pytorch-basic-neural-networks

December 17, 2025

1 Lab 4: Pytorch - basic neural networks

1.0.1 Full Name: Nguyen Anh Thư

1.0.2 Student's ID: 23127266

2 PyTorch Feed Forward Neural Network

2.1 STEP 1 — Environment Setup

Import required libraries: `torch`, `numpy`

[34]:
```
# Uncomment if required to install dependencies for this lab


#!pip -q install torch numpy
```

[35]:
```
import torch
import numpy as np
import torch.nn as nn

print(f"PyTorch version: {torch.__version__}")
print(f"NumPy version: {np.__version__}")
```

```
PyTorch version: 2.7.1+cpu
NumPy version: 2.2.3
```

2.2 STEP 2 — Working with Tensors

Perform conversions between NumPy arrays and PyTorch Tensors, demonstrating shared memory mechanism.

[36]:
```
# Create NumPy array
np_array = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.float32)
print("NumPy array:")
print(np_array)

# Convert NumPy → PyTorch Tensor
tensor_from_np = torch.from_numpy(np_array)
print("\nPyTorch Tensor from NumPy:")
```

```
print(tensor_from_np)

# Demonstrate shared memory: changing NumPy array will affect Tensor
np_array[0, 0] = 999
print("\nAfter modifying NumPy array:")
print("NumPy array:", np_array)
print("PyTorch Tensor (shared memory):", tensor_from_np)

# Convert PyTorch Tensor → NumPy
np_array_from_tensor = tensor_from_np.numpy()
print("\nNumPy array from Tensor:")
print(np_array_from_tensor)

# Perform addition
result = np_array + np_array_from_tensor
print("\nAddition result:")
print(result)
```

```
NumPy array:
[[1. 2. 3.]
 [4. 5. 6.]]

PyTorch Tensor from NumPy:
tensor([[1., 2., 3.],
        [4., 5., 6.]])

After modifying NumPy array:
NumPy array: [[999.   2.   3.]
 [  4.   5.   6.]]
PyTorch Tensor (shared memory): tensor([[999.,   2.,   3.],
        [  4.,   5.,   6.]])

NumPy array from Tensor:
[[999.   2.   3.]
 [  4.   5.   6.]]

Addition result:
[[1998.    4.    6.]
 [   8.   10.   12.]]
```

2.3 STEP 3 — Define Activation Functions

Class `ActivationFunction` contains activation functions and their derivatives.

[37]:
```
class ActivationFunction:
    """Utility class for activation functions and their derivatives.

    This class provides static methods for sigmoid activation function
```

```
    and its derivative, using PyTorch operations only.
    """

    @staticmethod
    def sigmoid(s):
        """Compute sigmoid activation function.

        Args:
            s: Input tensor

        Returns:
            Tensor: Sigmoid activation of input, output range [0, 1]
        """
        return 1 / (1 + torch.exp(-s))

    @staticmethod
    def sigmoid_derivative(s):
        """Compute derivative of sigmoid function.

        Args:
            s: Input tensor (typically sigmoid output)

        Returns:
            Tensor: Derivative of sigmoid at input point
        """
        return s * (1 - s)
```

[38]:
```
# Test activation functions
test_input = torch.tensor([0.0, 1.0, -1.0], dtype=torch.float32)
print("Input:", test_input)
print("Sigmoid:", ActivationFunction.sigmoid(test_input))
print("Sigmoid derivative:", ActivationFunction.
 ↪sigmoid_derivative(ActivationFunction.sigmoid(test_input)))
```

```
Input: tensor([ 0.,  1., -1.])
Sigmoid: tensor([0.5000, 0.7311, 0.2689])
Sigmoid derivative: tensor([0.2500, 0.1966, 0.1966])
```

2.4 STEP 4-8 — Define Feed Forward Neural Network

Class `FFNeuralNetwork` inherits from `nn.Module`, containing: - Initialization with weights and intermediate variables - Forward propagation - Backward propagation
- Training function - Save/Load weights

