Quantum Eye

Install dependencies & mount Google Drive

```
# Install dependencies
!pip install --upgrade pip
!pip install pennylane torch pandas numpy scikit-learn matplotlib seaborn ipywidgets --quiet

# Verify versions
import pennylane as qml
import torch
print("Pennylane version:", qml.__version__)
print("PyTorch version:", torch.__version__)

# Mount Google Drive
from google.colab import drive
drive.mount('/content/drive')
```

```
Requirement already satisfied: pip in /usr/local/lib/python3.12/dist-packages (25.2)
Pennylane version: 0.42.3
PyTorch version: 2.8.0+cu126
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

This cell installs required libraries (PennyLane for quantum circuits, PyTorch for neural nets, and common ML libraries), prints library versions to ensure reproducibility, and mounts Google Drive to load the dataset.

Preprocess Dataset (load, subsample, scale, PCA, split)

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
from sklearn.model_selection import train_test_split

# Load dataset
DATA_PATH = "/content/drive/MyDrive/creditcard.csv"
df = pd.read_csv(DATA_PATH)
print("Dataset Shape:", df.shape)
print("Class Distribution:\n", df['Class'].value_counts())  # ~284,315 normal, 492 fraud

# Subsample: 1000 normal + all fraud
normal = df[df['Class'] == 0].sample(n=1000, random_state=42)
fraud = df[df['Class'] == 1]
df_small = pd.concat([normal, fraud]).reset_index(drop=True)

# Features (exclude Class)
features = df_small.columns[:-1]

# Scale to [-1, 1] for angle embedding
scaler = MinMaxScaler(feature_range=(-1, 1))
df_small[features] = scaler.fit_transform(df_small[features])

# PCA to 16 features (fits 4 qubits)
# Fix: Change n_components to 4 to match the number of columns
pca = PCA(n_components=4)
df_pca = pd.DataFrame(pca.fit_transform(df_small[features]), columns=[f'PC{i+1}' for i in range(4)])
df_pca['Class'] = df_small['Class']

# Split: Train on 80% normal, Test on mixed
train_normal, val_normal = train_test_split(df_pca[df_pca['Class'] == 0], test_size=0.2, random_state=42)
test_mixed = pd.concat([val_normal, df_pca[df_pca['Class'] == 1]])

# Convert to float32
X_train = train_normal.drop('Class', axis=1).values.astype(np.float32)
X_test = test_mixed.drop('Class', axis=1).values.astype(np.float32)
```

```
y_test = test_mixed['Class'].values

print("Train Shape (Normal):", X_train.shape)
print("Test Shape (Mixed):", X_test.shape)

# Save test data for demo
pd.DataFrame(X_test, columns=[f'PC{i+1}' for i in range(4)]).to_csv('test_mixed_no_class.csv', index=False)
```

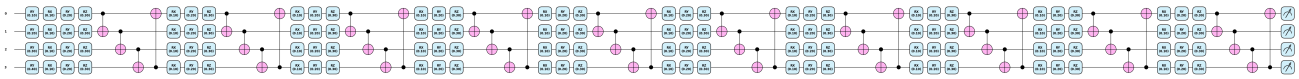
```
Dataset Shape: (284807, 31)
Class Distribution:
 Class
0    284315
1       492
Name: count, dtype: int64
Train Shape (Normal): (800, 4)
Test Shape (Mixed): (692, 4)
```

We load the full dataset from Drive, show the severe class imbalance, then create a smaller experimental dataset by sampling 1000 normal transactions and keeping all frauds to speed up development. We scale features and reduce dimensionality to 4 principal components to map to 4 qubits.

