MLP forward and back pass implementation

Implementation of a **Multilayer Perceptron (MLP)** for the MNIST dataset using only **NumPy**. This implementation includes:

1. Loading and preprocessing the MNIST dataset.
2. Defining the MLP architecture (input layer, hidden layer, output layer).
3. Implementing forward propagation, backward propagation, and gradient descent.
4. Training the model and evaluating its performance.

Implementation

```
import numpy as np
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder

# Load MNIST dataset
mnist = fetch_openml('mnist_784', version=1)
X, y = mnist["data"], mnist["target"]

# Preprocess data
X = X / 255.0  # Normalize pixel values to [0, 1]
y = y.astype(int)

# One-hot encode labels
encoder = OneHotEncoder(sparse=False)
y = encoder.fit_transform(y.reshape(-1, 1))

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# MLP Class
class MLP:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.1):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.learning_rate = learning_rate

        # Initialize weights and biases
        self.W1 = np.random.randn(self.input_size, self.hidden_size) * 0.01
        self.b1 = np.zeros((1, self.hidden_size))
        self.W2 = np.random.randn(self.hidden_size, self.output_size) * 0.01
```

```python
        self.b2 = np.zeros((1, self.output_size))

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def softmax(self, x):
        exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
        return exp_x / np.sum(exp_x, axis=1, keepdims=True)

    def forward(self, X):
        # Input to hidden layer
        self.z1 = np.dot(X, self.W1) + self.b1
        self.a1 = self.sigmoid(self.z1)

        # Hidden to output layer
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = self.softmax(self.z2)

        return self.a2

    def backward(self, X, y, output):
        # Output layer error
        m = y.shape[0]
        self.dz2 = output - y
        self.dW2 = np.dot(self.a1.T, self.dz2) / m
        self.db2 = np.sum(self.dz2, axis=0, keepdims=True) / m

        # Hidden layer error
        self.dz1 = np.dot(self.dz2, self.W2.T) * self.sigmoid_derivative(self.a1)
        self.dW1 = np.dot(X.T, self.dz1) / m
        self.db1 = np.sum(self.dz1, axis=0, keepdims=True) / m

    def update_weights(self):
        # Update weights and biases
        self.W1 -= self.learning_rate * self.dW1
        self.b1 -= self.learning_rate * self.db1
        self.W2 -= self.learning_rate * self.dW2
        self.b2 -= self.learning_rate * self.db2

    def train(self, X, y, epochs=1000):
        for epoch in range(epochs):
            # Forward pass
            output = self.forward(X)

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

            # Update weights
            self.update_weights()

            # Print loss every 100 epochs
            if epoch % 100 == 0:
                loss = self.cross_entropy_loss(y, output)
                print(f"Epoch {epoch}, Loss: {loss}")

```

```
    def cross_entropy_loss(self, y_true, y_pred):
        m = y_true.shape[0]
        loss = -np.sum(y_true * np.log(y_pred + 1e-15)) / m  # Add small epsilon to avoid log(0)
        return loss

    def predict(self, X):
        output = self.forward(X)
        return np.argmax(output, axis=1)

    def accuracy(self, y_true, y_pred):
        return np.mean(y_true == y_pred)

# Initialize MLP
input_size = 784  # 28x28 pixels
hidden_size = 64
output_size = 10  # 10 classes (digits 0-9)
mlp = MLP(input_size, hidden_size, output_size, learning_rate=0.1)

# Train the model
mlp.train(X_train, y_train, epochs=1000)

# Evaluate the model
y_pred = mlp.predict(X_test)
y_true = np.argmax(y_test, axis=1)
accuracy = mlp.accuracy(y_true, y_pred)
print(f"Test Accuracy: {accuracy * 100:.2f}%")
```

Explanation of the Code

1. **Data Preprocessing**:
   - The MNIST dataset is loaded and normalized to pixel values between 0 and 1.
   - Labels are one-hot encoded for multi-class classification.
2. **MLP Class**:
   - **Initialization**: Weights and biases are initialized randomly.
   - **Activation Functions**:
     - Sigmoid is used for the hidden layer.
     - Softmax is used for the output layer to handle multi-class classification.
   - **Forward Propagation**: Computes the output of each layer.
   - **Backward Propagation**: Computes gradients using the chain rule.
   - **Weight Update**: Updates weights and biases using gradient descent.
   - **Training**: Iterates over epochs, performs forward and backward passes, and updates weights.
   - **Loss Function**: Cross-entropy loss is used to measure the error.
   - **Prediction**: Predicts the class with the highest probability.
3. **Training and Evaluation**:
   - The model is trained on the training set.
   - Accuracy is computed on the test set.

