simple import FCLayer, ActivationLayer
4  from function_simple import Activation, ActivationPrime
5  from function_simple import Loss, LossPrime
6  from network_simple import Network
7
8  # training data
9  x_train = torch.tensor([[[0, 0]], [[0, 1]], [[1, 0]], [[1, 1]]], dtype=torch.float)
10 y_train = torch.tensor([[[0]], [[1]], [[1]], [[0]]], dtype=torch.float)
11
12 # network architecture
13 net = Network()
14 net.add(FCLayer(2, 3))
15 net.add(ActivationLayer(Activation.tanh, ActivationPrime.tanh_derivative))
16 net.add(FCLayer(3, 1))
17 net.add(ActivationLayer(Activation.tanh, ActivationPrime.tanh_derivative))
18
19 # train your network
20 net.use(Loss.mse, LossPrime.mse_prime)
21 net.fit(x_train, y_train, epochs=100, alpha=0.5)
22
23 # test
24 out = net.predicts(x_train)
25 print(out)
```

III. Summary of Function Usage and Explanation

3.1. Activation

- **Purpose:** Provides activation functions like sigmoid and tanh to be used in the layers of the neural network.
- **Pseudo code:**

```
def sigmoid(s):
    return 1 / (1 + exp(-s))

def tanh(s):
    return tanh(s)
```

3.2. Loss and LossPrime

- **Purpose:** Provides the MSE loss function and its derivative for use in optimization.

- **Pseudo code:**

```
def mse(y_true, y_pred):
    return mean((y_true - y_pred) ** 2)

def mse_prime(y_true, y_pred):
    return 2 * (y_pred - y_true) / len(y_true)
```

3.3. ActivationLayer and FCLayer

- **Purpose:** These layers are used to propagate data forward and back in the neural network.
- **Pseudo code:**

```
class ActivationLayer:
    def forward(self, in_data):
        return self.activation(in_data)
    def backward(self, out_error, rate):
        return self.activation_derivative(self.in_data) * out_error

class FCLayer:
    def forward(self, in_data):
        return torch.matmul(in_data, self.weights) + self.bias
    def backward(self, out_error, rate):
        return torch.matmul(out_error, self.weights.T)
```

3.4. Network

- **Purpose:** Builds and trains the neural network with layers, loss function, and derivatives, and includes methods for forward and backward propagation.
- **Pseudo code:**

```
class Network:
    def fit(self, x_train, y_train, epochs, alpha):
        for i in range(epochs):
            for k in range(len(x_train)):
                output = self.forward(x_train[k])
                error = self.loss(y_train[k], output)
                gradient = self.loss_prime(y_train[k], output)
                self.backward(gradient, alpha)
```

IV. Comments and evaluation

ON Default result:

