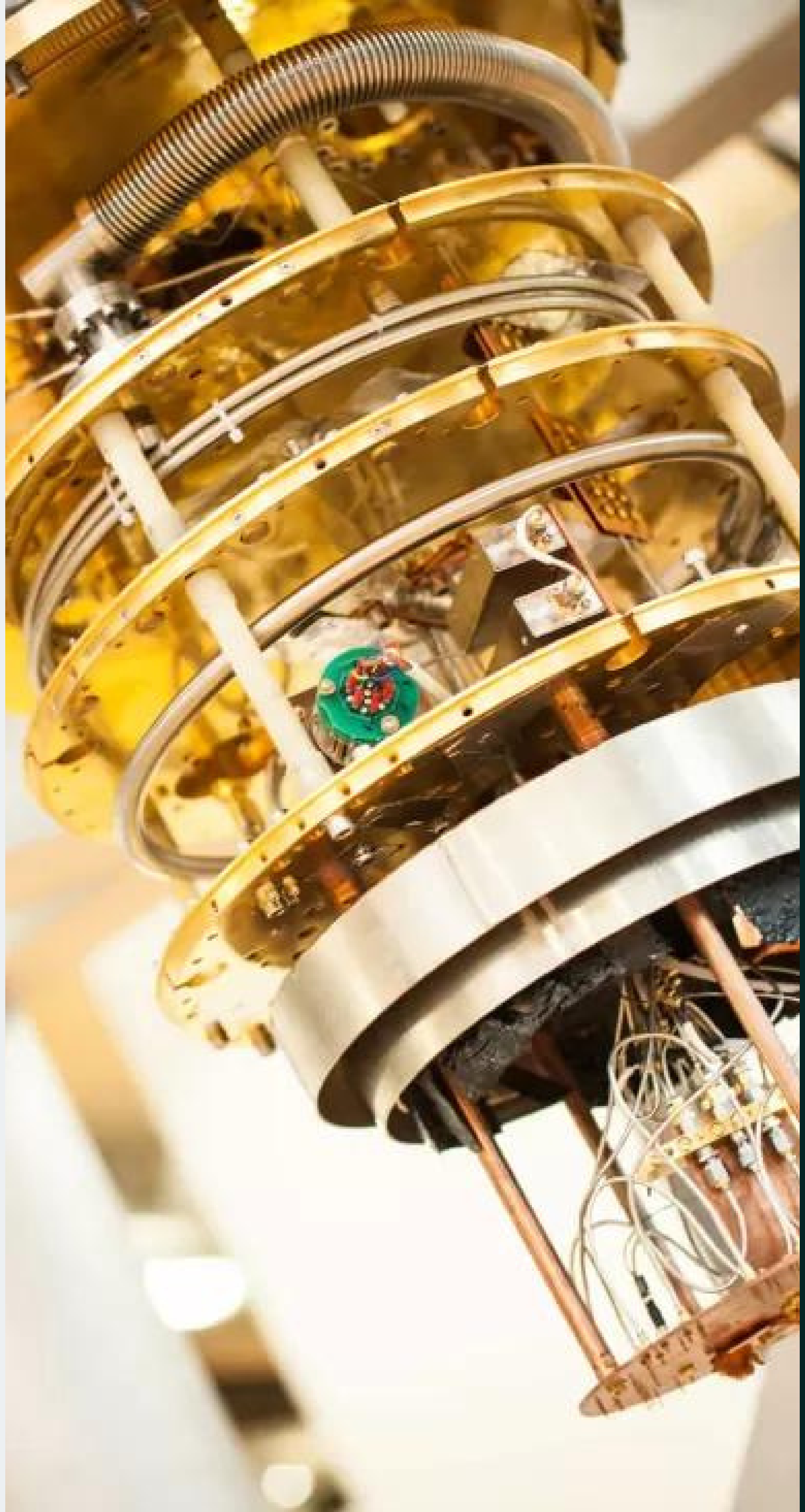
Project Presentation



QCourse 551-1

Project 27: QNN Algorithm Implementation

Mentor: Tal Michaeli

Team Members: Ashmit Gupta
Asif Saad
Roman Ledenov

Implementing a Hybrid Network to classify the MNIST Dataset

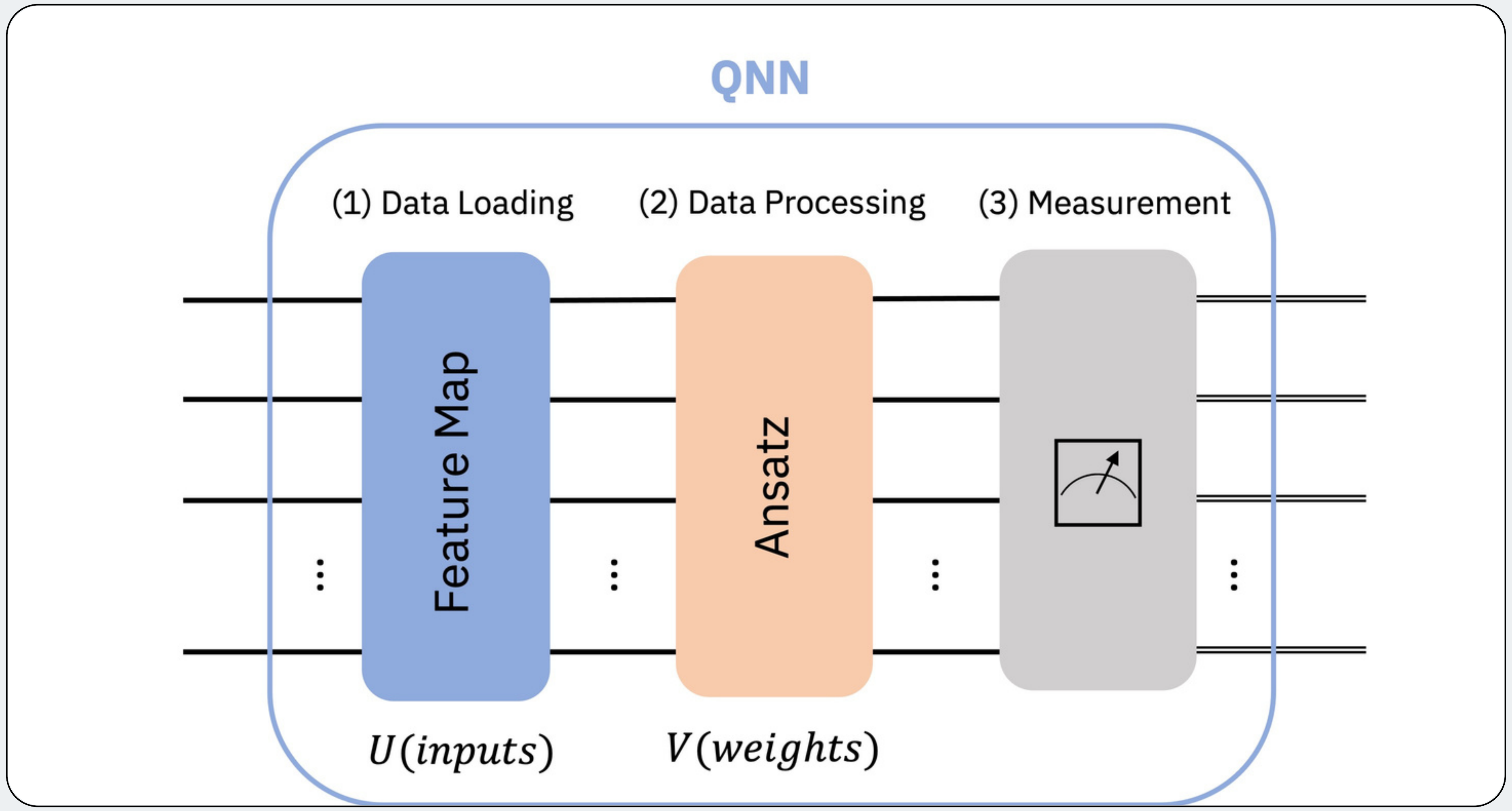
Agenda

Quantum Neural Network

Quantum Neural Networks are quantum algorithms based on parametrized quantum circuits that can be trained in a variational manner using classical optimizers.

These algorithmic models can be trained to find hidden patterns in data similar to their classical counterparts. These models can load classical data (inputs) into a quantum state, and later process it with quantum gates parametrized by trainable weights.