[39]:
```
class FFNeuralNetwork(nn.Module):
    """Feed Forward Neural Network implementation from scratch.

    A simple 3-layer neural network (input -> hidden -> output) with
```

```
    sigmoid activation function and manual backpropagation implementation.

    Attributes:
        inputSize: Number of input features (default: 3)
        hiddenSize: Number of neurons in hidden layer (default: 4)
        outputSize: Number of output neurons (default: 1)
        W1: Weight matrix from input to hidden layer (inputSize x hiddenSize)
        W2: Weight matrix from hidden to output layer (hiddenSize x outputSize)
    """

    def __init__(self, input_size=3, hidden_size=4, output_size=1):
        """Initialize the neural network.

        Args:
            input_size: Number of input features
            hidden_size: Number of neurons in hidden layer
            output_size: Number of output neurons
        """
        super(FFNeuralNetwork, self).__init__()

        self.inputSize = input_size
        self.hiddenSize = hidden_size
        self.outputSize = output_size

        # Initialize weights with random values from normal distribution
        self.W1 = nn.Parameter(torch.randn(self.inputSize, self.hiddenSize))
        self.W2 = nn.Parameter(torch.randn(self.hiddenSize, self.outputSize))

        # Intermediate variables for forward pass
        self.z = None  # Input to hidden layer (before activation)
        self.z2 = None  # Hidden layer output (after activation)
        self.z3 = None  # Input to output layer (before activation)

        # Intermediate variables for backward pass
        self.z_activation = None
        self.z_activation_derivative = None
        self.out_error = None
        self.out_delta = None
        self.z2_error = None
        self.z2_delta = None

    def activation(self, z):
        """Apply sigmoid activation function.

        Args:
            z: Input tensor
```

```
        Returns:
            Tensor: Activated output
        """
        self.z_activation = ActivationFunction.sigmoid(z)
        return self.z_activation

    def activation_derivative(self, z):
        """Compute derivative of sigmoid activation.

        Args:
            z: Input tensor (typically activated output)

        Returns:
            Tensor: Derivative of activation
        """
        self.z_activation_derivative = ActivationFunction.sigmoid_derivative(z)
        return self.z_activation_derivative

    def forward(self, X):
        """Perform forward propagation through the network.

        Args:
            X: Input tensor of shape (batch_size, inputSize)

        Returns:
            Tensor: Output of the network, shape (batch_size, outputSize)
        """
        # Layer 1: Input to Hidden
        self.z = torch.matmul(X, self.W1)  # z = X @ W1
        self.z2 = self.activation(self.z)  # z2 = sigmoid(z)

        # Layer 2: Hidden to Output
        self.z3 = torch.matmul(self.z2, self.W2)  # z3 = z2 @ W2
        output = self.activation(self.z3)  # output = sigmoid(z3)

        return output

    def backward(self, X, y, output, learning_rate):
        """Perform backward propagation and update weights.

        Args:
            X: Input tensor of shape (batch_size, inputSize)
            y: Target tensor of shape (batch_size, outputSize)
            output: Predicted output from forward pass
            learning_rate: Learning rate for weight updates
        """
        # Output layer error and delta
```

```
        self.out_error = y - output  # Error at output
        self.out_delta = self.out_error * self.activation_derivative(output)

        # Hidden layer error and delta
        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
        with torch.no_grad():
            self.W1 += torch.matmul(torch.t(X), self.z2_delta) * learning_rate
            self.W2 += torch.matmul(torch.t(self.z2), self.out_delta) *␣
↪learning_rate

    def train(self, X, y, learning_rate):
        """Train the network for one epoch.

        Args:
            X: Input tensor of shape (batch_size, inputSize)
            y: Target tensor of shape (batch_size, outputSize)
            learning_rate: Learning rate for weight updates
        """
        # Forward pass
        output = self.forward(X)

        # Backward pass
        self.backward(X, y, output, learning_rate)

    @staticmethod
    def save_weights(model, path):
        """Save model state_dict to file.

        This method saves only the weights (state_dict) instead of the entire
        model object, which is more secure and avoids pickle-related issues.

        Args:
            model: FFNeuralNetwork instance to save
            path: File path to save the state_dict
        """
        torch.save(model.state_dict(), path)
        print(f"Model state_dict saved to {path}")

    @staticmethod
    def load_weights(path, input_size=3, hidden_size=4, output_size=1):
        """Load model weights from file.

        This method creates a new model instance and loads the saved state_dict.
        This approach avoids pickle security issues and is compatible with
```

```
        PyTorch 2.6+ default security settings.

        Args:
            path: File path to load the state_dict from
            input_size: Number of input features (must match saved model)
            hidden_size: Number of neurons in hidden layer (must match saved␣
↪model)
            output_size: Number of output neurons (must match saved model)

        Returns:
            FFNeuralNetwork: Model instance with loaded weights
        """
        # Create a new model instance with the same architecture
        model = FFNeuralNetwork(input_size=input_size,
                                hidden_size=hidden_size,
                                output_size=output_size)

        # Load the state_dict
        state_dict = torch.load(path, weights_only=True)
        model.load_state_dict(state_dict)

        print(f"Model state_dict loaded from {path}")
        return model

    def predict(self, x_predict):
        """Make prediction on input data.

        Args:
            x_predict: Input tensor for prediction

        Returns:
            Tensor: Predicted output
        """
        print("Predict data based on trained weights:")
        print("Input:\n", x_predict)
        output = self.forward(x_predict)
        print("Output:\n", output)
        return output
```

2.5 STEP 9 — Prepare Sample Data

Create and normalize training data (X, y) and test data (x_predict).

