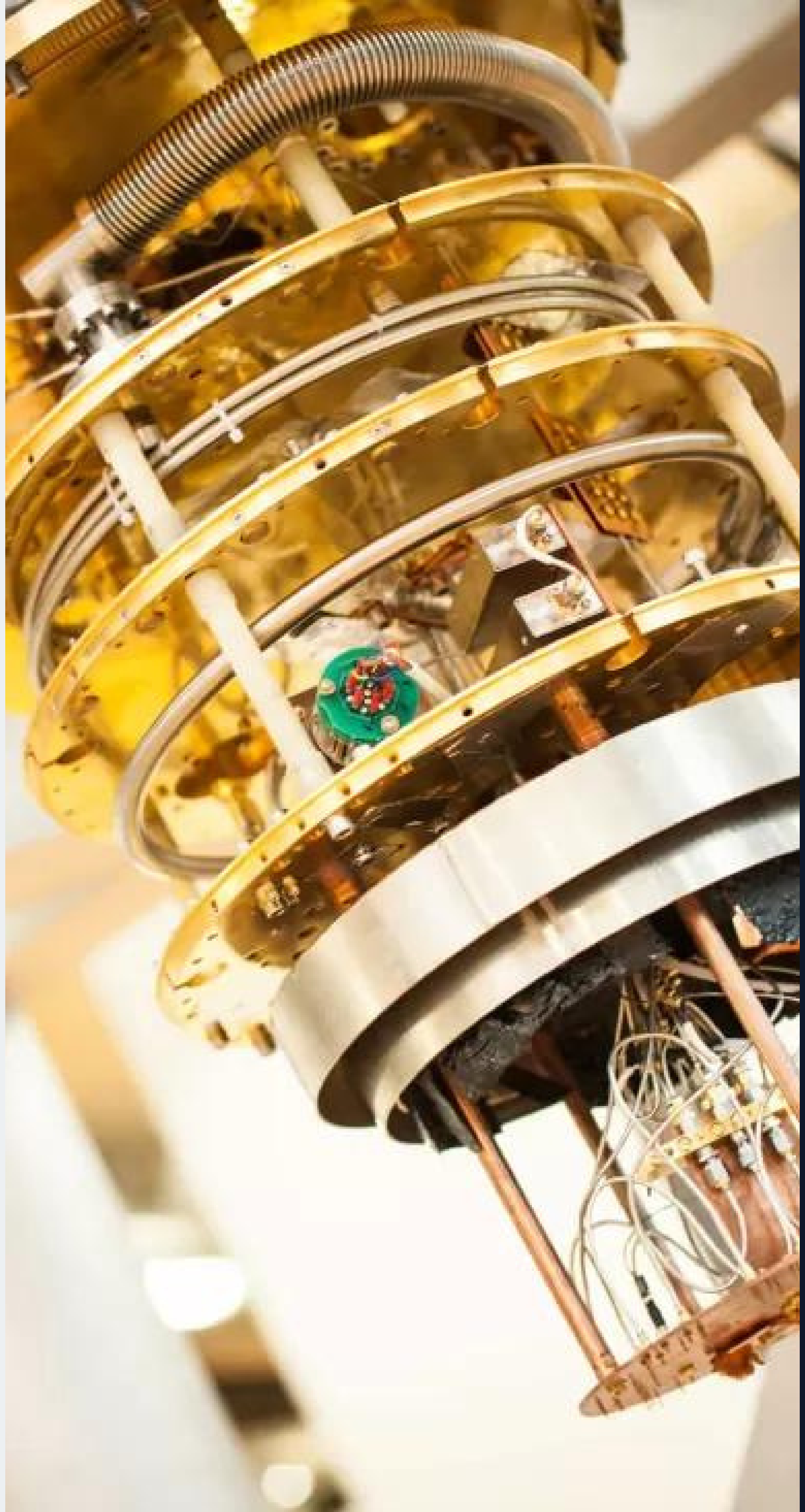
Project Presentation



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

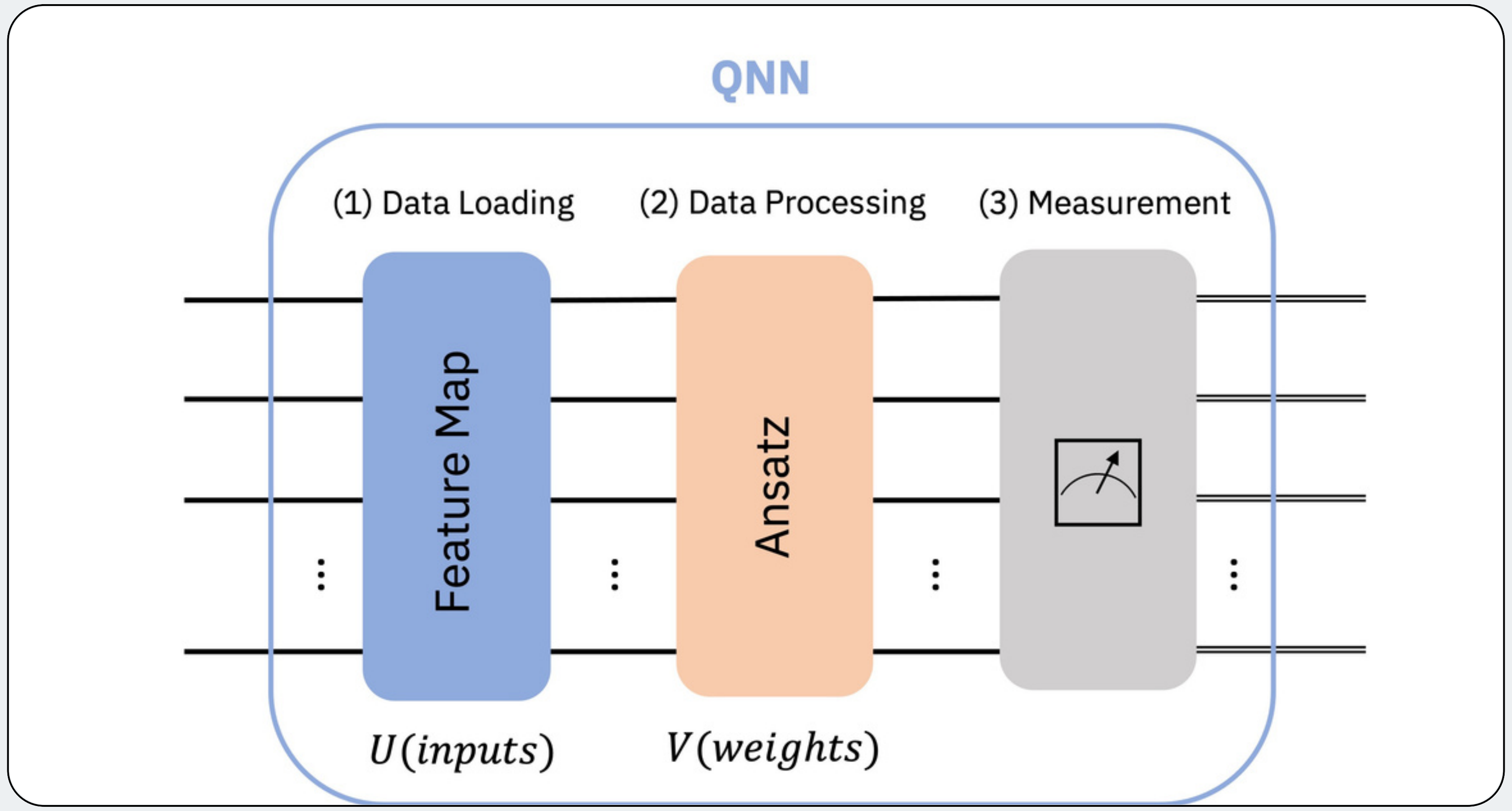
- Introduction to QNN
- Problem Statement
- Project Milestone
- Re-evaluating QNN Architecture
- Getting MNIST Dataset
- Preprocessing MNIST Datasets
- Preparing Data loader
- Quantum Function for encoding image pixels
- Quantum Function for mixing
- Quantum Function for CZ Block
- Overall Circuit for Quantum Model
- Quantum Neural Network
- Loss Function, Optimizer, Training & Testing

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.



Classical Data Preprocessing → Quantum Encoding → Quantum Entanglement using multi-qubit rotation gates → Quantum Entanglement using CZ-Block → Classical Output Post processing → Classification

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, make our project code modular, and start writing our project research paper.	Finishing up and final presentation.