A Quantum layer take in classical data and return classical data.



Quantum Neural Network Algorithm Implementation

Problem Statement: Create a hybrid network consisting of both classical and quantum layers to classify the MNIST dataset.

Project Milestones

WE ARE HERE!

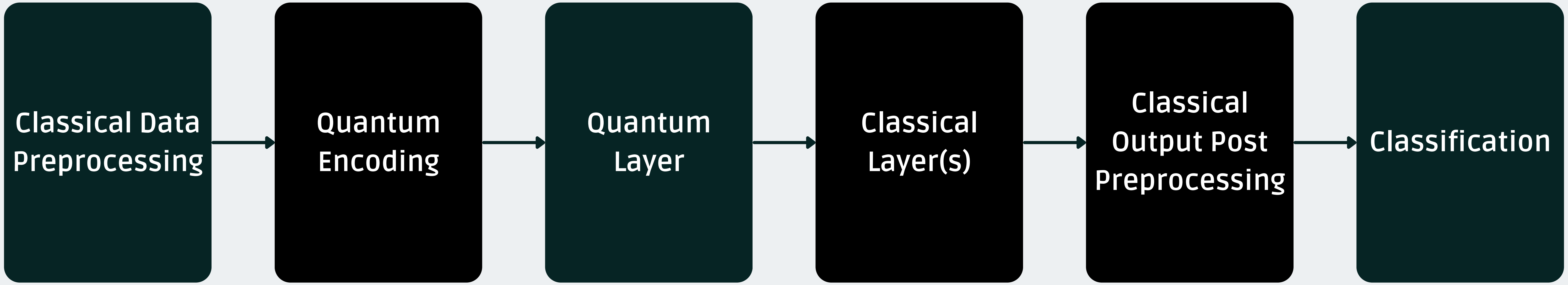
01	02	03	04	05	06
WEEK 1-3	**WEEK 4-5**	**WEEK 6-7**	**WEEK 8-9**	**WEEK 10-11**	**WEEK 12**
Discuss project, understanding the prerequisites and creating a work plan.	Pre-processing MNIST dataset and encoding it into a Quantum Circuit.	Design and Implement circuit for Quantum Layer.	Training and Testing of our QNN,	Extend our research and start writing our project research paper.	Finishing up and final presentation.

