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



# Define the KAN model
class KAN(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=128, output_dim=10):
        super(KAN, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.act1 = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.act2 = nn.ReLU()
        self.fc3 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act1(x)
        x = self.fc2(x)
        x = self.act2(x)
        x = self.fc3(x)
        return x



transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform=transform)
testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)

train_loader = DataLoader(trainset, batch_size=64, shuffle=True)
test_loader = DataLoader(testset, batch_size=64, shuffle=False)
```

```
100%|████████| 9.91M/9.91M [00:00<00:00, 11.4MB/s]
100%|████████| 28.9k/28.9k [00:00<00:00, 359kB/s]
100%|████████| 1.65M/1.65M [00:00<00:00, 2.73MB/s]
100%|████████| 4.54k/4.54k [00:00<00:00, 4.81MB/s]
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = KAN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)



# Training loop
epochs = 10
for epoch in range(epochs):
    model.train()
    total_loss = 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        images = images.view(images.size(0), -1)  # Flatten
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    print(f"Epoch {epoch+1}/{epochs}, Loss: {total_loss/len(train_loader):.4f}")
```

```
Epoch 1/10, Loss: 0.3737
Epoch 2/10, Loss: 0.1693
Epoch 3/10, Loss: 0.1234
Epoch 4/10, Loss: 0.0978
Epoch 5/10, Loss: 0.0832
Epoch 6/10, Loss: 0.0713
Epoch 7/10, Loss: 0.0633
Epoch 8/10, Loss: 0.0589
Epoch 9/10, Loss: 0.0505
Epoch 10/10, Loss: 0.0463
```

```
# Evaluation
model.eval()
```

```
KAN(
  (fc1): Linear(in_features=784, out_features=128, bias=True)
  (act1): ReLU()
  (fc2): Linear(in_features=128, out_features=128, bias=True)
  (act2): ReLU()
  (fc3): Linear(in_features=128, out_features=10, bias=True)
)
```

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

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

```
Test Accuracy: 97.47%
```

Extending Kolmogorov-Arnold Networks (KAN) to a Quantum KAN (QKAN): Architecture and Potential Enhancements

Extending the classical Kolmogorov-Arnold Network (KAN) architecture to a quantum KAN (QKAN) involves leveraging quantum computing principles to enhance the network's computational capabilities. In a QKAN, classical activation functions are replaced with quantum operations, utilizing quantum linear algebra techniques such as quantum singular value transformation. This approach allows for the application of parameterized activation functions on the edges of the network, facilitating efficient processing of high-dimensional data.

The architecture of a QKAN typically includes block-encoding methods, making it inherently suitable for direct quantum input. This design not only aligns with the principles of the Kolmogorov-Arnold representation theorem but also exploits the advantages of quantum computing, such as parallelism and entanglement, to potentially achieve superior performance over classical counterparts.

To implement a QKAN for tasks like MNIST digit classification, the architecture would involve encoding classical image data into quantum states, processing them through quantum layers that emulate the KAN structure, and then measuring the quantum states to obtain the final classification outputs. This hybrid approach aims to combine the strengths of both quantum and classical computing paradigms.

Potential extensions of QKANs could involve integrating them with variational quantum circuits to enhance their adaptability and performance on noisy intermediate-scale quantum (NISQ) devices. Additionally, exploring different quantum feature maps and entanglement strategies may further improve the network's expressiveness and capability to model complex functions.

In summary, transitioning from a classical KAN to a quantum KAN involves substituting classical components with quantum analogs, thereby harnessing quantum computational advantages to potentially outperform classical networks in specific tasks.

Start coding or generate with AI.