CODE(5 QUBIT 64 LAYER):

import torch

import torchreid

from torchvision import transforms

import pennylane as qml

# Define the quantum circuit using PennyLane

n\_qubits = 5

n\_layers = 64

dev = qml.device("default.qubit", wires=n\_qubits)

@qml.qnode(dev)

def qnode(inputs, weights):

    qml.AmplitudeEmbedding(inputs, wires=range(n\_qubits), pad\_with=0.0, normalize=True)

    qml.BasicEntanglerLayers(weights, wires=range(n\_qubits))

    return [qml.expval(qml.PauliZ(wires=i)) for i in range(n\_qubits)]

# Define the number of quantum layers dynamically

weight\_shapes = {"weights": (n\_layers, n\_qubits)}

class HybridReIDModel(torch.nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(HybridReIDModel, self).\_\_init\_\_()

        # Pre-trained ResNet50 for feature extraction

        self.backbone = torchreid.models.build\_model(

            name='resnet50',

            num\_classes=num\_classes,

            loss='softmax',

            pretrained=True,  # Using pre-trained weights

        ).cuda()

        # Freeze all layers except the final few (fine-tune deeper layers)

        for param in self.backbone.parameters():

            param.requires\_grad = False  # Freeze all layers initially

        # Unfreeze the last few layers (for example, the last block of ResNet50)

        for param in self.backbone.layer4.parameters():

            param.requires\_grad = True  # Unfreeze the deeper layers

        # Quantum layers

        self.qlayers = torch.nn.ModuleList([

            qml.qnn.TorchLayer(qnode, weight\_shapes) for \_ in range(n\_layers)

        ])

        # Fully connected layer for final classification

        self.fc = torch.nn.Linear(n\_layers \* n\_qubits, num\_classes)

    def forward(self, x):

        # Extract features

        features = self.backbone(x)  # Now, this will fine-tune based on the "requires\_grad"

        # Quantum layers and fully connected output

        quantum\_outputs = []

        part\_size = features.size(1) // n\_layers

        for i in range(n\_layers):

            start\_idx = i \* part\_size

            end\_idx = (i + 1) \* part\_size if i < n\_layers - 1 else None

            x\_part = features[:, start\_idx:end\_idx]

            quantum\_outputs.append(self.qlayers[i](x\_part))

        x = torch.cat(quantum\_outputs, dim=1)

        x = self.fc(x)

        return x

# Wrap execution code in main guard

if \_\_name\_\_ == '\_\_main\_\_':

    # Define transformations

    transform\_pipeline = [

        'random\_flip',       # Random horizontal flip

        'random\_rotate',     # Random rotation

        'random\_crop',       # Random resized crop

        'color\_jitter',      # Color jitter

        'normalize'          # Normalize (mean, std)

    ]

    # Create data manager for PRID2011

    datamanager = torchreid.data.VideoDataManager(

        root='',

        sources='prid2011',

        height=256,

        width=128,

        batch\_size\_train=8,

        batch\_size\_test=64,

        seq\_len=6,

        sample\_method='random',

        transforms=transform\_pipeline ,

        num\_instances=4,

        workers=8

    )

    # Access train and test loaders

    datamanager.train\_loader.num\_workers = 0  # Ensure compatibility with Windows

    train\_loader = datamanager.train\_loader

    test\_loader = datamanager.test\_loader

    query\_loader = test\_loader['prid2011']['query']

    gallery\_loader = test\_loader['prid2011']['gallery']

    # Get number of unique identities (classes)

    num\_classes = datamanager.num\_train\_pids

    # Instantiate hybrid model

    model = HybridReIDModel(num\_classes).cuda()

    # Build optimizer and scheduler

# Build optimizer and scheduler

    optimizer = torch.optim.Adam(

        model.parameters(),

        lr=0.0001,  # Smaller learning rate for fine-tuning

        weight\_decay=5e-4  # L2 regularization to prevent overfitting

    )

    # Scheduler for adjusting the learning rate

    scheduler = torch.optim.lr\_scheduler.CosineAnnealingLR(optimizer, T\_max=20)  # Gradual learning rate reduction

    # Create training engine

    engine = torchreid.engine.VideoSoftmaxEngine(

        datamanager,

        model,

        optimizer,

        scheduler=scheduler,

        pooling\_method='avg',

        use\_gpu=True

    )

    engine.run(

        max\_epoch=30,

        save\_dir='log/hybrid\_resnet50\_dynamic\_layers7-16(3)',

        print\_freq=1,

        test\_only=False,

        eval\_freq=5

    )

RESULTS:  
EPOCH 5:

Speed: 10.3442 sec/batch

Computing distance matrix with metric=euclidean ...

Computing CMC and mAP ...

\*\* Results \*\*

mAP: 64.5%

CMC curve

Rank-1 : 52.8%

Rank-5 : 75.3%

Rank-10 : 88.8%

Rank-20 : 93.3%

EPOCH 10:

Done, obtained 89-by-89 matrix

Speed: 18.5080 sec/batch

Computing distance matrix with metric=euclidean ...

Computing CMC and mAP ...

\*\* Results \*\*

mAP: 73.3%

CMC curve

Rank-1 : 65.2%

Rank-5 : 85.4%

Rank-10 : 91.0%

Rank-20 : 95.5%

EPOCH 15:

Speed: 10.0631 sec/batch

Computing distance matrix with metric=euclidean ...

Computing CMC and mAP ...

\*\* Results \*\*

mAP: 77.6%

CMC curve

Rank-1 : 69.7%

Rank-5 : 88.8%

Rank-10 : 92.1%

Rank-20 : 96.6%

EPOCH 20:

Speed: 9.9684 sec/batch

Computing distance matrix with metric=euclidean ...

Computing CMC and mAP ...

\*\* Results \*\*

mAP: 77.1%

CMC curve

Rank-1 : 68.5%

Rank-5 : 91.0%

Rank-10 : 95.5%

Rank-20 : 100.0%

EPOCH 25:

Speed: 9.7694 sec/batch

Computing distance matrix with metric=euclidean ...

Computing CMC and mAP ...

\*\* Results \*\*

mAP: 76.0%

CMC curve

Rank-1 : 65.2%

Rank-5 : 89.9%

Rank-10 : 95.5%

Rank-20 : 100.0%

EPOCH 30:

Speed: 9.6567 sec/batch

Computing distance matrix with metric=euclidean ...

Computing CMC and mAP ...

\*\* Results \*\*

mAP: 77.9%

CMC curve

Rank-1 : 70.3%

Rank-5 : 91.3%

Rank-10 : 96.0%

Rank-20 : 100.0%