Re-evaluating QNN Architecture

This Hybrid QNN architecture combines classical and quantum processing for efficient MNIST digit classification. Classical data preprocessing is followed by quantum encoding, where classical information is translated into quantum states.

The Quantum Layer utilizes quantum gates to perform computations, followed by Classical Layer(s) for further processing. The Quantum/Classical interfacing or pooling layer integrates quantum and classical information, leading to classical output post-processing. Finally, the classification step provides the ultimate prediction.

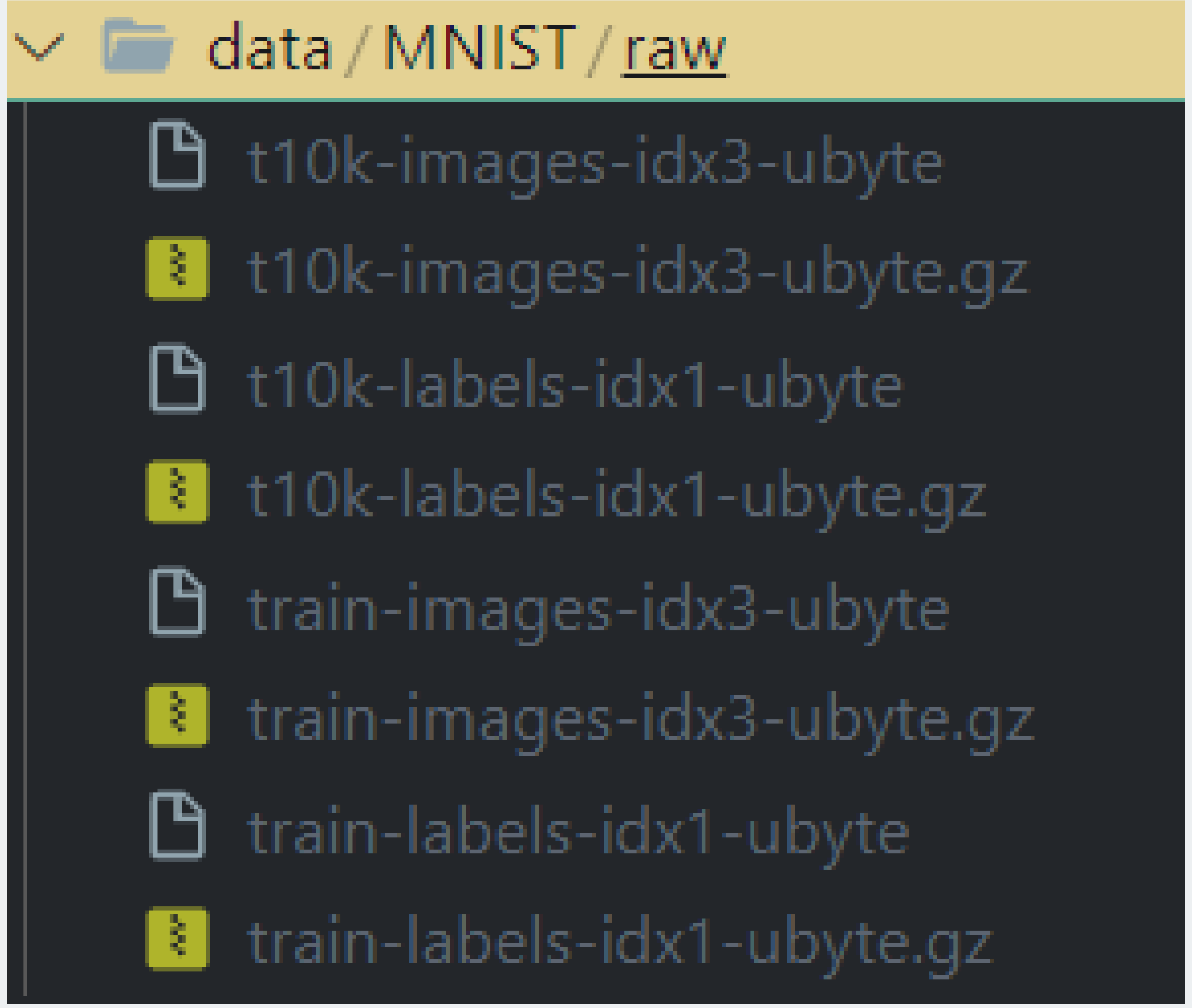
This hybrid approach leverages the strengths of quantum computing while maintaining compatibility with classical methods, offering a promising paradigm for enhanced machine learning tasks on quantum devices.