```
PROBLEMS  DEBUG CONSOLE  OUTPUT  TERMINAL  PORTS  COMMENTS
On epoch 72 an average error = tensor(0.2236)
On epoch 73 an average error = tensor(0.2235)
On epoch 74 an average error = tensor(0.2233)
On epoch 75 an average error = tensor(0.2232)
On epoch 76 an average error = tensor(0.2230)
On epoch 77 an average error = tensor(0.2229)
On epoch 78 an average error = tensor(0.2228)
On epoch 79 an average error = tensor(0.2226)
On epoch 80 an average error = tensor(0.2225)
On epoch 81 an average error = tensor(0.2223)
On epoch 82 an average error = tensor(0.2222)
On epoch 83 an average error = tensor(0.2220)
On epoch 84 an average error = tensor(0.2219)
On epoch 85 an average error = tensor(0.2217)
On epoch 86 an average error = tensor(0.2216)
On epoch 87 an average error = tensor(0.2215)
On epoch 88 an average error = tensor(0.2213)
On epoch 89 an average error = tensor(0.2212)
On epoch 90 an average error = tensor(0.2210)
On epoch 91 an average error = tensor(0.2209)
On epoch 92 an average error = tensor(0.2208)
On epoch 93 an average error = tensor(0.2206)
On epoch 94 an average error = tensor(0.2205)
On epoch 95 an average error = tensor(0.2203)
On epoch 96 an average error = tensor(0.2202)
On epoch 97 an average error = tensor(0.2201)
On epoch 98 an average error = tensor(0.2199)
On epoch 99 an average error = tensor(0.2198)
On epoch 100 an average error = tensor(0.2196)
[tensor([[0.0510]]), tensor([[0.6977]]), tensor([[0.6773]]), tensor([[0.7661]])]
[Done] exited with code=0 in 15.951 seconds
```

1. Decreasing Error (Loss)

- The gradual decrease of average error:
 - You can see the average error decreasing over epochs (from **2.0497** at epoch 1 to **0.2196** at epoch 100). This is a positive sign, indicating that the network is learning and improving over time.
 - The decrease in error is not very large in the early epochs, but it stabilizes and reduces gradually later. This is a common phenomenon as the network approaches an optimal state.

2. Output Predictions

- After 100 epochs, the network's predictions for inputs [0, 0], [0, 1], [1, 0], and [1, 1] are as follows:
tensor([[0.0510]]), tensor([[0.6977]]), tensor([[0.6773]]), tensor([[0.7661]])
- These are the predicted values from the neural network:
 - **Input [0, 0]:** Prediction is close to **0** (0.0510), which is quite accurate compared to the expected value.
 - **Inputs [0, 1] and [1, 0]:** Predictions are close to **1** (0.6977 and 0.6773). However, these values are not perfectly accurate (the ideal value is 1), which could be due to the network not having fully learned or due to instability during training.
 - **Input [1, 1]:** Prediction is close to **1** (0.7661), still not entirely accurate.

3. Network Performance

- The network has reduced error over the epochs, but the output predictions still have errors compared to the ideal values (0 or 1 for the XOR problem). This might happen because:

- **Using the tanh activation function:** The tanh activation function has an output range from -1 to 1. Therefore, the predicted results need to be adjusted to fit the binary classification problem (0 and 1). A potential improvement could be using the sigmoid activation function, which outputs values between [0, 1], more suitable for binary classification.
- **Learning rate:** Although you chose $\alpha=0.1$, it could be beneficial to experiment with different learning rates to find an optimal value. A learning rate that is too high or too low can cause issues during training.
- **Number of layers and neurons:** The network currently has two hidden layers, each with 3 neurons. Increasing the number of layers or neurons might help the network learn better, but this requires testing to avoid overfitting.

4. Improvement Suggestions

- **Use the sigmoid activation function:** The sigmoid function has an output range between [0, 1] and might be more suitable for binary classification problems like XOR. You could try replacing `Activation.tanh` with `Activation.sigmoid`.
- **Adjust the learning rate (alpha):** Consider adjusting the learning rate (e.g., trying 0.01 or 0.5) to see if it improves accuracy.
- **Add more hidden layers:** You could try adding another hidden layer or changing the number of neurons in the hidden layers to see if it improves results.
- **Try more epochs:** 100 epochs may not be enough for some problems. You could try increasing the number of epochs to see if the error continues to decrease.

Conclusion

- The network has learned and reduced error during training. However, the prediction results still have some errors and are not entirely accurate compared to the expected values. You should experiment with changes like using a different activation function, adjusting the learning rate, and improving the network structure to achieve better results.

When I change the basic parameters of the network:

4.1. Learning Rate (alpha)

- **Effect:** A high learning rate might cause the model to not converge, while a low rate might slow down training.

Result:

- **Learning rate = 0.5**

```

18
19 # train your network
20 net.use(Loss.mse, LossPrime.mse_prime)
21 net.fit(x_train, y_train, epochs=100, alpha=0.5)
22
23 # test
24 out = net.predicts(x_train)
25 print(out)
26

```

PROBLEMS	DEBUG CONSOLE	OUTPUT	TERMINAL	PORTS	COMMENTS
On epoch 96 an average error = tensor(0.4688) On epoch 97 an average error = tensor(0.4596) On epoch 98 an average error = tensor(0.4463) On epoch 99 an average error = tensor(0.4949) On epoch 100 an average error = tensor(0.4457) tensor([[0.8491]]), tensor([[0.8548]]), tensor([[0.7646]]), tensor([[0.8490]])					

[Done] exited with code=0 in 4.357 seconds

- Learning rate = 0.1