Getting MNIST Data Set

I have created two different functions:

- To download and use MNIST dataset provided by PyTorch (torchvision.datasets) constuting classes from 0 to 9 of any subset size.
- To use custom binary dataset.

```python
def create_dataloaders_from_folders(
    train_dir: str,
    test_dir: str,
    batch_size: int,
    transform: Optional[Callable] = None,
    target_transform: Optional[Callable] = None,
    num_workers: int = NUM_WORKERS
):
    """Creates training and testing DataLoaders. ···
    # Use ImageFolder to create dataset(s)
    train_data = datasets.ImageFolder(train_dir, transform=transform, target_transform=target_transform)
    test_data = datasets.ImageFolder(test_dir, transform=transform, target_transform=target_transform)

    # Get class names      devilkiller-ag, 5 days ago • made everything modular for easier experimenta…
    class_names = train_data.classes

    # Turn images into data loaders
    train_dataloader = DataLoader(
        train_data,
        batch_size=batch_size,
        shuffle=True,
        num_workers=num_workers,
        pin_memory=True,
    )
    test_dataloader = DataLoader(
        test_data,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers,
        pin_memory=True,
    )

    return train_dataloader, test_dataloader, class_names
```

```
data \ MNIST \ raw
    t10k-images-idx3-ubyte
    t10k-images-idx3-ubyte.gz
    t10k-labels-idx1-ubyte
    t10k-labels-idx1-ubyte.gz
    train-images-idx3-ubyte
    train-images-idx3-ubyte.gz
    train-labels-idx1-ubyte
    train-labels-idx1-ubyte.gz
mini_data_128_bin
    test
        0
        1
    train
```

```python
def create_mnist_dataloaders(
    batch_size: int,
    root: str = "data",
    transform: Optional[Callable] = None,
    target_transform: Optional[Callable] = None,
    num_workers: int = NUM_WORKERS,
    create_subset: bool = False,
    subset_size:int = 64
):
    """Creates training and testing DataLoaders. ···
    # Setup training data
    train_data = datasets.MNIST(
        root=root, train=True, download=True,
        transform=transform, target_transform=target_transform
    )

    # Setup testing data
    test_data = datasets.MNIST(
        root=root, train=False, download=True,
        transform=transform, target_transform=target_transform
    )
          devilkiller-ag, 3 weeks ago • made everything modular for easier experimenta…
    # Get class names
    class_names = train_data.classes

    # Create subsets of the datasets
    if create_subset:
        train_data = Subset(train_data, range(subset_size))
        test_data = Subset(test_data, range(subset_size))

    # Turn datasets into iterables (batches)
    train_dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True, num_workers=num_workers,)
    test_dataloader = DataLoader(test_data, batch_size=batch_size, shuffle=False, num_workers=num_workers,)

    return train_dataloader, test_dataloader, class_names
```

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.

```python
def input_transform(image):
    """
    The input MNIST images are all 28 × 28 px. 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.Grayscale(num_output_channels=1)(image)
    image = transforms.ToTensor()(image)
    image = transforms.CenterCrop(24)(image)
    image = transforms.Resize(size = (4,4), antialias=True)(image)
    image = image.squeeze()
    image_pixels = torch.flatten(image)
    angles = torch.sqrt(image_pixels / 256)

    return angles

def target_transform(label):
    """
    Transforms labels into their one-hot encoded format of size 10.
    Ex.: 2 -> [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
    """
    label_tensor = torch.LongTensor([label])
    one_hot_label = torch.nn.functional.one_hot(label_tensor, 10)
    return one_hot_label.squeeze()

def target_transform_bin(label):
    """
    Transforms labels into their one-hot encoded format of size 2.
    Ex.: 0 -> [1, 0]
    """
    label_tensor = torch.LongTensor([label])
    one_hot_label = torch.nn.functional.one_hot(label_tensor, 2)
    return one_hot_label.squeeze()
```

Input Image:

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

Output Array:

```
(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]),
```

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
```

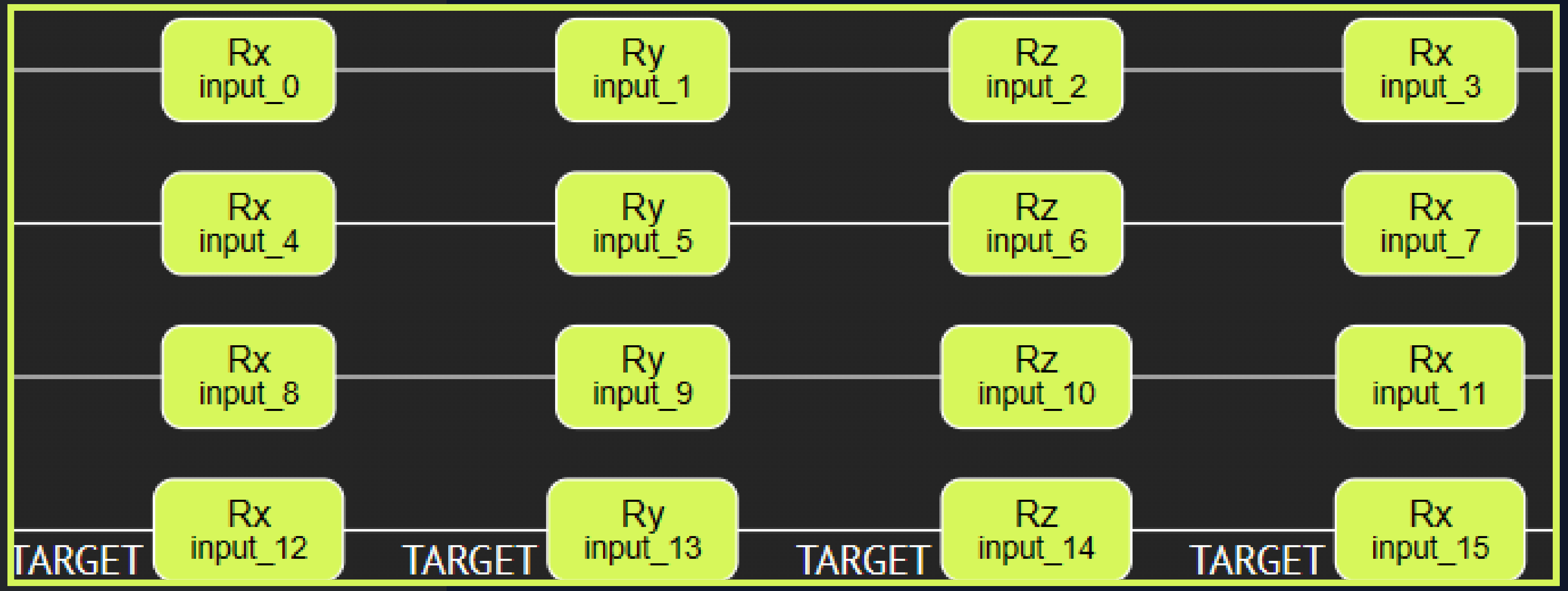


Fig: Snapshot of a quantum circuit in classiq fot encoding image dataset into 4 qubits.

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])
```

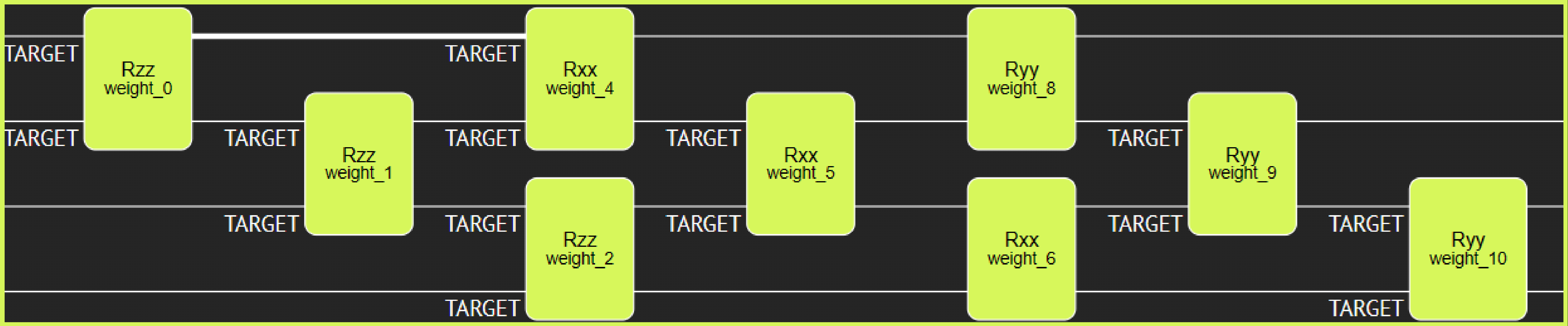


Fig: Snapshot of a quantum circuit in classiq for Mixing.

