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


# Set device
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# Step 1: Prepare the MNIST Dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

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

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/raw/train-images-idx3-ubyte.gz
100%|██████████| 9912422/9912422 [00:00<00:00, 81865801.93it/s]
Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/raw/train-labels-idx1-ubyte.gz
100%|██████████| 28881/28881 [00:00<00:00, 17871893.45it/s]Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./d

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw/t10k-images-idx3-ubyte.gz

100%|██████████| 1648877/1648877 [00:00<00:00, 23860983.77it/s]
Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
100%|██████████| 4542/4542 [00:00<00:00, 10738742.26it/s]
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
```

```
# Step 2: Define the Vision Transformer Model
class VisionTransformer(nn.Module):
    def __init__(self, image_size, patch_size, num_classes, dim, depth, heads, mlp_dim):
        super(VisionTransformer, self).__init__()
        num_patches = (image_size // patch_size) ** 2
        self.patch_embedding = nn.Conv2d(1, dim, kernel_size=patch_size, stride=patch_size)
        self.positional_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.transformer_encoder = nn.TransformerEncoder(nn.TransformerEncoderLayer(d_model=dim, nhead=heads, dim_feedforwar
        self.fc = nn.Linear(dim, num_classes)

    def forward(self, x):
        B, C, H, W = x.shape
        x = self.patch_embedding(x)
        x = x.flatten(2).transpose(1, 2)
        x = torch.cat((self.positional_embedding.repeat(B, 1, 1), x), dim=1)
        x = self.transformer_encoder(x)
        x = x.mean(1)
        x = self.fc(x)
        return x


# Step 3: Training
image_size = 28
patch_size = 7
num_classes = 10
dim = 64
depth = 6
heads = 8
mlp_dim = 128
num_epochs = 10

model = VisionTransformer(image_size, patch_size, num_classes, dim, depth, heads, mlp_dim).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/transformer.py:286: UserWarning: enable_nested_tensor is True,
  warnings.warn(f"enable_nested_tensor is True, but self.use_nested_tensor is False because {why_not_sparsity_fast_path}
```

```
# Training loop
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * images.size(0)
    epoch_loss = running_loss / len(train_loader.dataset)
    print(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {epoch_loss:.4f}')
```

```
Epoch [1/10], Loss: 1.1184
Epoch [2/10], Loss: 0.2773
Epoch [3/10], Loss: 0.1810
Epoch [4/10], Loss: 0.1451
Epoch [5/10], Loss: 0.1245
Epoch [6/10], Loss: 0.1134
Epoch [7/10], Loss: 0.1041
Epoch [8/10], Loss: 0.0963
Epoch [9/10], Loss: 0.0903
Epoch [10/10], Loss: 0.0831
```

```
# Step 4: Evaluation
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
print(f'Accuracy on the test set: {accuracy:.2f}%')
```

```
Accuracy on the test set: 97.54%
```

Comments on potential areas of extending Classical Vision Transformers to Quantum Vision Transformers

1. **Conceptual Differences:**

   1.1 **Input Representation**: In a CVT, the input is a preprocessed image represented as a 2D tensor. In a QVT, the input image would be encoded into a quantum state using a suitable encoding method (e.g., Amplitude Encoding or NEQR).
   1.2 **Self-Attention**: In a CVT, self-attention uses linear projections and softmax functions. In a QVT, we can explore these options:

   ```
   Quantum Gates: Apply parameterized quantum gates (e.g., Ry, Rz) on qubits representing different image patches
   ```

   Quantum Entanglement: Utilize entanglement between qubits to capture long-range dependencies within the image.

2. **Description of a possible QVT Architecture:**

   ```
   2.1 Quantum Image Encoding: Encode the preprocessed image into a quantum state using a chosen encoding method.

   2.2 Patch-wise Encoding: Divide the image into patches and encode each patch into a separate quantum state. App
   ```

   ```
   2.3 Interacting and Entangled Qubits: Allow qubits representing different image patches to interact using controlled operations (C

   2.4 Quantum Attention Function: Design a quantum circuit that acts as the attention function, potentially using entanglement and m

   2.5 Apply parameterized quantum circuits (series of Ry, Rz, and controlled rotations) on the output of the self-attention layer to

   2.6 Hybrid Networks: with softwares such as pennylane supporting qnn's where quantum layers can be sandwiched between classical la
   ```

```
2.7 Hybrid training structure, even in case of QVT, like others, with Quantum Forward pass circuits and classical dataloading, opt
```

Start coding or generate with AI.