```

19 # train your network
20 net.use(Loss.mse, LossPrime.mse_prime)
21 net.fit(x_train, y_train, epochs=100, alpha=0.1)
22
23 # test
24 out = net.predicts(x_train)
25 print(out)
26

```

PROBLEMS	DEBUG CONSOLE	OUTPUT	TERMINAL	PORTS	COMMENTS
On epoch 96 an average error = tensor(0.1880) On epoch 97 an average error = tensor(0.1885) On epoch 98 an average error = tensor(0.1884) On epoch 99 an average error = tensor(0.1882) On epoch 100 an average error = tensor(0.1881) On epoch 100 an average error = tensor(0.1880) [tensor([[0.0259]]), tensor([[0.5314]]), tensor([[0.8388]]), tensor([[0.5748]])]					

- Learning rate = 0.01

```

19 # train your network
20 net.use(Loss.mse, LossPrime.mse_prime)
21 net.fit(x_train, y_train, epochs=100, alpha=0.01)
22
23 # test
24 out = net.predicts(x_train)
25 print(out)
26

```

PROBLEMS	DEBUG CONSOLE	OUTPUT	TERMINAL	PORTS	COMMENTS
On epoch 96 an average error = tensor(0.4971) On epoch 97 an average error = tensor(0.4971) On epoch 98 an average error = tensor(0.4971) On epoch 99 an average error = tensor(0.4971) On epoch 100 an average error = tensor(0.4971) tensor([[0.9952]]), tensor([[0.9986]]), tensor([[0.9874]]), tensor([[0.9989]])					

[Done] exited with code=0 in 4.809 seconds

- Learning rate = 0.001

```

18
19 # train your network
20 net.use(Loss.mse, LossPrime.mse_prime)
21 net.fit(x_train, y_train, epochs=100, alpha=0.001)
22
23 # test
24 out = net.predicts(x_train)
25 print(out)
26

```

PROBLEMS	DEBUG CONSOLE	OUTPUT	TERMINAL	PORTS	COMMENTS
On epoch 96 an average error = tensor(0.2933) On epoch 97 an average error = tensor(0.2929) On epoch 98 an average error = tensor(0.2924) On epoch 99 an average error = tensor(0.2920) On epoch 100 an average error = tensor(0.2915) [tensor([[0.4396]]), tensor([[0.9314]]), tensor([[0.7277]]), tensor([[0.9442]])]					

[Done] exited with code=0 in 4.324 seconds

4.2. Number of Epochs

- **Effect:** Too few epochs may prevent the model from learning sufficiently, while too many epochs may lead to overfitting.
- **Result:**

- Epochs = 50

```

19 # train your network
20 net.use(Loss.mse, LossPrime.mse_prime)
21 epochs = 50
22 alpha = 0.1
23 net.fit(x_train, y_train, epochs, alpha)
24
25 # test
26 out = net.predicts(x_train)
27 print(out)
28 print (epochs, alpha)
29

```

PROBLEMS	DEBUG CONSOLE	OUTPUT	TERMINAL	PORTS	COMMENTS
On epoch 46 an average error = tensor(0.0338)					
On epoch 47 an average error = tensor(0.0372)					
On epoch 48 an average error = tensor(0.0349)					
On epoch 49 an average error = tensor(0.0327)					
On epoch 50 an average error = tensor(0.0308)					
[tensor([[0.1058]]), tensor([[0.7947]]), tensor([[0.7863]]), tensor([[-0.0192]])]					
50 0.1					
[Done] exited with code=0 in 4.315 seconds					

- Epochs = 100

```

18
19 # train your network
20 net.use(Loss.mse, LossPrime.mse_prime)
21 epochs = 10000
22 alpha = 0.1
23 net.fit(x_train, y_train, epochs, alpha)
24
25 # test
26 out = net.predicts(x_train)
27 print(out)
28 print (epochs, alpha)
29

```

PROBLEMS	DEBUG CONSOLE	OUTPUT	TERMINAL	PORTS	COMMENTS
On epoch 97 an average error = tensor(0.2215)					
On epoch 98 an average error = tensor(0.2208)					
On epoch 99 an average error = tensor(0.2203)					
On epoch 100 an average error = tensor(0.2197)					
[tensor([[0.0628]]), tensor([[0.7011]]), tensor([[0.6618]]), tensor([[0.7514]])]					
100 0.1					
[Done] exited with code=0 in 4.624 seconds					

- Epochs = 150