Getting MNIST Data Set

We downloaded the MNIST Dataset from torchvision.datasets which is a 60000 training datasets and 10000 test dataset images of handwritten digits, each of 28x28 px.

```
# Setup training data
train_data = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=input_transform,
    target_transform=target_transform
)

test_data = datasets.MNIST(
    root="data",
    train=False,
    download=True,
    transform=input_transform,
    target_transform=target_transform
)
```



Pre-processing the MNIST Dataset

The input MNIST images are all 28 × 28. In this step, we will firstly center-crop input images to 24 × 24 and then down-sample them to 4 × 4 for MNIST. Then we convert the image pixels into angles for passing them into Rotation gates later for encoding.

```
def input_transform(image):
    """
    The input MNIST images are all 28 × 28. This function will firstly center-crop
    them to 24 × 24 and then down-sample them to 4 × 4 for MNIST. Then we convert
    the image pixels into angles for passing them into Rotation gates later for encoding.
    """
    image = transforms.ToTensor()(image)
    image = transforms.CenterCrop(24)(image)
    image = transforms.Resize(size = (4,4))(image)
    image = image.squeeze()
    image_pixels = torch.flatten(image)
    angles = torch.sqrt(image_pixels / 256)

    return angles
```

Input Image:

```
<PIL.Image.Image image mode=L size=28x28 at 0x7F8BDACBB750>
```

```
(tensor([0.0000, 0.0000, 0.0317, 0.0378, 0.0000, 0.0336, 0.0000, 0.0000, 0.0000,
         0.0000, 0.0477, 0.0000, 0.0295, 0.0620, 0.0000, 0.0000]),
```

Output : Array

Preparing Data loader

The data loader converts our dataset into python iterables of mini-batches having size 32.

```
# Setup the batch size hyperparameter
BATCH_SIZE = 32

# Turn datasets into iterables (batches)
train_dataloader = DataLoader(train_data,
    batch_size=BATCH_SIZE,
    shuffle=True
)

test_dataloader = DataLoader(test_data,
    batch_size=BATCH_SIZE,
    shuffle=False
)

# Let's check out what we've created
print(f"Dataloaders: {train_dataloader, test_dataloader}")
print(f"Length of train dataloader: {len(train_dataloader)} batches of {BATCH_SIZE}")
print(f"Length of test dataloader: {len(test_dataloader)} batches of {BATCH_SIZE}")
```

```
Dataloaders: (<torch.utils.data.dataloader.DataLoader object at 0x7f8be9382e90>, <torch.utils.data.dataloader.DataLoader object at 0x7f8be8fac990>)
Length of train dataloader: 1875 batches of 32
Length of test dataloader: 313 batches of 32
```

Quantum Function for encoding image pixels

We encode 16 pixel image such as 4 px per qubit by using different rotation gates on each qubit.

```python
@QFunc
def encoding(q: QArray[QBit]) -> None:
    """
    This function encodes the input data into the qubits. This input data is a 4x4 image pixel values
    converted into angle for rotation gates (RX, RY, RZ, RX) in form of a 16x1 vector.
    We encode 4 pixels per qubit.

    Args:
        q (QArray[QBit]): Array of four Qubits to encode the input data into.
    """
    RX(theta="input_0", target=q[0]) # Pixel 0 on Qubit 0
    RY(theta="input_1", target=q[0]) # Pixel 1 on Qubit 0
    RZ(theta="input_2", target=q[0]) # Pixel 2 on Qubit 0
    RX(theta="input_3", target=q[0]) # Pixel 3 on Qubit 0

    RX(theta="input_4", target=q[1]) # Pixel 4 on Qubit 1
    RY(theta="input_5", target=q[1]) # Pixel 5 on Qubit 1
    RZ(theta="input_6", target=q[1]) # Pixel 6 on Qubit 1
    RX(theta="input_7", target=q[1]) # Pixel 7 on Qubit 1

    RX(theta="input_8", target=q[2]) # Pixel 8 on Qubit 2
    RY(theta="input_9", target=q[2]) # Pixel 9 on Qubit 2
    RZ(theta="input_10", target=q[2]) # Pixel 10 on Qubit 2
    RX(theta="input_11", target=q[2]) # Pixel 11 on Qubit 2

    RX(theta="input_12", target=q[3]) # Pixel 12 on Qubit 3
    RY(theta="input_13", target=q[3]) # Pixel 13 on Qubit 3
    RZ(theta="input_14", target=q[3]) # Pixel 14 on Qubit 3
    RX(theta="input_15", target=q[3]) # Pixel 15 on Qubit 3
```