Output

- During training, the loss is printed every 100 epochs.
- After training, the test accuracy is displayed.

Example Output

```
Epoch 0, Loss: 2.3025850929940455
Epoch 100, Loss: 0.3456789123456789
Epoch 200, Loss: 0.2345678912345678
...
Test Accuracy: 95.23%
```

Pytorch Example

Implementation of an **MLP (Multilayer Perceptron)** for MNIST classification using **PyTorch**. This implementation includes:

1. Loading and preprocessing the MNIST dataset.
2. Defining the MLP architecture.
3. Training the model using PyTorch's automatic differentiation and optimization tools.
4. Evaluating the model on the test set.

Implementation

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# Set device (GPU if available, else CPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Hyperparameters
input_size = 784  # 28x28 pixels
hidden_size = 128
output_size = 10  # 10 classes (digits 0-9)
learning_rate = 0.001
batch_size = 64
num_epochs = 5

# Load MNIST dataset
```

```python
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])

train_dataset = datasets.MNIST(root="./data", train=True, transform=transform, download=True)
test_dataset = datasets.MNIST(root="./data", train=False, transform=transform, download=True)

train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)

# Define MLP model
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten the input
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out

# Initialize model, loss function, and optimizer
model = MLP(input_size, hidden_size, output_size).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Training loop
total_steps = len(train_loader)
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Move tensors to the configured device
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Print loss every 100 steps
        if (i + 1) % 100 == 0:
            print(f"Epoch [{epoch + 1}/{num_epochs}], Step [{i + 1}/{total_steps}], Loss: {loss.item():.4f}")

# Test the model
model.eval()  # Set model to evaluation mode
with torch.no_grad():
    correct = 0
    total = 0
```

```python
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print(f"Test Accuracy: {100 * correct / total:.2f}%")
```

Explanation of the Code

1. **Data Loading and Preprocessing**:
   - The MNIST dataset is loaded using `torchvision.datasets`.
   - Images are normalized using `transforms.Normalize` with mean `0.1307` and standard deviation `0.3081`.
   - Data is split into training and test sets using `DataLoader`.
2. **Model Definition**:
   - The MLP is defined as a subclass of `nn.Module`.
   - It has one hidden layer with ReLU activation and an output layer.
   - The `forward` method defines the forward pass, including flattening the input.
3. **Training**:
   - The model is trained using the Adam optimizer and CrossEntropyLoss.
   - The training loop iterates over the dataset for a specified number of epochs.
   - Loss is printed every 100 steps.
4. **Evaluation**:
   - The model is evaluated on the test set.
   - Accuracy is calculated by comparing predicted labels with true labels.

Output

During training, the loss is printed every 100 steps. After training, the test accuracy is displayed.

Example Output:

```
Epoch [1/5], Step [100/938], Loss: 0.4567
Epoch [1/5], Step [200/938], Loss: 0.3456
...
Epoch [5/5], Step [900/938], Loss: 0.1234
Test Accuracy: 97.89%
```

Key Points

1. **Device Setup**:
   - The code automatically uses GPU if available, otherwise CPU.
2. **Model Architecture**:
   - The MLP has one hidden layer with 128 neurons and ReLU activation.
3. **Optimization**:
   - Adam optimizer is used for faster convergence.
4. **Evaluation**:
   - The model achieves high accuracy on the MNIST test set.

Keras Example

Below is the implementation of an **MLP (Multilayer Perceptron)** for MNIST classification using **Keras** (with TensorFlow as the backend). This implementation includes:

1. Loading and preprocessing the MNIST dataset.
2. Defining the MLP architecture using Keras.
3. Training the model using Keras's high-level API.
4. Evaluating the model on the test set.

Implementation