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])
```

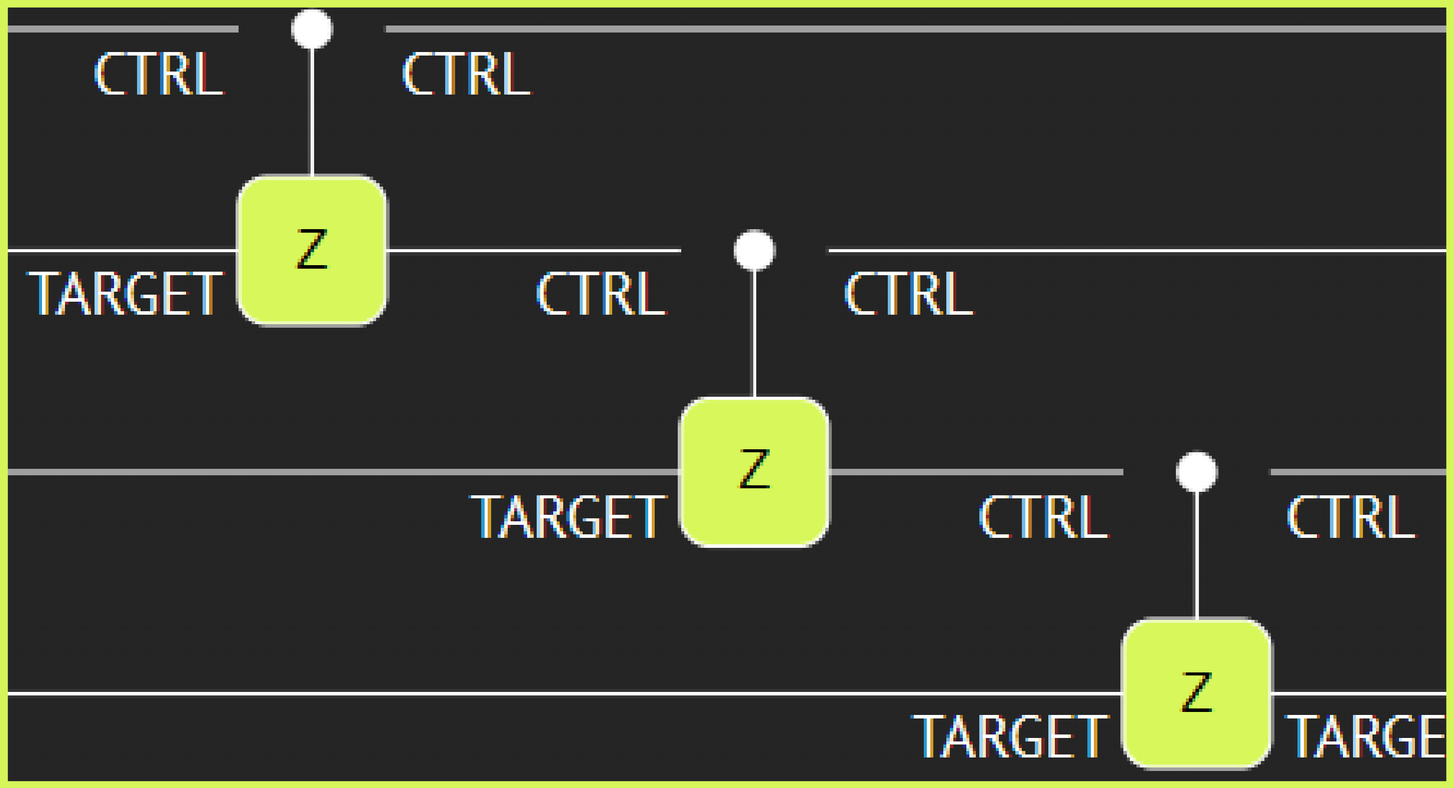


Fig: Snapshot of a quantum circuit in classiq for CZ-Block.

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