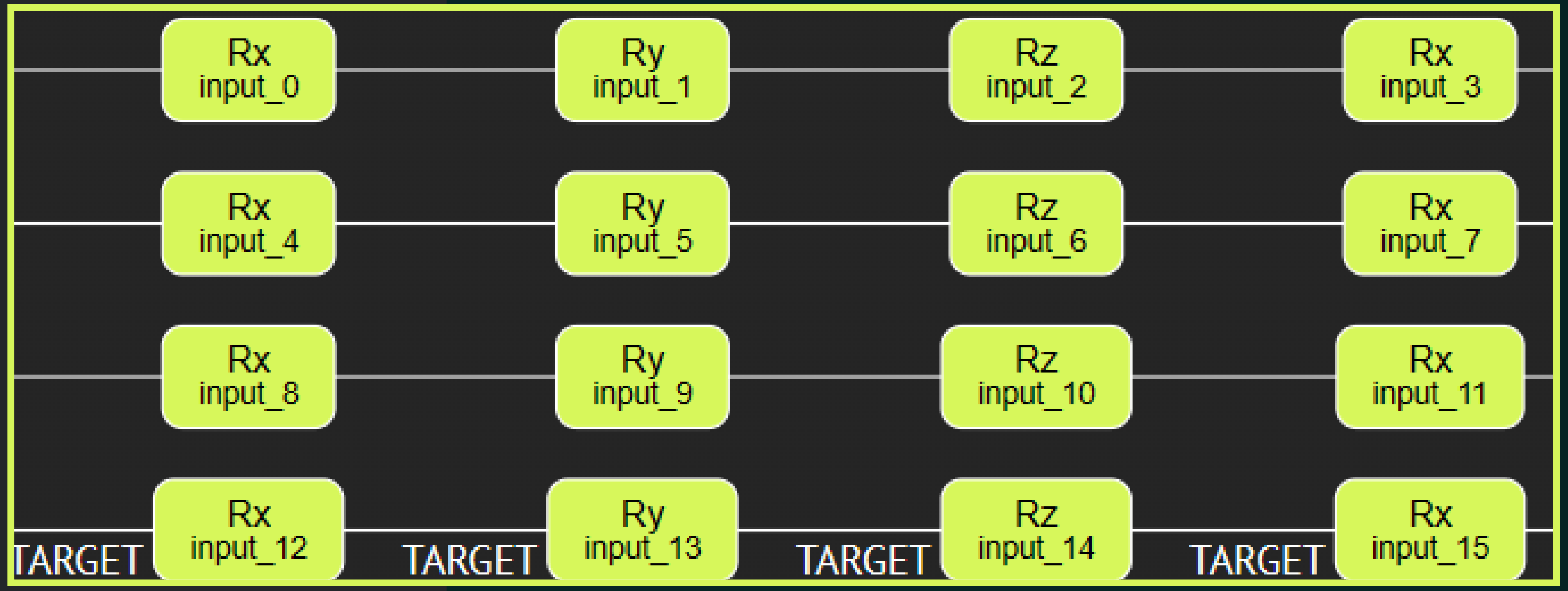


Fig: Snapshot of a quantum circuit in classiq fot encoding image dataset into 4 qubits.

Quantum Function for mixing

After encoding, we performs the mixing operation on these four qubits such that:

(i) RZZ layer: add RZZ gates to all logical adjacent wires and the logical farthest wires to form a ring connection, for example, an RZZ layer in a 4-qubit circuit contains 4 RZZ gates which lie on wires 1 and 2, 2 and 3, 3 and 4, 4 and 1;

(ii) RXX layer: same structure as in RZZ layer;

(iii) RZX layer: same structure as in RZZ layer;

```
@QFunc
def mixing(q: QArray[QBit]) -> None:
    """
    This function performs the mixing operation on the qubits.
    This is done by applying a series of RZZ, RXX, RYY gates to form a
    ring connection.

    Args:
        q (QArray[QBit]): Array of four Qubits to apply the mixing operation on.
    """
    RZZ(theta="weight_0", target=q[0:2])
    RZZ(theta="weight_1", target=q[1:3])
    RZZ(theta="weight_2", target=q[2:4])

    RXX(theta="weight_4", target=q[0:2])
    RXX(theta="weight_5", target=q[1:3])
    RXX(theta="weight_6", target=q[2:4])

    RYY(theta="weight_8", target=q[0:2])
    RYY(theta="weight_9", target=q[1:3])
    RYY(theta="weight_10", target=q[2:4])
```