```python
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load MNIST dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

# Preprocess data
X_train = X_train.reshape(-1, 28 * 28).astype('float32') / 255.0  # Flatten and normalize
X_test = X_test.reshape(-1, 28 * 28).astype('float32') / 255.0

# One-hot encode labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Define MLP model
model = models.Sequential([
    layers.Dense(128, activation='relu', input_shape=(28 * 28,)),  # Hidden layer with 128 neurons
    layers.Dense(10, activation='softmax')  # Output layer with 10 neurons (for 10 classes)
])

```

```
# Compile the model
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
model.fit(X_train, y_train, epochs=5, batch_size=64, validation_split=0.2)

# Evaluate the model
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

Explanation of the Code

1. **Data Loading and Preprocessing**:
   - The MNIST dataset is loaded using `tensorflow.keras.datasets.mnist`.
   - The images are flattened (from 28x28 to 784) and normalized to the range `[0, 1]`.
   - Labels are one-hot encoded using `to_categorical`.
2. **Model Definition**:
   - The MLP is defined using Keras's `Sequential` API.
   - It has one hidden layer with 128 neurons and ReLU activation.
   - The output layer has 10 neurons with softmax activation for multi-class classification.
3. **Model Compilation**:
   - The model is compiled with the Adam optimizer and categorical cross-entropy loss.
   - Accuracy is used as the evaluation metric.
4. **Training**:
   - The model is trained for 5 epochs with a batch size of 64.
   - 20% of the training data is used as a validation set.
5. **Evaluation**:
   - The model is evaluated on the test set, and the test accuracy is printed.

Output

During training, the loss and accuracy for both the training and validation sets are printed for each epoch. After training, the test accuracy is displayed.

Example Output:

```
Epoch 1/5
750/750 [==============================] - 2s 2ms/step - loss: 0.3456 - accuracy: 0.9012 - val_loss: 0.1789 - val_accuracy: 0.9487
Epoch 2/5
750/750 [==============================] - 1s 2ms/step - loss: 0.1567 - accuracy: 0.9543 - val_loss: 0.1234 - val_accuracy: 0.9632
Epoch 3/5
750/750 [==============================] - 1s 2ms/step - loss: 0.1123 - accuracy: 0.9678 - val_loss: 0.1023 - val_accuracy: 0.9701
Epoch 4/5
750/750 [==============================] - 1s 2ms/step - loss: 0.0890 - accuracy: 0.9745 - val_loss: 0.0898 - val_accuracy: 0.9734
Epoch 5/5
750/750 [==============================] - 1s 2ms/step - loss: 0.0734 - accuracy: 0.9789 - val_loss: 0.0812 - val_accuracy: 0.9765
313/313 [==============================] - 0s 1ms/step - loss: 0.0789 - accuracy: 0.9756
Test Accuracy: 97.56%
```

Key Points

1. **Simplicity**:
   - Keras provides a high-level API, making the implementation concise and easy to understand.
2. **Model Architecture**:
   - The MLP has one hidden layer with 128 neurons and ReLU activation.
   - The output layer uses softmax activation for multi-class classification.
3. **Optimization**:
   - The Adam optimizer is used for faster convergence.
4. **Evaluation**:
   - The model achieves high accuracy on the MNIST test set.

Pytorch VS Keras

The two implementations of the **MLP for MNIST classification**—one using **PyTorch** and the other using **Keras**—highlight the differences in design philosophy, syntax, and workflow between the two frameworks. Below is a detailed comparison of the two APIs for this specific implementation:

---

1. Syntax and Code Structure

PyTorch

- **Explicit and Flexible**:
  - PyTorch requires more explicit code for defining the model, training loop, and evaluation.
  - The model is defined as a class inheriting from `nn.Module`, and the forward pass is explicitly implemented in the `forward` method.
  - The training loop (forward pass, loss computation, backward pass, and weight update) is written manually.
- **Example**:

```
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten the input
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```

Keras

- **High-Level and Concise**:
  - Keras provides a high-level API that abstracts away many details, making the code more concise.
  - The model is defined using the `Sequential` API, where layers are stacked sequentially.
  - The training loop is handled internally by the `fit` method, and evaluation is done using `evaluate`.
- **Example**:

```
model = models.Sequential([
    layers.Dense(128, activation='relu', input_shape=(28 * 28,)),
    layers.Dense(10, activation='softmax')
])
```

---

2. Model Definition

PyTorch

- **Customizable**:
  - The model is defined as a Python class, allowing for highly customizable architectures.
  - The `forward` method explicitly defines how input data flows through the network.
- **Example**:

```python
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten the input
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out
```

Keras

- **Simplified**:
  - The model is defined using a sequential stack of layers, which is ideal for simple architectures.
  - Custom architectures can still be defined using the Functional API or subclassing `tf.keras.Model`.
- **Example**:

```python
model = models.Sequential([
    layers.Dense(128, activation='relu', input_shape=(28 * 28,)),
    layers.Dense(10, activation='softmax')
])
```

---

3. Training Loop

PyTorch

- **Manual**:
  - The training loop is written manually, including the forward pass, loss computation, backward pass, and weight update.
  - This provides full control over the training process but requires more code.
- **Example**:

```python
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward pass and optimization
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Keras

- **Automatic**:
  - The training loop is handled internally by the `fit` method.
  - This simplifies the code but provides less flexibility for custom training loops.
- **Example**:

```
model.fit(X_train, y_train, epochs=5, batch_size=64, validation_split=0.2)
```

---

4. Evaluation

PyTorch

- **Manual**:
  - Evaluation is done manually by iterating over the test dataset and computing accuracy.
- **Example**:

```
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print(f"Test Accuracy: {100 * correct / total:.2f}%")
```

Keras

- **Automatic**:
  - Evaluation is done using the `evaluate` method, which computes the loss and metrics (e.g., accuracy) automatically.
- **Example**:

```
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

---

5. Device Management

PyTorch

- **Explicit**:
  - The device (CPU or GPU) must be explicitly specified, and tensors must be moved to the device manually.
- **Example**:

```python
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = MLP(input_size, hidden_size, output_size).to(device)
images = images.to(device)
labels = labels.to(device)
```

Keras

- **Automatic**:
  - Keras automatically uses the available GPU (if TensorFlow is installed with GPU support) without requiring explicit device management.

---

6. Flexibility vs. Simplicity

PyTorch

- **Flexibility**:
  - PyTorch is more flexible and suitable for research and custom architectures.
  - It provides full control over the training process and model definition.

Keras

- **Simplicity**:
  - Keras is more user-friendly and ideal for quick prototyping and standard use cases.
  - It abstracts away many details, making the code shorter and easier to write.

Summary of Differences

Feature	PyTorch	Keras
Syntax	Explicit, more code	High-level, concise
Model Definition	Custom class with `forward` method	Sequential or Functional API
Training Loop	Manual	Automatic (`fit` method)
Evaluation	Manual	Automatic (`evaluate` method)
Device Management	Explicit	Automatic
Flexibility	High	Moderate

Feature	PyTorch	Keras
Use Case	Research, custom architectures	Prototyping, standard use cases

---

Which One to Use?

- **PyTorch**: If you need full control over the training process, are working on research, or need to implement custom architectures.
- **Keras**: If you want to quickly prototype models, prefer a high-level API, or are working on standard use cases.

Both frameworks are powerful and widely used, so the choice depends on your specific needs and preferences!

Pytorch Wrapper Example

In PyTorch, there is no direct equivalent of Keras's `model.compile()` method, as PyTorch is designed to be more flexible and explicit. However, starting from **PyTorch 1.10**, the `torch.compile()` function was introduced to optimize model execution, but it is not the same as Keras's `compile()`.

To mimic the simplicity of Keras's `model.compile()` in PyTorch, we can create a wrapper class that abstracts away the training loop, loss function, and optimizer setup. Below is an implementation of an **MLP for MNIST classification** in PyTorch, using a custom `compile()`-like method for simplicity.

Implementation

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms

# Set device (GPU if available, else CPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Hyperparameters
input_size = 784  # 28x28 pixels
hidden_size = 128
output_size = 10  # 10 classes (digits 0-9)
learning_rate = 0.001
batch_size = 64
num_epochs = 5

# Load MNIST dataset
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])

train_dataset = datasets.MNIST(root="./data", train=True, transform=transform, download=True)
```

```python
test_dataset = datasets.MNIST(root="./data", train=False, transform=transform, download=True)

train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)

# Define MLP model
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten the input
        out = self.fc1(x)
        out = self.relu(out)
        out = self.fc2(out)
        return out

# Custom wrapper to mimic Keras's compile() and fit()
class PyTorchModel:
    def __init__(self, model):
        self.model = model.to(device)
        self.criterion = None
        self.optimizer = None

    def compile(self, loss_fn, optimizer, metrics=None):
        self.criterion = loss_fn
        self.optimizer = optimizer
        self.metrics = metrics

    def fit(self, train_loader, epochs, val_loader=None):
        for epoch in range(epochs):
            self.model.train()
            for i, (images, labels) in enumerate(train_loader):
                images = images.to(device)
                labels = labels.to(device)

                # Forward pass
                outputs = self.model(images)
                loss = self.criterion(outputs, labels)

                # Backward pass and optimization
                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()

                # Print loss every 100 steps
                if (i + 1) % 100 == 0:
                    print(f"Epoch [{epoch + 1}/{epochs}], Step [{i + 1}/{len(train_loader)}], Loss: {loss.item():.4f}")

            # Validation
            if val_loader:
                self.model.eval()
                with torch.no_grad():
```

```python
                correct = 0
                total = 0
                for images, labels in val_loader:
                    images = images.to(device)
                    labels = labels.to(device)
                    outputs = self.model(images)
                    _, predicted = torch.max(outputs.data, 1)
                    total += labels.size(0)
                    correct += (predicted == labels).sum().item()

                print(f"Epoch [{epoch + 1}/{epochs}], Validation Accuracy: {100 * correct / total:.2f}%")

    def evaluate(self, test_loader):
        self.model.eval()
        with torch.no_grad():
            correct = 0
            total = 0
            for images, labels in test_loader:
                images = images.to(device)
                labels = labels.to(device)
                outputs = self.model(images)
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

            print(f"Test Accuracy: {100 * correct / total:.2f}%")

# Initialize model, loss function, and optimizer
model = MLP(input_size, hidden_size, output_size)
pytorch_model = PyTorchModel(model)
pytorch_model.compile(loss_fn=nn.CrossEntropyLoss(), optimizer=optim.Adam(model.parameters(), lr=learning_rate))

# Train the model
pytorch_model.fit(train_loader, epochs=num_epochs, val_loader=test_loader)

# Evaluate the model
pytorch_model.evaluate(test_loader)
```

Explanation of the Code

1. **Custom Wrapper Class**:
   - The `PyTorchModel` class mimics Keras's `compile()` and `fit()` methods.
   - `compile()` sets up the loss function, optimizer, and metrics.
   - `fit()` handles the training loop, including validation if a validation loader is provided.
   - `evaluate()` computes the accuracy on the test set.
2. **Model Definition**:
   - The MLP is defined as a standard PyTorch `nn.Module`.
3. **Training Loop**:
   - The `fit()` method iterates over the dataset for a specified number of epochs.

- It prints the loss every 100 steps and computes validation accuracy at the end of each epoch.

4. **Evaluation**:

- The `evaluate()` method computes the test accuracy.

Output

During training, the loss is printed every 100 steps, and validation accuracy is printed at the end of each epoch. After training, the test accuracy is displayed.

Example Output:

```
Epoch [1/5], Step [100/938], Loss: 0.4567
Epoch [1/5], Step [200/938], Loss: 0.3456
...
Epoch [5/5], Step [900/938], Loss: 0.1234
Epoch [5/5], Validation Accuracy: 97.89%
Test Accuracy: 97.89%
```

Key Points

1. **Custom Wrapper**:

- The `PyTorchModel` class abstracts away the training loop, making the code more Keras-like.

2. **Flexibility**:

- PyTorch still provides full control over the model and training process.