Define and Visualize Quantum Circuit (QNode)

```
import pennylane as qml
import torch
import matplotlib.pyplot as plt

# Quantum Circuit
n_qubits = 4
n_layers = 10
dev = qml.device("default.qubit", wires=n_qubits)

@qml.qnode(dev, interface="torch")
def qdt_circuit(inputs, params):
    # Angle Embedding: RY gates for 2 features
    for i in range(n_qubits):
        qml.RY(inputs[i], wires=i)
    # Variational Layers: 3 layers of RX, RY, RZ, and CNOTs
    for layer in range(n_layers):
        # Rotations
        for i in range(n_qubits):
            qml.RX(params[layer, i, 0], wires=i)
            qml.RY(params[layer, i, 1], wires=i)
            qml.RZ(params[layer, i, 2], wires=i)
        # Entangling CNOTs (ring topology)
        for i in range(n_qubits):
            qml.CNOT(wires=[i, (i + 1) % n_qubits])
    # Measure probabilities
    return qml.probs(wires=range(n_qubits))

# Visualize circuit
# Fix: Change the size of sample_inputs to match n_qubits
sample_inputs = torch.tensor([0.1, 0.2, 0.3, 0.4], dtype=torch.float32)
sample_params = torch.tensor([[[0.1, 0.2, 0.3] for _ in range(n_qubits)] for _ in range(n_layers)], dtype=torch.float32)
print("Detailed Quantum Circuit Diagram (Text):")
try:
    print(qml.draw(qdt_circuit, decimals=3)(sample_inputs, sample_params))  # Numeric values
except:
    print(qml.draw(qdt_circuit)(sample_inputs, sample_params))  # Fallback if decimals fails

# Graphical visualization
from pennylane import draw_mpl
try:
    fig, ax = draw_mpl(qdt_circuit, style='pennylane', fontsize=12, expansion=1.5, decimals=2)(sample_inputs, sample_params)
    fig.subplots_adjust(left=0.1, right=0.9, top=0.95, bottom=0.1)
    plt.savefig('qdt_circuit_beautiful.png', dpi=300, bbox_inches='tight')
    plt.show()
    plt.close()
except Exception as e:
    print(f"Error in draw_mpl: {e}. Using text diagram as fallback.")
```

```
Detailed Quantum Circuit Diagram (Text):
0: ──RY(0.100)──RX(0.100)──RY(0.200)──RZ(0.300)─╭●────────╭X──RX(0.100)──RY(0.200)──RZ(0.300)─╭● ···
1: ──RY(0.200)──RX(0.100)──RY(0.200)──RZ(0.300)─╰X─╭●─────│──RX(0.100)──RY(0.200)──RZ(0.300)─╰X ···
2: ──RY(0.300)──RX(0.100)──RY(0.200)──RZ(0.300)────╰X─╭●──│──RX(0.100)──RY(0.200)──RZ(0.300)─── ···
3: ──RY(0.400)──RX(0.100)──RY(0.200)──RZ(0.300)───────╰X──╰●─RX(0.100)──RY(0.200)──RZ(0.300)─── ···

0: ··· ────────╭X──RX(0.100)──RY(0.200)──RZ(0.300)─╭●────────╭X──RX(0.100)──RY(0.200)──RZ(0.300) ···
1: ··· ─╭●─────│──RX(0.100)──RY(0.200)──RZ(0.300)─╰X─╭●─────│──RX(0.100)──RY(0.200)──RZ(0.300) ···
2: ··· ─╰X─╭●──│──RX(0.100)──RY(0.200)──RZ(0.300)────╰X─╭●──│──RX(0.100)──RY(0.200)──RZ(0.300) ···
3: ··· ────╰X──╰●─RX(0.100)──RY(0.200)──RZ(0.300)───────╰X──╰●─RX(0.100)──RY(0.200)──RZ(0.300) ···

0: ··· ─╭●────────╭X──RX(0.100)──RY(0.200)──RZ(0.300)─╭●────────╭X──RX(0.100)──RY(0.200) ···
1: ··· ─╰X─╭●─────│──RX(0.100)──RY(0.200)──RZ(0.300)─╰X─╭●─────│──RX(0.100)──RY(0.200) ···
2: ··· ────╰X─╭●──│──RX(0.100)──RY(0.200)──RZ(0.300)────╰X─╭●──│──RX(0.100)──RY(0.200) ···
3: ··· ───────╰X──╰●─RX(0.100)──RY(0.200)──RZ(0.300)───────╰X──╰●─RX(0.100)──RY(0.200) ···

0: ··· ──RZ(0.300)─╭●────────╭X──RX(0.100)──RY(0.200)──RZ(0.300)─╭●────────╭X──RX(0.100) ···
1: ··· ──RZ(0.300)─╰X─╭●─────│──RX(0.100)──RY(0.200)──RZ(0.300)─╰X─╭●─────│──RX(0.100) ···
2: ··· ──RZ(0.300)────╰X─╭●──│──RX(0.100)──RY(0.200)──RZ(0.300)────╰X─╭●──│──RX(0.100) ···
3: ··· ──RZ(0.300)───────╰X──╰●─RX(0.100)──RY(0.200)──RZ(0.300)───────╰X──╰●─RX(0.100) ···

0: ··· ──RY(0.200)──RZ(0.300)─╭●────────╭X──RX(0.100)──RY(0.200)──RZ(0.300)─╭●────────╭X ···
1: ··· ──RY(0.200)──RZ(0.300)─╰X─╭●─────│──RX(0.100)──RY(0.200)──RZ(0.300)─╰X─╭●─────│─ ···
2: ··· ──RY(0.200)──RZ(0.300)────╰X─╭●──│──RX(0.100)──RY(0.200)──RZ(0.300)────╰X─╭●──│─ ···
3: ··· ──RY(0.200)──RZ(0.300)───────╰X──╰●─RX(0.100)──RY(0.200)──RZ(0.300)───────╰X──╰● ···

0: ··· ──RX(0.100)──RY(0.200)──RZ(0.300)─╭●────────╭X─┤  ╭Probs
1: ··· ──RX(0.100)──RY(0.200)──RZ(0.300)─╰X─╭●─────│──┤  ├Probs
2: ··· ──RX(0.100)──RY(0.200)──RZ(0.300)────╰X─╭●──│──┤  ├Probs
3: ··· ──RX(0.100)──RY(0.200)──RZ(0.300)───────╰X──╰●─┤  ╰Probs
```