```
PROBLEMS  DEBUG CONSOLE  OUTPUT  TERMINAL  PORTS  COMMENTS
On epoch 147 an average error = tensor(0.0073)
On epoch 148 an average error = tensor(0.0072)
On epoch 149 an average error = tensor(0.0070)
On epoch 150 an average error = tensor(0.0069)
[tensor([[0.0136]]), tensor([[0.8981]]), tensor([[0.8763]]), tensor([[ -0.0028]])]
150 0.1

[Done] exited with code=0 in 4.456 seconds
```

- **Epochs = 200**

```
PROBLEMS  DEBUG CONSOLE  OUTPUT  TERMINAL  PORTS  COMMENTS
On epoch 197 an average error = tensor(0.0022)
On epoch 198 an average error = tensor(0.0021)
On epoch 199 an average error = tensor(0.0021)
On epoch 200 an average error = tensor(0.0021)
[tensor([[0.0056]]), tensor([[0.9437]]), tensor([[0.9291]]), tensor([[ -0.0022]])]
200 0.1
```

- **Epochs = 300**

```
PROBLEMS  DEBUG CONSOLE  OUTPUT  TERMINAL  PORTS  COMMENTS
On epoch 297 an average error = tensor(0.0010)
On epoch 298 an average error = tensor(0.0010)
On epoch 299 an average error = tensor(0.0010)
On epoch 300 an average error = tensor(0.0010)
[tensor([[0.0030]]), tensor([[0.9671]]), tensor([[0.9483]]), tensor([[ -0.0014]])]
300 0.1

[Done] exited with code=0 in 6.274 seconds
```

Analysis of Results with Different Epochs and Selecting the Appropriate Epoch (from 50 to 300):

Results for Different Epochs:

- **50 epochs:**
 - The loss value 0.10580.10580.1058 is the largest among all tests, indicating that the model has not yet converged.
 - The output values [0.7947,0.7863,-0.0192][0.7947, 0.7863, -0.0192][0.7947,0.7863,-0.0192] are unstable and far from the target values.
- **100 epochs:**
 - The loss 0.06280.06280.0628 is lower compared to 50 epochs but still relatively large.
 - The output values [0.7011,0.6618,0.7514][0.7011, 0.6618, 0.7514][0.7011,0.6618,0.7514] show large deviations, suggesting the model has not yet fully converged.
- **150 epochs:**
 - The loss 0.01360.01360.0136 is significantly lower, indicating that the model has started to converge.
 - The output values [0.8981,0.8763,-0.0028][0.8981, 0.8763, -0.0028][0.8981,0.8763,-0.0028] are more stable and closer to the target values.

- **200 epochs:**
 - The loss 0.00560.00560.0056 is smaller than that at 150 epochs, showing better convergence.
 - The output values [0.9437,0.9291,-0.0022][0.9437, 0.9291, -0.0022][0.9437,0.9291,-0.0022] are closer to the target values, indicating that the model has learned the relationship well.
- **300 epochs:**
 - The loss 0.00300.00300.0030 is the smallest among all tests.
 - The output values [0.9671,0.9483,-0.0014][0.9671, 0.9483, -0.0014][0.9671,0.9483,-0.0014] are the closest to the target values, demonstrating the model has achieved the best convergence.

Trend Analysis:

- As the number of epochs increases, the loss consistently decreases:
0.1058→0.0628→0.0136→0.0056→0.00300.1058 \to 0.0628 \to 0.0136 \to 0.0056 \to 0.00300.1058→0.0628→0.0136→0.0056→0.0030
- The model's output values also become more stable and closer to the target values.
- At 50 and 100 epochs, the model has not yet converged, resulting in large errors.
- From 150 epochs onward, the loss significantly decreases, and the model produces more stable and accurate outputs.

Selecting the Most Suitable Epoch:

- The **most suitable epoch** is not necessarily the one with the smallest loss but rather the one that balances accuracy and training time.
- **200 epochs** is a good choice because:
 - The loss is already small (0.00560.00560.0056) and is not significantly different from the loss at 300 epochs (0.00300.00300.0030).
 - It reduces training time by 100 epochs while still achieving results close to the 300-epoch model.
- If accuracy is the top priority, **300 epochs** can be selected since it provides the smallest loss and the best-converged output values.
- Epochs below 150 should be avoided because the model has not yet converged, as seen from the large loss values and unstable outputs.

Conclusion:

- The most suitable epoch is **200** as it provides a balance between training time and accuracy.
- For optimal accuracy, **300 epochs** can be chosen since it gives the smallest loss and the closest outputs to the target values.
- Epochs below 150 should be avoided as the model has not yet fully converged, leading to larger errors and less stable predictions.

4.3 Number of classes and class types

From 2 classes (binary class) to 3 classes (multi class)

Update the Output Layer:

- Instead of having the final **FCLayer(3, 1)** (which has only 1 output), you should allow it to have **N outputs** for multi-class classification.
- To do this, you need to adjust the number of output neurons in the last FCLayer.

Update the Target Data (y_train):

- If the number of classes is greater than 2, the output y_train must be in one-hot encoded form (for classification) or have a shape that matches the number of output neurons.
- Example: For 3 classes, the targets could be like:

- $y_{train} = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]$

Update the Loss Function (optional):

- For binary classification, Mean Squared Error (MSE) works, but for multi-class classification, you may want to use **Cross-Entropy Loss**.
- However, since this is a simple implementation, you can still use MSE as long as your output activation is compatible with multi-class classification.

