Task9

March 25, 2025

1 Task 9:Kolmogorov-Arnold Network

We will be training a KAN network using activation function B-spline on MNIST dataset

We will start by importing the required libraries

[1]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torchvision import datasets, transforms
```

1.1 BSpline Activation Function

This is a custom activation function

It is used to smoothly transform the inputs

`knots` are special points which are used to define the curve

`coeff` are the values that adjust the shape of the curve

`cubic_bspline` function the curve. It creates a smooth, non-linear transformation of input.

`forward` function is like the activation function for neuron in neural network as it applies bspline fxn to each each input x and return the transformed output

[3]:
```
class BSplineActivation(nn.Module):
    def __init__(self, num_basis=5, knot_min=-2, knot_max=2):
        super(BSplineActivation, self).__init__()
        self.num_basis = num_basis
        # Fixed knot locations, they need not be learned.
        self.register_buffer('knots', torch.linspace(knot_min, knot_max,␣
 ↪num_basis))
        self.coeffs = nn.Parameter(torch.ones(num_basis))

    def cubic_bspline(self, x):
        # Compute the cubic B-spline basis value.
        abs_x = torch.abs(x)
        val1 = (2/3) - (x**2) + 0.5 * (abs_x**3)
        val2 = ((2 - abs_x)**3) / 6
```

```python
        return torch.where(abs_x < 1, val1,torch.where((abs_x >= 1) & (abs_x <␣
↪2), val2, torch.zeros_like(x)))

    def forward(self, x):
        #  compute the B-spline value
        out = 0
        for i in range(self.num_basis):
            # calculate (x - knot).
            b_val = self.cubic_bspline(x - self.knots[i])
            out = out + self.coeffs[i] * b_val
        return out
```

1.2 KAN Model

We will define the neural network model

An input layer `fc1`: Converts 28x28 images into a long vector.

A hidden layer `spline`: Uses the B-Spline activation.

An output layer `fc2`: Gives 10 output values (one for each digit 0-9).

`Forward` The image is flattened (28x28 → 784).

A linear transformation (fc1)

The B-Spline activation function

Another linear transformation (fc2)

The final output is 10 values, representing predictions for digits 0-9.

[4]:
```python
class KANNetwork(nn.Module):
    def __init__(self, input_dim=784, hidden_dim=256, num_basis=5,␣
↪num_classes=10):
        super(KANNetwork, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.spline = BSplineActivation(num_basis=num_basis)
        self.fc2 = nn.Linear(hidden_dim, num_classes)

    def forward(self, x):
        # Flatten MNIST images (28x28 -> 784)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        # Apply the spline activation (nonlinear univariate function)
        x = self.spline(x)
        x = self.fc2(x)
        return x
```

1.3 Dataset

Transformations are applied:

Convert images to tensors.

Normalize pixel values for better learning.

Data is divided into batches of 64 images.

[5]:
```
batch_size = 64
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
 ↪1307,), (0.3081,))])
train_dataset = datasets.MNIST('./data', train=True, download=True,␣
 ↪transform=transform)
test_dataset  = datasets.MNIST('./data', train=False, download=True,␣
 ↪transform=transform)
train_loader  = torch.utils.data.DataLoader(train_dataset,␣
 ↪batch_size=batch_size, shuffle=True)
test_loader   = torch.utils.data.DataLoader(test_dataset,␣
 ↪batch_size=batch_size, shuffle=False)
```

set model, use adam optimizer and cross entropy loss

[6]:
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = KANNetwork().to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.CrossEntropyLoss()
```

Training and Testing fxn

[7]:
```
def train(model, device, train_loader, optimizer, criterion, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss   = criterion(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 100 == 0:
            print(f"Train Epoch {epoch} [{batch_idx*len(data)}/
 ↪{len(train_loader.dataset)}]  Loss: {loss.item():.6f}")

def test(model, device, test_loader, criterion):
    model.eval()
    test_loss = 0
    correct   = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += criterion(output, target).item() * data.size(0)
```

```
            pred = output.argmax(dim=1)
            correct += pred.eq(target).sum().item()
    test_loss /= len(test_loader.dataset)
    accuracy = 100. * correct / len(test_loader.dataset)
    print(f"Test set: Average loss: {test_loss:.4f}, Accuracy: {correct}/
↪{len(test_loader.dataset)} ({accuracy:.2f}%)")
    return accuracy
```