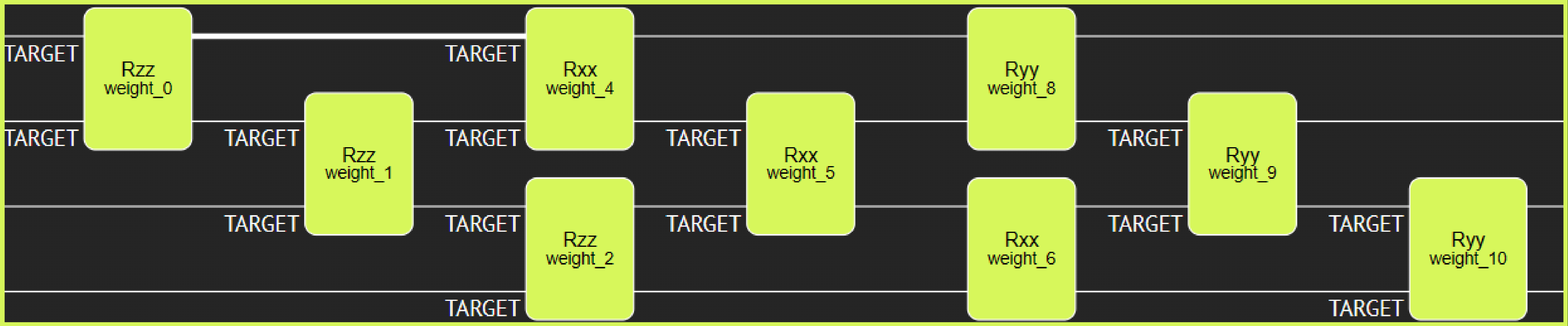


Fig: Snapshot of a quantum circuit in classiq for Mixing.

Quantum Function for CZ Block

In this block we add CZ gates to all logical adjacent wires.

```python
@QFunc
def cz_block(q: QArray[QBit]) -> None:
    """
    This function applies CZ gates between each qubit.

    Args:
        q (QArray[QBit]): Array of four Qubits to apply the entanglement operation on.
    """
    CZ(control=q[0], target=q[1])
    CZ(control=q[1], target=q[2])
    CZ(control=q[2], target=q[3])
```