[40]:
```
# Training data
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

print("Original X:")
print(X)
print("\nOriginal y:")
print(y)

# Normalize X by dividing each column by its max value
X_max, _ = torch.max(X, 0)
X = torch.div(X, X_max)
print("\nNormalized X (divided by column max):")
print(X)

# Normalize y by dividing by 100 (max test score)
y = y / 100
print("\nNormalized y (divided by 100):")
print(y)
print(X)
```

```
Original X:
tensor([[2., 9., 0.],
        [1., 5., 1.],
        [3., 6., 2.]])

Original y:
tensor([[ 90.],
        [100.],
        [ 88.]])

Normalized X (divided by column max):
tensor([[0.6667, 1.0000, 0.0000],
        [0.3333, 0.5556, 0.5000],
        [1.0000, 0.6667, 1.0000]])

Normalized y (divided by 100):
tensor([[0.9000],
        [1.0000],
        [0.8800]])
tensor([[0.6667, 1.0000, 0.0000],
        [0.3333, 0.5556, 0.5000],
        [1.0000, 0.6667, 1.0000]])
```

[41]:
```
# Prediction data
x_predict = torch.tensor([[3, 8, 4]], dtype=torch.float)  # 1 x 3 tensor
```

```
print("Original x_predict:")
print(x_predict)

# Normalize x_predict
x_predict = torch.div(x_predict, X_max)
print("\nNormalized x_predict:")
print(x_predict)
print(X_max)
```

```
Original x_predict:
tensor([[3., 8., 4.]])

Normalized x_predict:
tensor([[1.0000, 0.8889, 2.0000]])
tensor([3., 9., 2.])
```

2.6 STEP 10 — Training Loop

Train the network for 1000 epochs, print loss after each epoch.

[42]:
```
# Create neural network instance
model = FFNeuralNetwork(input_size=3, hidden_size=4, output_size=1)

# Training parameters
num_epochs = 1000
learning_rate = 0.1

print("Starting training...")
print(f"Number of epochs: {num_epochs}")
print(f"Learning rate: {learning_rate}\n")

# Training loop
for epoch in range(num_epochs):
    # Forward pass to compute loss
    output = model(X)
    loss = torch.mean((y - output) ** 2).detach().item()

    # Print loss every 100 epochs
    if (epoch + 1) % 100 == 0 or epoch == 0:
        print(f"Epoch #{epoch + 1} - Loss: {loss:.6f}")

    # Training step
    model.train(X, y, learning_rate)

print("\nTraining completed!")
```

```
Starting training…
Number of epochs: 1000
Learning rate: 0.1
```

```
Epoch #1 - Loss: 0.374021
Epoch #100 - Loss: 0.030377
Epoch #200 - Loss: 0.009770
Epoch #300 - Loss: 0.005774
Epoch #400 - Loss: 0.004424
Epoch #500 - Loss: 0.003831
Epoch #600 - Loss: 0.003528
Epoch #700 - Loss: 0.003356
Epoch #800 - Loss: 0.003249
Epoch #900 - Loss: 0.003179
Epoch #1000 - Loss: 0.003129

Training completed!
```

[43]:
```
model.predict(x_predict)
```

```
Predict data based on trained weights:
Input:
 tensor([[1.0000, 0.8889, 2.0000]])
Output:
 tensor([[0.9303]], grad_fn=<MulBackward0>)
```

[43]:
```
tensor([[0.9303]], grad_fn=<MulBackward0>)
```

[44]:
```
p = model.predict(x_predict)

print(f"\nPredicted value (scaled): {p.item():.6f}")
```

```
Predict data based on trained weights:
Input:
 tensor([[1.0000, 0.8889, 2.0000]])
Output:
 tensor([[0.9303]], grad_fn=<MulBackward0>)

Predicted value (scaled): 0.930308
```

[45]:
```
model.state_dict()
```

[45]:
```
OrderedDict([('W1',
              tensor([[ 1.0158e+00,  2.8769e-01, -4.5191e-01, -1.5456e+00],
                      [ 1.4163e-03,  4.6565e-01, -6.5942e-01, -2.3856e-01],
                      [ 1.5531e+00, -5.8867e-01, -8.9156e-01,  4.2244e-01]])),
             ('W2',
              tensor([[1.9637],
                      [1.2179],
                      [0.2600],
                      [0.6201]]))])
```

2.7 STEP 8 — Save Model

Save the trained model's state_dict (weights only) to file. This approach is more secure and compatible with PyTorch 2.6+ security defaults, avoiding pickle-related issues.

[46]:
```
print(model.state_dict().keys())
```

```
odict_keys(['W1', 'W2'])
```

[47]:
```
# Save trained model
model_path = "ffnn_model.pth"
FFNeuralNetwork.save_weights(model, model_path)
```

```
Model state_dict saved to ffnn_model.pth
```

2.8 STEP 11 — Prediction

Load saved state_dict by creating a new model instance with matching architecture, then load the weights. Perform prediction on new data.