Main fxn

[8]:
```
num_epochs = 10
for epoch in range(1, num_epochs + 1):
    train(model, device, train_loader, optimizer, criterion, epoch)
    test(model, device, test_loader, criterion)
```

```
Train Epoch 1 [0/60000]   Loss: 2.456431
Train Epoch 1 [6400/60000]   Loss: 0.356771
Train Epoch 1 [12800/60000]   Loss: 0.325261
Train Epoch 1 [19200/60000]   Loss: 0.128849
Train Epoch 1 [25600/60000]   Loss: 0.338014
Train Epoch 1 [32000/60000]   Loss: 0.179312
Train Epoch 1 [38400/60000]   Loss: 0.116476
Train Epoch 1 [44800/60000]   Loss: 0.098487
Train Epoch 1 [51200/60000]   Loss: 0.175551
Train Epoch 1 [57600/60000]   Loss: 0.074753
Test set: Average loss: 0.1197, Accuracy: 9650/10000 (96.50%)
Train Epoch 2 [0/60000]   Loss: 0.124895
Train Epoch 2 [6400/60000]   Loss: 0.142201
Train Epoch 2 [12800/60000]   Loss: 0.186275
Train Epoch 2 [19200/60000]   Loss: 0.106183
Train Epoch 2 [25600/60000]   Loss: 0.094119
Train Epoch 2 [32000/60000]   Loss: 0.025040
Train Epoch 2 [38400/60000]   Loss: 0.175447
Train Epoch 2 [44800/60000]   Loss: 0.067399
Train Epoch 2 [51200/60000]   Loss: 0.050091
Train Epoch 2 [57600/60000]   Loss: 0.072920
Test set: Average loss: 0.0889, Accuracy: 9723/10000 (97.23%)
Train Epoch 3 [0/60000]   Loss: 0.064827
Train Epoch 3 [6400/60000]   Loss: 0.048971
Train Epoch 3 [12800/60000]   Loss: 0.033758
Train Epoch 3 [19200/60000]   Loss: 0.083907
Train Epoch 3 [25600/60000]   Loss: 0.179422
Train Epoch 3 [32000/60000]   Loss: 0.044245
Train Epoch 3 [38400/60000]   Loss: 0.086158
Train Epoch 3 [44800/60000]   Loss: 0.025418
Train Epoch 3 [51200/60000]   Loss: 0.020873
Train Epoch 3 [57600/60000]   Loss: 0.015032
Test set: Average loss: 0.0951, Accuracy: 9697/10000 (96.97%)
```

```
Train Epoch 4 [0/60000]  Loss: 0.032289
Train Epoch 4 [6400/60000]  Loss: 0.034286
Train Epoch 4 [12800/60000]  Loss: 0.043628
Train Epoch 4 [19200/60000]  Loss: 0.009331
Train Epoch 4 [25600/60000]  Loss: 0.023568
Train Epoch 4 [32000/60000]  Loss: 0.093715
Train Epoch 4 [38400/60000]  Loss: 0.093260
Train Epoch 4 [44800/60000]  Loss: 0.055522
Train Epoch 4 [51200/60000]  Loss: 0.084395
Train Epoch 4 [57600/60000]  Loss: 0.021980
Test set: Average loss: 0.0932, Accuracy: 9717/10000 (97.17%)
Train Epoch 5 [0/60000]  Loss: 0.050172
Train Epoch 5 [6400/60000]  Loss: 0.060684
Train Epoch 5 [12800/60000]  Loss: 0.111625
Train Epoch 5 [19200/60000]  Loss: 0.129068
Train Epoch 5 [25600/60000]  Loss: 0.030935
Train Epoch 5 [32000/60000]  Loss: 0.025470
Train Epoch 5 [38400/60000]  Loss: 0.092779
Train Epoch 5 [44800/60000]  Loss: 0.088442
Train Epoch 5 [51200/60000]  Loss: 0.077101
Train Epoch 5 [57600/60000]  Loss: 0.032946
Test set: Average loss: 0.0955, Accuracy: 9699/10000 (96.99%)
