```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
import numpy as np

# Load MNIST data
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Normalize images to the range [0, 1]
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

x_train = np.expand_dims(x_train, axis=-1)
x_test = np.expand_dims(x_test, axis=-1)

# One-hot encode labels
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

```
from tensorflow.keras import layers, models

# Parameters
image_size = 28  # MNIST images are 28x28
patch_size = 7   # Size of each patch
num_patches = (image_size // patch_size) ** 2
projection_dim = 64
num_heads = 4
transformer_units = [
    projection_dim * 2,
    projection_dim,
]
transformer_layers = 8
mlp_head_units = [2048, 1024]

class Patches(layers.Layer):
    def __init__(self, patch_size):
        super().__init__()
        self.patch_size = patch_size

    def call(self, images):
        batch_size = tf.shape(images)[0]
        patches = tf.image.extract_patches(
            images=images,
            sizes=[1, self.patch_size, self.patch_size, 1],
            strides=[1, self.patch_size, self.patch_size, 1],
            rates=[1, 1, 1, 1],
            padding='VALID',
        )
        patch_dims = patches.shape[-1]
        patches = tf.reshape(patches, [batch_size, -1, patch_dims])
        return patches

class PatchEncoder(layers.Layer):
    def __init__(self, num_patches, projection_dim):
        super().__init__()
        self.num_patches = num_patches
        self.projection = layers.Dense(units=projection_dim)
        self.position_embedding = layers.Embedding(
            input_dim=num_patches, output_dim=projection_dim
        )

    def call(self, patch):
        positions = tf.range(start=0, limit=self.num_patches, delta=1)
        encoded = self.projection(patch) + self.position_embedding(positions)
        return encoded

def create_vit_classifier():
    inputs = layers.Input(shape=(image_size, image_size, 1))
    patches = Patches(patch_size)(inputs)
    encoded_patches = PatchEncoder(num_patches, projection_dim)(patches)

    for _ in range(transformer_layers):
        x1 = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
        attention_output = layers.MultiHeadAttention(
            num_heads=num_heads, key_dim=projection_dim, dropout=0.1
        )(x1, x1)
        x2 = layers.Add()([attention_output, encoded_patches])
```

```
        x3 = layers.LayerNormalization(epsilon=1e-6)(x2)
        x3 = layers.Dense(units=transformer_units[0], activation='relu')(x3)
        x3 = layers.Dense(units=transformer_units[1], activation='relu')(x3)
        x3 = layers.Dropout(0.1)(x3)
        encoded_patches = layers.Add()([x3, x2])

    representation = layers.LayerNormalization(epsilon=1e-6)(encoded_patches)
    representation = layers.Flatten()(representation)
    representation = layers.Dropout(0.5)(representation)
    features = layers.Dense(units=mlp_head_units[0], activation='relu')(representation)
    features = layers.Dropout(0.5)(features)
    features = layers.Dense(units=mlp_head_units[1], activation='relu')(features)
    features = layers.Dropout(0.5)(features)
    logits = layers.Dense(10)(features)
    model = models.Model(inputs=inputs, outputs=logits)
    return model

vit_classifier = create_vit_classifier()
vit_classifier.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-4),
    loss=tf.keras.losses.CategoricalCrossentropy(from_logits=True),
    metrics=['accuracy'],
)



# Train the model
history = vit_classifier.fit(
    x_train, y_train,
    batch_size=64,
    epochs=20,
    validation_data=(x_test, y_test),
)
```