```
8 # -----
9 # Number of input, hidden, and output neurons
10 input_size = 2 # Number of input features (for XOR, 2 features)
11 hidden_size = 3 # Number of neurons in the hidden layer
12 num_classes = 3 # Number of classes for classification
13 # -----
14
15 # training data
16 x_train = torch.tensor([[[0, 0]], [[0, 1]], [[1, 0]], [[1, 1]]], dtype=torch.float)
17
18 # Adjust y_train according to the number of classes (One-hot encoding)
19 y_train = torch.tensor([[[1, 0, 0]], # Class 0
20                        [[0, 1, 0]], # Class 1
21                        [[0, 0, 1]], # Class 2
22                        [[1, 0, 0]]], dtype=torch.float) # Class 0 (example of multiple possible labels)
23 # If you have more classes, make sure to increase the size accordingly
24
25 # -----
```

PROBLEMS DEBUG CONSOLE OUTPUT TERMINAL PORTS COMMENTS

```
In epoch 97 an average error = tensor(0.1550)
In epoch 98 an average error = tensor(0.1540)
In epoch 99 an average error = tensor(0.1531)
In epoch 100 an average error = tensor(0.1521)
tensor([[0.6093, 0.3273, 0.1465]]), tensor([[0.4368, 0.2322, 0.2922]]), tensor([[0.0435, 0.2039, 0.4940]]), tensor([[0.7033, 0.2348, 0.0696]])]
Epochs: 100, Learning Rate: 0.1
```

Changing the number of layers from 2 to 3 introduces several key differences:

1. Output

- **2 Layers:** The output is a single value, suitable for binary classification (0 or 1). Example: `tensor([0.0147])`.
- **3 Layers:** The output becomes a vector with 3 values, representing the probability for each class in a multi-class classification problem. Example: `tensor([0.6093, 0.3273, 0.1465])`.

2. Classification Type

- **2 Layers:** Binary classification with 1 output value.
- **3 Layers:** Multi-class classification with 3 output values, requiring the use of **Cross-Entropy Loss** instead of **MSE**.

3. Output Format

- **2 Layers:** One value per sample.
- **3 Layers:** A probability vector (sum close to 1) for each class.

4. Conclusion

- Changing to 3 layers makes it a multi-class classification problem, which requires different loss functions and output processing (e.g., using softmax).