Quantum transformer

- Quantum Transformer:
  - At the core of any Transformer sits the so-called ***Multi-Headed Attention.***
  - We apply three different linear transformations $W_Q, W_K,$ and $W_V,$ to each element of the input sequence to transform each element embedding into some other internal representation states called Query (Q), Key (K) and Value (V). These states are then passed to the function that calculates the attention weights, which is simply defined as:

$$Attention(Q, K, V) = softmax_k(\frac{QK^T}{\sqrt{d_k}})V$$

  - To promote the Transformer from the classical to quantum real, one can simply replace the linear transformations $W_Q, W_K,$ and W_V with variational quantum circuits.

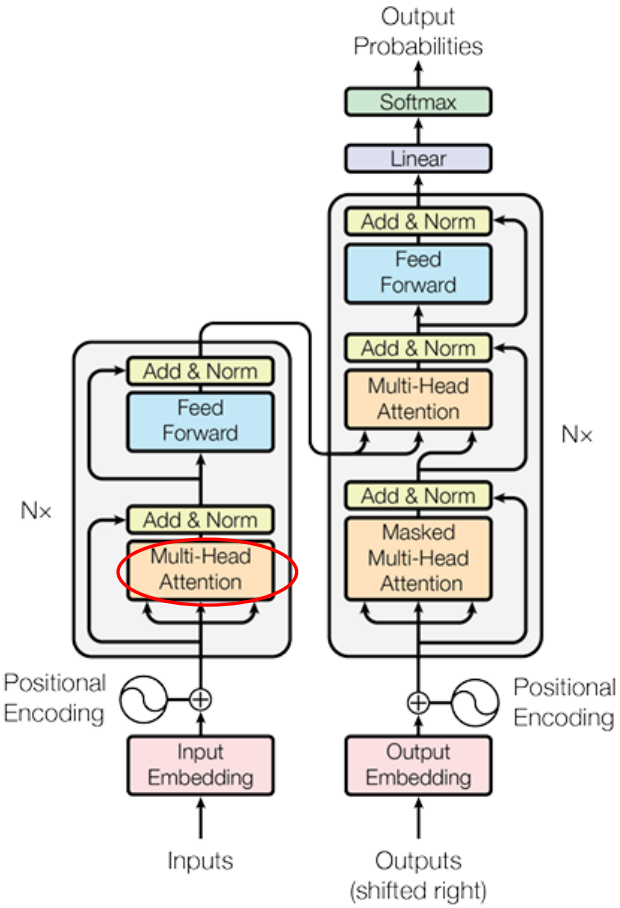


Figure 1: The Transformer - model architecture.

Code detail

Quantum transformer: (This code following here)

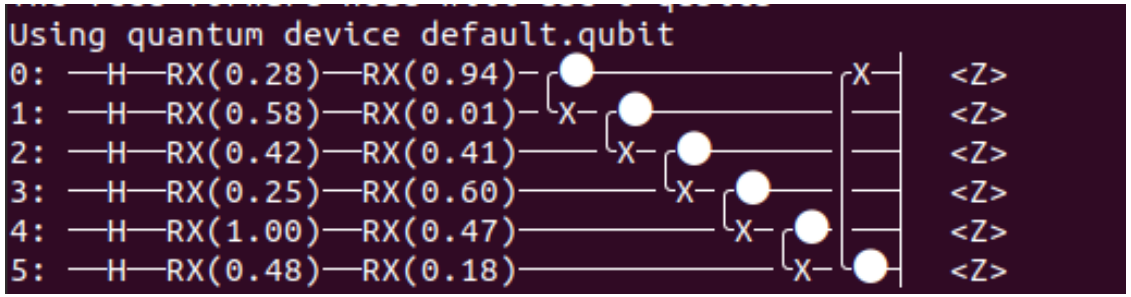
- The MultiHeadAttentionQuantum block
- Change the linear transformations.

```
class MultiHeadAttentionClassical(MultiHeadAttentionBase):
    def __init__(self,
                 embed_dim: int,
                 num_heads: int,
                 dropout=0.1,
                 mask=None,
                 use_bias=False):
        super(MultiHeadAttentionClassical, self).__init__(embed_dim=embed_dim, num_heads=num_heads, dropout=dropout, mask=mask, use_bias=use_bias)

        self.k_linear = nn.Linear(embed_dim, embed_dim, bias=use_bias)
        self.q_linear = nn.Linear(embed_dim, embed_dim, bias=use_bias)
        self.v_linear = nn.Linear(embed_dim, embed_dim, bias=use_bias)
        self.combine_heads = nn.Linear(embed_dim, embed_dim, bias=use_bias)
        self.head_dim = embed_dim // num_heads

    def forward(self, x, mask=None):
        batch_size, seq_len, embed_dim = x.size()
        assert embed_dim == self.embed_dim, f"Input embedding ({embed_dim}) does not match layer embedding size ({self.embed_dim})"

        K = self.k_linear(x)
        Q = self.q_linear(x)
        V = self.v_linear(x)

        x = self.downstream(Q, K, V, batch_size, mask)
        output = self.combine_heads(x)
        return output
```

```
self.n_qubits = n_qubits
self.n_qlayers = n_qlayers
self.q_device = q_device
self.head_dim = embed_dim // num_heads
if 'qulacs' in q_device:
    self.dev = qml.device(q_device, wires=self.n_qubits, gpu=True)
elif 'braket' in q_device:
    self.dev = qml.device(q_device, wires=self.n_qubits, parallel=True)
else:
    self.dev = qml.device(q_device, wires=self.n_qubits)
def _circuit(inputs, weights):
    for i in range(n_qubits):
        qml.Hadamard(wires=i)
    qml.AngleEmbedding(inputs, wires=range(n_qubits))
    qml.BasicEntanglerLayers(weights, wires=range(n_qubits))
    return [qml.expval(qml.PauliZ(wires=i)) for i in range(n_qubits)]

self.qlayer = qml.QNode(_circuit, self.dev, interface="torch")
self.weight_shapes = {"weights": (n_qlayers, n_qubits)}

#draw plots
weights = np.random.random([n_qlayers, n_qubits])
X = np.random.rand(6)
print(qml.draw(self.qlayer, expansion_strategy="device")(X,weights))

print(f"weight_shapes = (n_qlayers, n_qubits) = ({n_qlayers}, {self.n_qubits})")

self.k_linear = qml.qnn.TorchLayer(self.qlayer, self.weight_shapes)
self.q_linear = qml.qnn.TorchLayer(self.qlayer, self.weight_shapes)
self.v_linear = qml.qnn.TorchLayer(self.qlayer, self.weight_shapes)
self.combine_heads = qml.qnn.TorchLayer(self.qlayer, self.weight_shapes)
```