```
Epoch 1/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 92s 38ms/step - accuracy: 0.5289 - loss: 1.4660 - val_accuracy: 0.9272 - val_loss: 0.2320
Epoch 2/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 40s 15ms/step - accuracy: 0.8824 - loss: 0.3717 - val_accuracy: 0.9548 - val_loss: 0.1473
Epoch 3/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 14s 15ms/step - accuracy: 0.9215 - loss: 0.2474 - val_accuracy: 0.9620 - val_loss: 0.1131
Epoch 4/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 20s 14ms/step - accuracy: 0.9418 - loss: 0.1872 - val_accuracy: 0.9702 - val_loss: 0.0906
Epoch 5/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 14s 15ms/step - accuracy: 0.9508 - loss: 0.1571 - val_accuracy: 0.9726 - val_loss: 0.0817
Epoch 6/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 13s 14ms/step - accuracy: 0.9605 - loss: 0.1294 - val_accuracy: 0.9760 - val_loss: 0.0704
Epoch 7/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 14s 15ms/step - accuracy: 0.9629 - loss: 0.1170 - val_accuracy: 0.9795 - val_loss: 0.0600
Epoch 8/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 20s 14ms/step - accuracy: 0.9658 - loss: 0.1086 - val_accuracy: 0.9808 - val_loss: 0.0570
Epoch 9/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 20s 14ms/step - accuracy: 0.9728 - loss: 0.0922 - val_accuracy: 0.9820 - val_loss: 0.0507
Epoch 10/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 20s 14ms/step - accuracy: 0.9723 - loss: 0.0871 - val_accuracy: 0.9832 - val_loss: 0.0507
Epoch 11/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 14s 15ms/step - accuracy: 0.9760 - loss: 0.0765 - val_accuracy: 0.9840 - val_loss: 0.0456
Epoch 12/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 21s 15ms/step - accuracy: 0.9785 - loss: 0.0677 - val_accuracy: 0.9847 - val_loss: 0.0494
Epoch 13/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 14s 15ms/step - accuracy: 0.9800 - loss: 0.0646 - val_accuracy: 0.9845 - val_loss: 0.0463
Epoch 14/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 13s 14ms/step - accuracy: 0.9818 - loss: 0.0591 - val_accuracy: 0.9846 - val_loss: 0.0484
Epoch 15/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 21s 15ms/step - accuracy: 0.9817 - loss: 0.0559 - val_accuracy: 0.9847 - val_loss: 0.0451
Epoch 16/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 21s 15ms/step - accuracy: 0.9834 - loss: 0.0512 - val_accuracy: 0.9858 - val_loss: 0.0484
Epoch 17/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 20s 15ms/step - accuracy: 0.9834 - loss: 0.0515 - val_accuracy: 0.9871 - val_loss: 0.0437
Epoch 18/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 13s 14ms/step - accuracy: 0.9858 - loss: 0.0441 - val_accuracy: 0.9862 - val_loss: 0.0443
Epoch 19/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 14s 15ms/step - accuracy: 0.9861 - loss: 0.0432 - val_accuracy: 0.9873 - val_loss: 0.0421
Epoch 20/20
938/938 ━━━━━━━━━━━━━━━━━━━━ 14s 15ms/step - accuracy: 0.9858 - loss: 0.0416 - val_accuracy: 0.9862 - val_loss: 0.0465
```

```
# Evaluate the model
test_loss, test_accuracy = vit_classifier.evaluate(x_test, y_test)
print(f'Test accuracy: {test_accuracy * 100:.2f}%')
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 6s 9ms/step - accuracy: 0.9819 - loss: 0.0637
Test accuracy: 98.62%
```

Comments on Extending Vision Transformer to Quantum Vision Transformer

The classical Vision Transformer (ViT) implemented above effectively learns spatial dependencies in MNIST images through self-attention mechanisms. To extend this approach to a Quantum Vision Transformer (QViT), we can leverage quantum data encoding, quantum self-attention, and hybrid quantum-classical training.

In a QViT, image patches would be encoded into quantum states using quantum feature maps, allowing for quantum parallelism in processing multiple patches simultaneously. The self-attention mechanism could be replaced by variational quantum circuits (VQCs), where entanglement enables efficient representation of relationships between patches. The model would then be trained using a hybrid approach, where classical optimizers update quantum gate parameters.

One challenge in this transition is efficient quantum data encoding, as current quantum devices have limited qubit capacity. Additionally, quantum circuits must be shallow to remain feasible on near-term quantum hardware. However, if implemented effectively, QViTs could potentially achieve superior feature extraction and computational efficiency compared to classical transformers, especially for high-dimensional datasets.

This approach could be further explored in future work, focusing on optimizing quantum attention mechanisms and evaluating performance on larger datasets beyond MNIST.