[48]:
```
# Load saved model
# Note: Must specify architecture parameters matching the saved model
loaded_model = FFNeuralNetwork.load_weights(model_path,
                                            input_size=3,
                                            hidden_size=4,
                                            output_size=1)

# Make prediction
prediction = loaded_model.predict(x_predict)
print(f"\nPredicted value (scaled): {prediction.item():.6f}")
print(f"Predicted value (original scale): {prediction.item() * 100:.2f}")
```

```
Model state_dict loaded from ffnn_model.pth
Predict data based on trained weights:
Input:
 tensor([[1.0000, 0.8889, 2.0000]])
Output:
 tensor([[0.9303]], grad_fn=<MulBackward0>)

Predicted value (scaled): 0.930308
Predicted value (original scale): 93.03
```

2.9 STEP 12 — Experiments

Experiment with different hyperparameters to observe changes in loss and output.

[49]:
```
# Experiment 1: Different learning rates
print("=" * 60)
print("Experiment 1: Testing different learning rates")
print("=" * 60)
```

```
learning_rates = [0.01, 0.1, 0.5, 1.0]
for lr in learning_rates:
    model_exp = FFNeuralNetwork(input_size=3, hidden_size=4, output_size=1)

    # Train for 500 epochs
    for epoch in range(500):
        model_exp.train(X, y, lr)

    output = model_exp(X)
    loss = torch.mean((y - output) ** 2).detach().item()
    print(f"Learning rate: {lr:4.2f} - Final Loss: {loss:.6f}")

print()
```

```
============================================================
Experiment 1: Testing different learning rates
============================================================

Learning rate: 0.01 - Final Loss: 0.063629
Learning rate: 0.10 - Final Loss: 0.004101
Learning rate: 0.50 - Final Loss: 0.002876
Learning rate: 1.00 - Final Loss: 0.003301
```

2.9.1 Config Hidden layer

[50]:
```
# Experiment 2: Different number of hidden neurons
print("=" * 60)
print("Experiment 2: Testing different hidden layer sizes")
print("=" * 60)

hidden_sizes = [2, 4, 6, 8, 16]
for hidden_size in hidden_sizes:
    model_exp = FFNeuralNetwork(input_size=3, hidden_size=hidden_size,␣
 ↪output_size=1)

    # Train for 1000 epochs
    for epoch in range(1000):
        model_exp.train(X, y, 0.1)

    output = model_exp(X)
    loss = torch.mean((y - output) ** 2).detach().item()
    print(f"Hidden size: {hidden_size:2d} - Final Loss: {loss:.6f}")

print()
```

```
============================================================
Experiment 2: Testing different hidden layer sizes
```

```
============================================================
Hidden size:  2 - Final Loss: 0.003283
Hidden size:  4 - Final Loss: 0.003004
Hidden size:  6 - Final Loss: 0.004316
Hidden size:  8 - Final Loss: 0.003604
Hidden size: 16 - Final Loss: 0.003059
```

[51]:
```python
# Experiment 3: Compare predictions with different configurations
print("=" * 60)
print("Experiment 3: Predictions with different models")
print("=" * 60)

configs = [
    {"hidden_size": 4, "lr": 0.1, "name": "Default (4 neurons, lr=0.1)"},
    {"hidden_size": 8, "lr": 0.1, "name": "More neurons (8 neurons, lr=0.1)"},
    {"hidden_size": 4, "lr": 0.5, "name": "Higher LR (4 neurons, lr=0.5)"},
]

for config in configs:
    model_exp = FFNeuralNetwork(
        input_size=3,
        hidden_size=config["hidden_size"],
        output_size=1
    )

    # Train for 1000 epochs
    for epoch in range(1000):
        model_exp.train(X, y, config["lr"])

    prediction = model_exp.predict(x_predict)
    print(f"{config['name']}: {prediction.item() * 100:.2f}")
    print()
```

```
============================================================
Experiment 3: Predictions with different models
============================================================

Predict data based on trained weights:
Input:
 tensor([[1.0000, 0.8889, 2.0000]])
Output:
 tensor([[0.9763]], grad_fn=<MulBackward0>)
Default (4 neurons, lr=0.1): 97.63

Predict data based on trained weights:
Input:
 tensor([[1.0000, 0.8889, 2.0000]])
Output:
```

```
 tensor([[0.9748]], grad_fn=<MulBackward0>)
More neurons (8 neurons, lr=0.1): 97.48

Predict data based on trained weights:
Input:
 tensor([[1.0000, 0.8889, 2.0000]])
Output:
 tensor([[0.9453]], grad_fn=<MulBackward0>)
Higher LR (4 neurons, lr=0.5): 94.53
```