3. **Simplicity**:

- The `compile()` and `fit()` methods make the code easier to read and write.

This implementation bridges the gap between PyTorch's flexibility and Keras's simplicity, providing a more user-friendly interface for training and evaluating models in PyTorch.

PyTorch vs Keras API Styles

Imperative vs Declarative API Styles

PyTorch's Imperative/Define-by-Run Style

PyTorch employs an imperative programming style with define-by-run computation graphs. This means:

- **Immediate Execution**: Operations execute as they're written, making the code feel more like native Python.
- **Dynamic Graphs**: The computation graph is created dynamically during execution, allowing for flexibility in model architecture.
- **Pythonic Nature**: Integrates seamlessly with Python, making it intuitive for developers familiar with Python.

Advantages:

- **Flexibility**: Ideal for complex models and research where architectures might change during development.
- **Debugging**: Errors appear immediately with clear stack traces, simplifying the debugging process.
- **Control**: Offers granular control over tensor operations and model components.

Disadvantages:

- **Verbosity**: Requires more code for model definition and training loops.
- **Performance**: Dynamic graphs can be less efficient for production deployment compared to static graphs.
- **Learning Curve**: Steeper for beginners due to the need to understand lower-level concepts.

Keras' Declarative/Define-and-Run Style

Keras uses a declarative programming style with define-and-run computation graphs. This means:

- **Model Definition First**: The model architecture is defined before any operations are executed.
- **Static Graphs**: The computation graph is compiled into a static structure before execution.
- **High-Level Abstraction**: Provides a simplified interface that abstracts many low-level details.

Advantages:

- **Simplicity**: Minimal code required for model definition and training, making it accessible for beginners.
- **Rapid Prototyping**: Faster development cycles due to less boilerplate code.
- **Production Readiness**: Static graphs can be more efficient for deployment in production environments.

Disadvantages:

- **Flexibility**: Less suitable for highly customized or complex model architectures.
- **Debugging**: Errors might appear during graph compilation rather than immediately when writing code.
- **Control**: Less direct access to low-level operations compared to imperative styles.

Other API Styles

Beyond the imperative and declarative styles, there are hybrid approaches and variations:

1. **TensorFlow's Eager Execution**: TensorFlow 2.x introduced eager execution, which combines the immediacy of imperative programming with the efficiency of static graphs through `tf.function` for graph compilation.
2. **Hybrid Approaches**: Some frameworks allow both styles within the same ecosystem. For example, PyTorch offers higher-level libraries like PyTorch Lightning for reduced boilerplate, while Keras can be used with TensorFlow's lower-level APIs when more control is needed.
3. **Low-Level vs High-Level APIs**: Many frameworks offer both low-level APIs for fine-grained control and high-level APIs for simplicity. PyTorch has `torch.nn` for high-level components and tensor operations for low-level manipulation, while Keras sits atop TensorFlow's lower-level functionality.

Web API Specification

Model Training Endpoint

```
POST /api/train
Content-Type: application/json

{
    "model_config": {
        "type": "MLP",
        "layers": [
            {"type": "Dense", "input_dim": 784, "output_dim": 256},
            {"type": "Activation", "func": "relu"},
            {"type": "Dense", "input_dim": 256, "output_dim": 10}
        ]
    },
    "dataset": "mnist",
    "hyperparameters": {
        "epochs": 10,
        "batch_size": 32,
        "optimizer": "adam",
        "loss": "cross_entropy",
        "metrics": ["accuracy"]
    }
}
```

Model Inference Endpoint

```
POST /api/predict
Content-Type: application/json

{
    "model_id": "mlp_mnist_20230525",
    "input_data": [[0.1, 0.2, ..., 0.784]]
}
```

Model Management Endpoint

```
GET /api/models
GET /api/models/{model_id}
DELETE /api/models/{model_id}
```

Training Status Endpoint

```
GET /api/train/{job_id}
```

Error Handling and Debugging

The API returns detailed error messages in JSON format:


```json
{
    "error": "InvalidInput",
    "message": "Input dimension mismatch",
    "details": "Expected 784 features but received 512",
    "traceback": "..."
}
```