```python
@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)
```

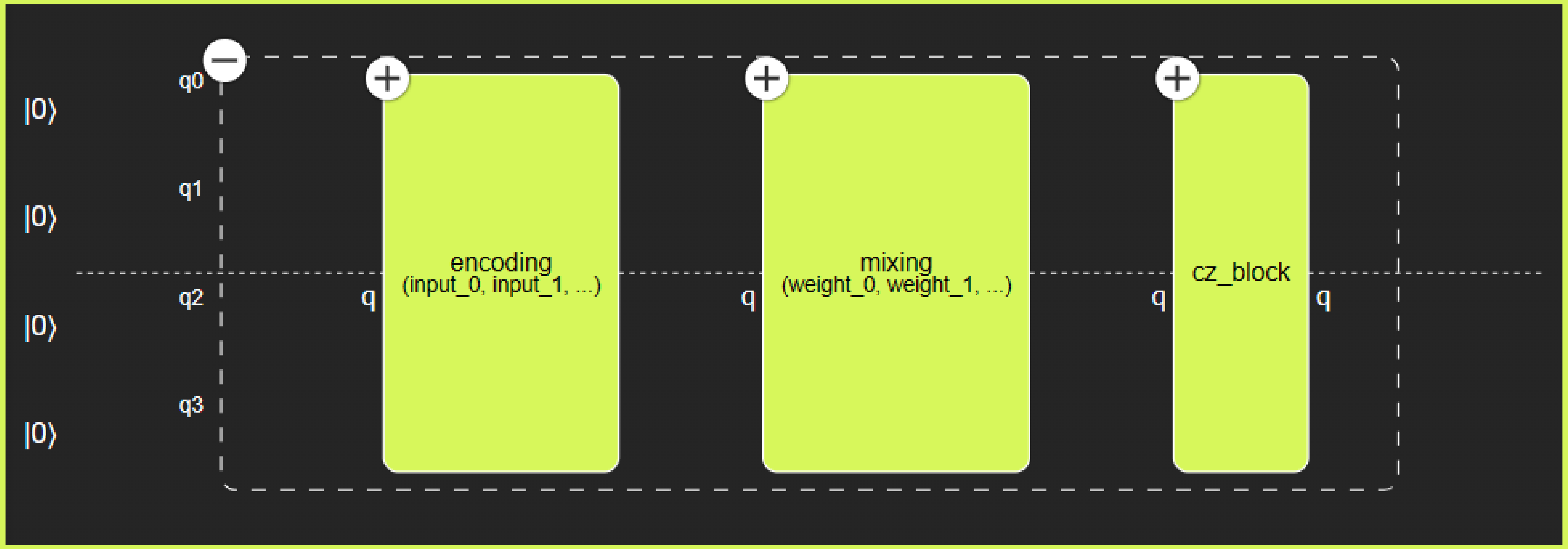


Fig: Snapshot of overall circuit of Quantum Layer

Post-processing the Measurement Results

We post process the measurement counts to get prediction probabilities of the labels.

```
def post_process_fn(result: SavedResult) -> torch.Tensor:
    counts: dict = result.value.counts

    # Calculate logits from counts
    logits: float = torch.zeros(16)
    for key, value in counts.items():
        logits[int(key, 2)] = value

    # Trim the logits from length 16 to length 2 since we have only 2 labels
    trimmed_logits = logits[:2]

    # Calculate prediction probabilities from logits by normalizing it
    pred_probs = torch.nn.functional.normalize(trimmed_logits, dim=0)

    return pred_probs.clone().detach()
```

```
SAMPLE EXAMPLE:
------------------------------------------------------------------------------------------------
COUNTS::
 {'1101': 33, '1100': 29, '1011': 19, '1000': 57, '0110': 327, '0010': 343, '0000': 413, '0011': 167,
 '0111': 147, '0001': 145, '1111': 47, '1001': 13, '0101': 91, '1110': 90, '1010': 79, '0100': 48}

LOGITS::
 tensor([413., 145., 343., 167.,  48.,  91., 327., 147.,  57.,  13.,  79.,  19.,
          29.,  33.,  90.,  47.])

TRIMMED LOGITS::
 tensor([413., 145.])

PREDICTION PROBABILITIES::
 tensor([0.9435, 0.3313])
```

Quantum Neural Network

```python
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()
```

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

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)
```

Experiment Results

Exp. No.	No. of Batches	Epochs	Classes	Train Loss	Train Time (mins)	Test Accuracy
1.	2	2	0-9	0.28655	2	9.375%
2.	8	2	0-9	0.20318	3	7.422%
3.	8	10	0-9	0.19270	35	8.203%
4.	80	10	0-1	0.10146	165	50%

We are getting approximately 50% accuracy for datasets of classes 0 and 1. Our future plan is to first increase the accuracy and then generalise the model for all classes of MNIST datasets (0 to 9).

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
- Project Weekly Report: https://jaisarita.hashnode.dev/qcourse511-qnn

Thank you!

Do you have any questions?