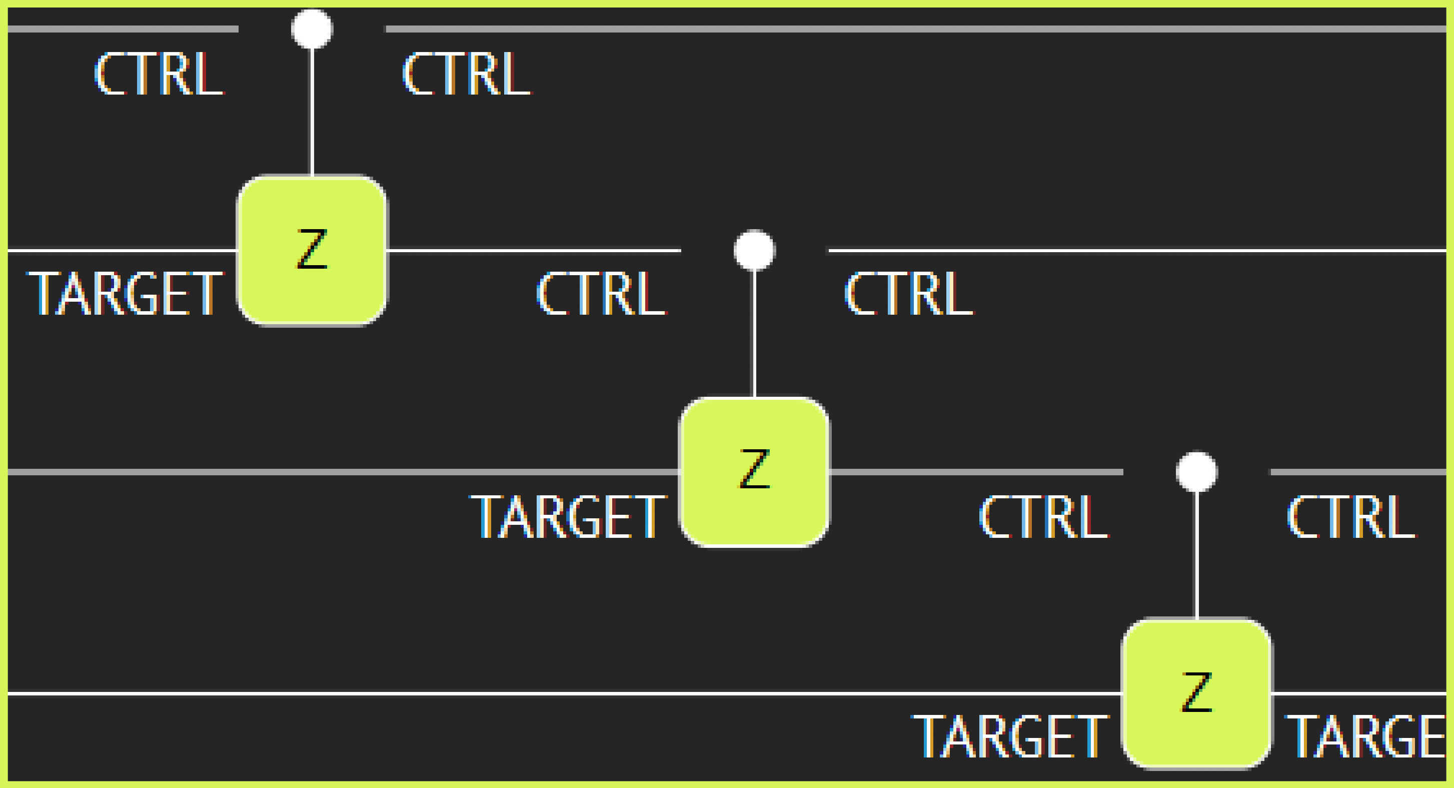


Fig: Snapshot of a quantum circuit in classiq for CZ-Block.

Overall Circuit for Quantum Model

The overall quantum model for the quantum neural network for classification of MNIST Dataset compromises of all three parts discussed above: Encoding, Mixing, and CZ Block

```
@QFunc
def main(res: Output[QArray[QBit]]) -> None:
    """
    This is the main function from which model will be created.
    It calls the other functions to perform the encoding, mixing and entanglement.

    Args:
        res (Output[QArray[QBit]]): Output QArray of QBits from which the model will be created.
    """
    allocate(4, res)
    encoding(q=res)
    mixing(q=res)
    cz_block(q=res)

# Create a model
model = create_model(main)
quantum_program = synthesize(model)
show(quantum_program)
```