Quantum transformer model

- We make use of Xanadu's PennyLane quantum machine learning library to add quantum layers.
  - Add a function to the class and performs the quantum calculation (with a circuit) .
  - Wrap this circuit with a "QNode" to tell TensorFlow how to calculate the gradient with the parameter-shift rule.



  - Finally, we create a KerasLayer to handle the I/O within the hybrid neural network.(W_Q, W_K, and W_V)
  - Other part is same as the classical transformer.
  - https://github.com/shaqiyu/Quantum_transformer

How to work

- Download miniconda
- conda create -n env_name(Change the name as you want) root==6.24.00 python=3.8.6 -c conda-forge

Or:

source /hpcfs/cepc/higgsgpu/shaqy/miniconda/etc/profile.d/conda.sh

conda activate QT_re

- pip install -r requirment/*
- Python train_test.py (Change the option)

```
if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('-D', '--q_device', default='default.qubit', type=str)      #Fix
    parser.add_argument('-B', '--batch_size', default=32, type=int)                 #Fix for now.
    parser.add_argument('-E', '--n_epochs', default=10, type=int)                   #Fix for now, change after fix all hyperparameters
    parser.add_argument('-C', '--n_classes', default=2, type=int)                   #Fix
    parser.add_argument('-l', '--lr', default=0.001, type=float)                    #Changeable
    parser.add_argument('-v', '--vocab_size', default=6, type=int)
    parser.add_argument('-e', '--embed_dim', default=6, type=int)
    parser.add_argument('-f', '--ffn_dim', default=2048, type=int)                    #hidden layer dimension of feedforward networks. Changeable
    parser.add_argument('-t', '--n_transformer_blocks', default=2, type=int)       #Changeable
    parser.add_argument('-H', '--n_heads', default=2, type=int)
    parser.add_argument('-q', '--n_qubits_transformer', default=0, type=int) #6 for Quantum
    parser.add_argument('-Q', '--n_qubits_ffn', default=0, type=int)         #6 for Quantum
    parser.add_argument('-L', '--n_qlayers', default=1, type=int)            # For Quantum
    parser.add_argument('-d', '--dropout_rate', default=0.1, type=float)           #Changeable, but for few events, 0.1 is good
    args = parser.parse_args()


    Num_dataset = 20000
    if args.n_qubits_transformer > 0:
        Type_model = "Quantum"
    else:
        Type_model = "Classical"
```

Quantum transformer

- The dataset we used now is the CEPC MC sample
  - $e^+e^- \to ZH \to q\bar{q}\gamma\gamma$ (signal) and $e^+e^- \to (Z/\gamma^*)\gamma\gamma$ (background)
- Simulator: Pennlylane default device.
  - Time consuming: O(n), ~80 mins in CPU for 10k dataset with one epoch and one block(Q_layer).
- Current results (Use CPU): ~76% acc on validation dataset both in Quantum transformer and classical transformer in 20k dataset (10k train, 10k val).
  - Quantum:

```
Epoch 8/30
Info in <TCanvas::Print>: pdf file ./plot/ROCs/ROC_10000_train.pdf has been created
Info in <TCanvas::Print>: pdf file ./plot/ROCs/ROC_10000_val.pdf has been created
Epoch: 08 | Epoch Time: 140m 18s
       Train Loss: 0.510 | Train Acc: 76.24%
        Val. Loss: 0.500 |  Val. Acc: 76.52%
```

  - Classical:

```
Epoch 10/10
Info in <TCanvas::Print>: pdf file ./plot/ROCs/ROC_10000_train_Classical.pdf has been created
Info in <TCanvas::Print>: pdf file ./plot/ROCs/ROC_10000_val_Classical.pdf has been created
Epoch: 10 | Epoch Time: 0m 50s
       Train Loss: 0.485 | Train Acc: 76.39%
        Val. Loss: 0.467 |  Val. Acc: 77.66%
```