Train Epoch 6 [0/60000]  Loss: 0.019628
Train Epoch 6 [6400/60000]  Loss: 0.029890
Train Epoch 6 [12800/60000]  Loss: 0.029617
Train Epoch 6 [19200/60000]  Loss: 0.131167
Train Epoch 6 [25600/60000]  Loss: 0.031547
Train Epoch 6 [32000/60000]  Loss: 0.132707
Train Epoch 6 [38400/60000]  Loss: 0.020274
Train Epoch 6 [44800/60000]  Loss: 0.064796
Train Epoch 6 [51200/60000]  Loss: 0.038765
Train Epoch 6 [57600/60000]  Loss: 0.141061
Test set: Average loss: 0.0974, Accuracy: 9706/10000 (97.06%)
Train Epoch 7 [0/60000]  Loss: 0.036692
Train Epoch 7 [6400/60000]  Loss: 0.027578
Train Epoch 7 [12800/60000]  Loss: 0.040076
Train Epoch 7 [19200/60000]  Loss: 0.021739
Train Epoch 7 [25600/60000]  Loss: 0.070987
Train Epoch 7 [32000/60000]  Loss: 0.035413
Train Epoch 7 [38400/60000]  Loss: 0.078171
Train Epoch 7 [44800/60000]  Loss: 0.028670
Train Epoch 7 [51200/60000]  Loss: 0.034445
Train Epoch 7 [57600/60000]  Loss: 0.068992
Test set: Average loss: 0.1165, Accuracy: 9635/10000 (96.35%)
Train Epoch 8 [0/60000]  Loss: 0.081017
Train Epoch 8 [6400/60000]  Loss: 0.023539
Train Epoch 8 [12800/60000]  Loss: 0.056131
Train Epoch 8 [19200/60000]  Loss: 0.066282
```

```
Train Epoch 8 [25600/60000]  Loss: 0.038217
Train Epoch 8 [32000/60000]  Loss: 0.059021
Train Epoch 8 [38400/60000]  Loss: 0.121459
Train Epoch 8 [44800/60000]  Loss: 0.033824
Train Epoch 8 [51200/60000]  Loss: 0.037281
Train Epoch 8 [57600/60000]  Loss: 0.012281
Test set: Average loss: 0.1010, Accuracy: 9705/10000 (97.05%)
Train Epoch 9 [0/60000]  Loss: 0.014294
Train Epoch 9 [6400/60000]  Loss: 0.022332
Train Epoch 9 [12800/60000]  Loss: 0.050132
Train Epoch 9 [19200/60000]  Loss: 0.042760
Train Epoch 9 [25600/60000]  Loss: 0.033948
Train Epoch 9 [32000/60000]  Loss: 0.034645
Train Epoch 9 [38400/60000]  Loss: 0.071121
Train Epoch 9 [44800/60000]  Loss: 0.022558
Train Epoch 9 [51200/60000]  Loss: 0.041229
Train Epoch 9 [57600/60000]  Loss: 0.124367
Test set: Average loss: 0.1102, Accuracy: 9672/10000 (96.72%)
Train Epoch 10 [0/60000]  Loss: 0.004530
Train Epoch 10 [6400/60000]  Loss: 0.043787
Train Epoch 10 [12800/60000]  Loss: 0.044037
Train Epoch 10 [19200/60000]  Loss: 0.194782
Train Epoch 10 [25600/60000]  Loss: 0.117504
Train Epoch 10 [32000/60000]  Loss: 0.039180
Train Epoch 10 [38400/60000]  Loss: 0.032302
Train Epoch 10 [44800/60000]  Loss: 0.024973
Train Epoch 10 [51200/60000]  Loss: 0.081640
Train Epoch 10 [57600/60000]  Loss: 0.015299
Test set: Average loss: 0.1128, Accuracy: 9654/10000 (96.54%)
```

The best accuracy is 97.23 %

2 Quantum KAN

We can extend the classical KAN to quantum KAN to use quantum properties like superposition and entanglement that will make the execution fast.