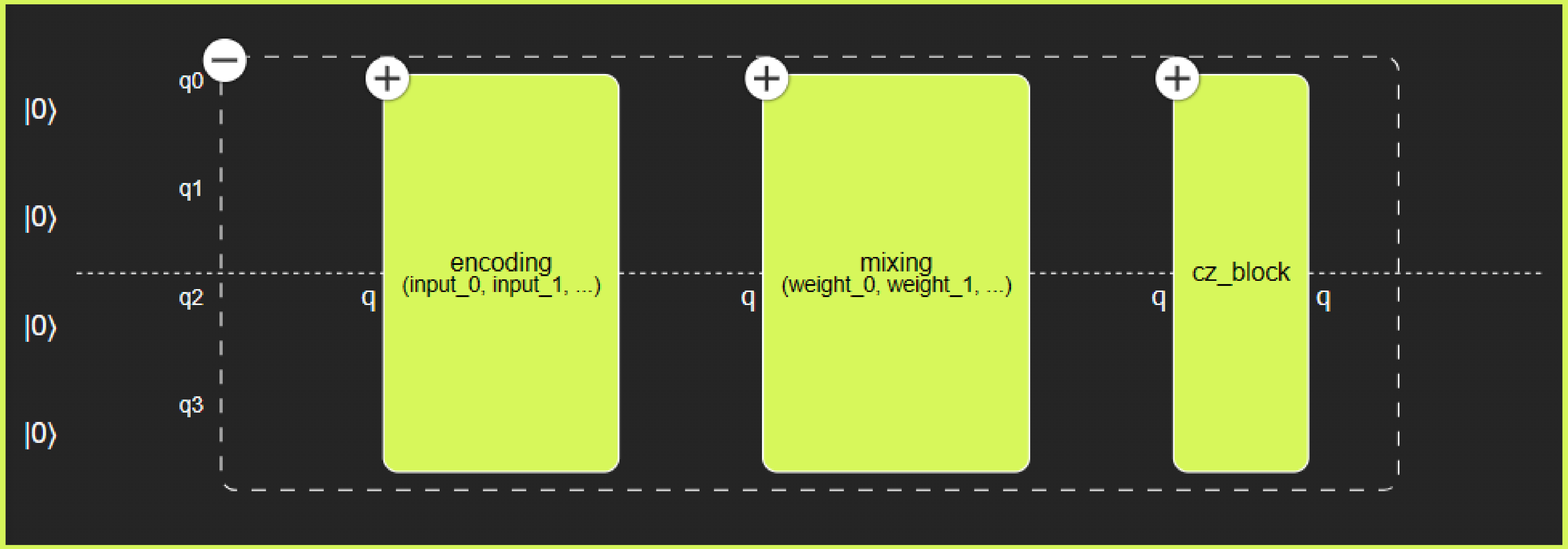


Fig: Snapshot of overall circuit of Quantum Layer

Quantum Neural Network

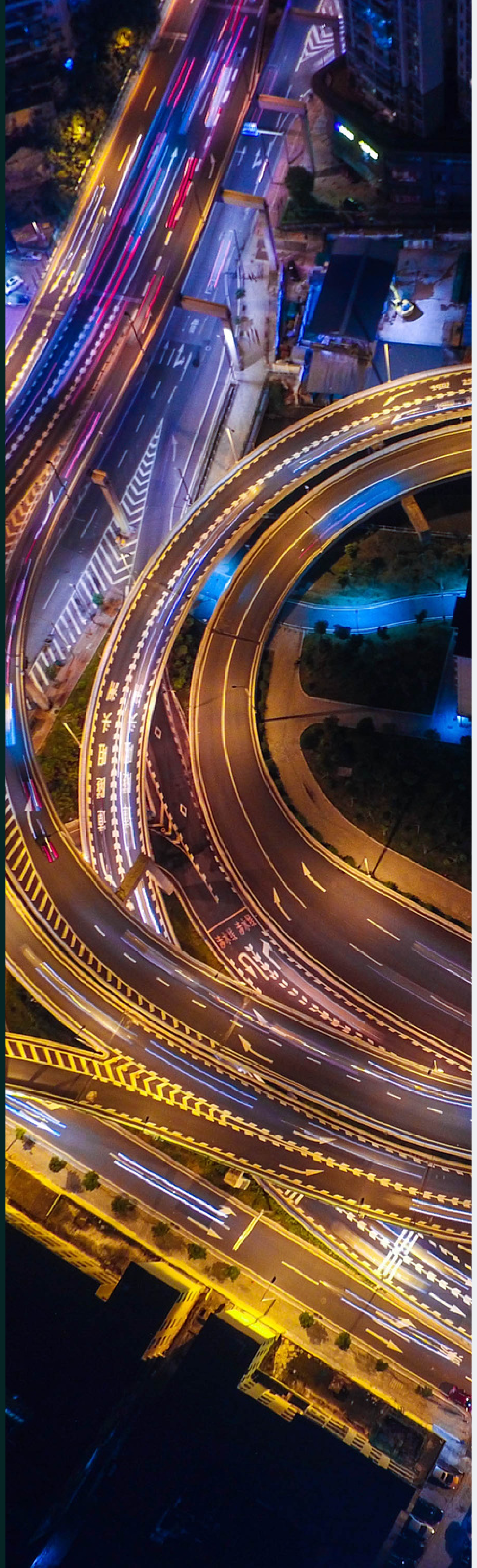
```python
def execute(quantum_program: SerializedQuantumProgram, arguments: MultipleArguments) -> ResultsCollection:
    return execute_qnn(quantum_program, arguments)

def post_process(result: SavedResult) -> torch.Tensor:
    counts: dict = result.value.counts
    # The probability of measuring |0>
    print(f"counts: {counts}")
    p_zero: float = counts.get("0", 0.0) / sum(counts.values())
    return torch.tensor(p_zero)

class Net(torch.nn.Module):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__()
        self.qlayer = QLayer(
            quantum_program,
            execute,
            post_process,
            *args,
            **kwargs
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.qlayer(x)
        return x

qnn = Net()
```

Loss Function, Optimizer, Training & Testing

```python
_LEARNING_RATE = 1.0

# choosing our loss function
loss_fn = nn.L1Loss()
# choosing our optimizer
optimizer = optim.SGD(qnn.parameters(), lr=_LEARNING_RATE)

def train(
    model: nn.Module,
    data_loader: DataLoader,
    loss_fn: nn.modules.loss._Loss,
    optimizer: optim.Optimizer,
    epochs: int = 20,
) -> None:
    for epoch in tqdm(range(epochs)):
        print(f"Epoch: {epoch}\n----------")
        for data, label in data_loader:
            optimizer.zero_grad()
            output = model(data)
            loss = loss_fn(output, label)
            loss.backward()
            optimizer.step()

train(model, train_dataloader, loss_fn, optimizer)
```

```python
def test(
    model: nn.Module,
    data_loader: DataLoader,
    atol=1e-4
) -> float:
    num_correct = 0
    total = 0

    # Put the model in eval mode
    model.eval()

    # Turn on inference mode context manager
    with torch.inference_mode():
        for data, labels in data_loader:
            # Let the model predict
            predictions = model(data)

            # Get a tensor of booleans, indicating if each label is close to the real label
            is_prediction_correct = predictions.isclose(labels, atol=atol)

            # Count the amount of `True` predictions
            num_correct += is_prediction_correct.sum().item()

            # Count the total evaluations
            #   the first dimension of `labels` is `batch_size`
            total += labels.size(0)

    # Calculate the accuracy
    accuracy = float(num_correct) / float(total)
    print(f"Test Accuracy of the model: {accuracy*100:.2f}")
    return accuracy

test(model, test_dataloader)
```

Future Plans



- Deciding and implementing Post-Processing Algorithm
- Start Training
- Optimization of our Quantum Neural Network
- Writing Project Paper
- Decreasing number of qubits

Resources

- Paper on Quantum On-Chip Training with Parameter Shift and Gradient Pruning
- Classiq QNN User Guide: https://docs.classiq.io/latest/user-guide/built-in-algorithms/qml/qnn/
- Project Repository on GitHub

Thank you!

Do you have any questions?