2.1 Architecture

1. Encoding The input of pixel can be encoded to quantum state using angle encoding or amplitude encoding
2. Linear Transformation
3. Quantum Activation function Instead of using Bspline, we can use quantum activation fxn like VQC, QFT etc.
4. Convert the quantum layer output to classical output and then map the output

```
Classical Input (Flattened 784 MNIST pixels)
```

```
                             |
                             V
+-------------------------------------------------------+
|  Linear Transformation (Classical Fully Connected)  |
+-------------------------------------------------------+
                             |
                             V
+-----------------------------------------------------+
| Quantum Feature Encoding (Map Classical to Qubits)|
| Example: Angle Encoding:    =   * normalized(x)    |
+-----------------------------------------------------+
                             |
                             V
+---------------------------------------------------------+
| Quantum Activation Function (Parameterized Circuit)    |
| - Qubits undergo transformations via quantum gates     |
| - Nonlinearity introduced through quantum interference|
| - Measurement extracts classical feature values        |
+---------------------------------------------------------+
                             |
                             V
+-------------------------------------------------------+
|  Fully Connected Layer (Classical Readout)          |
|  - Maps quantum features to hidden neurons          |
+-------------------------------------------------------+
                             |
                             V
+------------------------------------+
|  Output Layer (Classical)        |
|  - 10 neurons for digit classes  |
+------------------------------------+
```

2.2 Advantages of QKAN over KAN

Quantum entanglement allows good fxn approximations

Few parameter are needed

Speed Increases

[13]:
```
import pennylane as qml
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms

# using 2 qubits
dev = qml.device("default.qubit", wires=2)
```

```
@qml.qnode(dev, interface="torch")
def quantum_activation(inputs, weights):
    qml.RX(inputs[0], wires=0)
    qml.RX(inputs[1], wires=1)
    qml.CNOT(wires=[0, 1])
    qml.RZ(weights[0], wires=0)
    qml.RZ(weights[1], wires=1)
    return [qml.expval(qml.PauliZ(i)) for i in range(2)]

class QKAN(nn.Module):
    def __init__(self):
        super(QKAN, self).__init__()
        self.fc1 = nn.Linear(784, 2)
        self.q_weights = nn.Parameter(torch.rand(2))
        self.fc2 = nn.Linear(2, 10)

    def forward(self, x):
        device = x.device
        x = torch.tanh(self.fc1(x))
        q_out = torch.stack([
            torch.tensor(quantum_activation(x[i].to("cpu"), self.q_weights.
 ↪to("cpu")), dtype=torch.float32).to(device)
            for i in range(x.shape[0])
        ])
        x = self.fc2(q_out)
        return x



# Load MNIST Dataset
batch_size = 64
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.
 ↪1307,), (0.3081,))])
train_loader = torch.utils.data.DataLoader(datasets.MNIST('./data', train=True,␣
 ↪download=True, transform=transform), batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(datasets.MNIST('./data', train=False,␣
 ↪download=True, transform=transform), batch_size=batch_size, shuffle=False)

# Initialize Model, Loss Function, and Optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = QKAN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)

def train(model, device, train_loader, optimizer, criterion, epoch):
    model.train()
```

```python
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        data = data.view(data.size(0), -1)  #flatten images

        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        if batch_idx % 100 == 0:
            print(f"Train Epoch {epoch} [{batch_idx*len(data)}/
↪{len(train_loader.dataset)}]  Loss: {loss.item():.6f}")


def test(model, device, test_loader, criterion):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            data = data.view(data.size(0), -1)
            output = model(data)
            test_loss += criterion(output, target).item() * data.size(0)
            pred = output.argmax(dim=1)
            correct += pred.eq(target).sum().item()

    test_loss /= len(test_loader.dataset)
    accuracy = 100. * correct / len(test_loader.dataset)
    print(f"Test set: Average loss: {test_loss:.4f}, Accuracy: {correct}/
↪{len(test_loader.dataset)} ({accuracy:.2f}%)")
    return accuracy

num_epochs = 5
for epoch in range(1, num_epochs + 1):
    train(model, device, train_loader, optimizer, criterion, epoch)
    test(model, device, test_loader, criterion)
```

```
Train Epoch 1 [0/60000]  Loss: 2.442454
Train Epoch 1 [6400/60000]  Loss: 2.376703
Train Epoch 1 [12800/60000]  Loss: 2.305700
Train Epoch 1 [19200/60000]  Loss: 2.292043
Train Epoch 1 [25600/60000]  Loss: 2.310628
Train Epoch 1 [32000/60000]  Loss: 2.311791
Train Epoch 1 [38400/60000]  Loss: 2.303499
Train Epoch 1 [44800/60000]  Loss: 2.283513
```

```
Train Epoch 1 [51200/60000]  Loss: 2.281255
Train Epoch 1 [57600/60000]  Loss: 2.294804
Test set: Average loss: 2.2967, Accuracy: 1066/10000 (10.66%)
Train Epoch 2 [0/60000]  Loss: 2.283395
Train Epoch 2 [6400/60000]  Loss: 2.289648
Train Epoch 2 [12800/60000]  Loss: 2.301621
Train Epoch 2 [19200/60000]  Loss: 2.304470
Train Epoch 2 [25600/60000]  Loss: 2.292690
Train Epoch 2 [32000/60000]  Loss: 2.278935
Train Epoch 2 [38400/60000]  Loss: 2.286330
Train Epoch 2 [44800/60000]  Loss: 2.302229
Train Epoch 2 [51200/60000]  Loss: 2.279379
Train Epoch 2 [57600/60000]  Loss: 2.284999
Test set: Average loss: 2.2911, Accuracy: 1100/10000 (11.00%)
Train Epoch 3 [0/60000]  Loss: 2.283823
Train Epoch 3 [6400/60000]  Loss: 2.277641
Train Epoch 3 [12800/60000]  Loss: 2.293749
Train Epoch 3 [19200/60000]  Loss: 2.284212
Train Epoch 3 [25600/60000]  Loss: 2.270528
Train Epoch 3 [32000/60000]  Loss: 2.294852
Train Epoch 3 [38400/60000]  Loss: 2.293701
Train Epoch 3 [44800/60000]  Loss: 2.305026
Train Epoch 3 [51200/60000]  Loss: 2.291771
Train Epoch 3 [57600/60000]  Loss: 2.302829
Test set: Average loss: 2.2856, Accuracy: 1343/10000 (13.43%)
Train Epoch 4 [0/60000]  Loss: 2.297766
Train Epoch 4 [6400/60000]  Loss: 2.293232
Train Epoch 4 [12800/60000]  Loss: 2.291041
Train Epoch 4 [19200/60000]  Loss: 2.280107
Train Epoch 4 [25600/60000]  Loss: 2.270379
Train Epoch 4 [32000/60000]  Loss: 2.283283
Train Epoch 4 [38400/60000]  Loss: 2.270339
Train Epoch 4 [44800/60000]  Loss: 2.274036
Train Epoch 4 [51200/60000]  Loss: 2.293747
Train Epoch 4 [57600/60000]  Loss: 2.281500
Test set: Average loss: 2.2804, Accuracy: 1509/10000 (15.09%)
Train Epoch 5 [0/60000]  Loss: 2.274093
Train Epoch 5 [6400/60000]  Loss: 2.272843
Train Epoch 5 [12800/60000]  Loss: 2.285500
Train Epoch 5 [19200/60000]  Loss: 2.268864
Train Epoch 5 [25600/60000]  Loss: 2.271374
Train Epoch 5 [32000/60000]  Loss: 2.283595
Train Epoch 5 [38400/60000]  Loss: 2.263728
Train Epoch 5 [44800/60000]  Loss: 2.282680
Train Epoch 5 [51200/60000]  Loss: 2.271480
Train Epoch 5 [57600/60000]  Loss: 2.267489
Test set: Average loss: 2.2755, Accuracy: 1580/10000 (15.80%)
```

This is a sample code of implementation QKAN, we can redefine model and fine tune it to get better accuracy