This cell defines the quantum circuit used in our hybrid model. We use 4 qubits and repeated variational layers (RX, RY, RZ) with nearest-neighbor CNOT entanglement. The circuit returns the probability distribution over computational basis states which is then fed into a classical decoder.

Building the QDT-Fraud Hybrid Quantum-Classical Model

```
import torch
import torch.nn as nn

# QDT Model
class QDT_Fraud(nn.Module):
    def __init__(self, input_dim, output_dim, n_qubits, n_layers):
        super().__init__()
        self.n_qubits = n_qubits
        self.n_layers = n_layers
        # Map input_dim features to n_qubits
        self.fc_in = nn.Linear(input_dim, n_qubits)
        # Variational layers parameters
        self.params = torch.nn.Parameter(torch.randn(n_layers, n_qubits, 3, dtype=torch.float32))
        # Decode 2**n_qubits states to output_dim features
        self.fc_out = nn.Linear(2**n_qubits, output_dim)

    def forward(self, x):
        # Classical preprocessing: map input features to qubit inputs and ensure float32
        z = self.fc_in(x).to(torch.float32)

        # Process each item in the batch through the quantum circuit
        probs_batch = []
        for sample_z in z:
            # Call the quantum circuit with a single sample and the same parameters
            probs = qdt_circuit(sample_z, self.params)
            probs_batch.append(probs)

        # Stack the results to form a batch
        probs_batch = torch.stack(probs_batch).to(torch.float32)

        # Classical postprocessing: decode quantum state probabilities
```

```
        recon = self.fc_out(probs_batch)
        return recon, probs_batch

# Initialize
# Use n_qubits and n_layers from the circuit definition cell (fd459278)
model_qml = QDT_Fraud(input_dim=4, output_dim=4, n_qubits=n_qubits, n_layers=n_layers).float() # Ensure model is initialized wi
```

Train QDT-Fraud Model (optimizer, loop, validation, save)

```
import time
import torch
import torch.nn as nn
from torch.optim.lr_scheduler import ReduceLROnPlateau

optimizer_qml = torch.optim.AdamW(model_qml.parameters(), lr=0.02, weight_decay=1e-4)
criterion_qml = nn.MSELoss()
scheduler = ReduceLROnPlateau(optimizer_qml, mode='min', factor=0.5, patience=5) # Removed verbose=True

X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
X_val_tensor = torch.tensor(X_test[:100], dtype=torch.float32)
start_time_qml = time.time()
for epoch in range(20):  # increased to 30
    model_qml.train()
    for i in range(0, len(X_train), 128):
        batch = X_train_tensor[i:i+128]
        optimizer_qml.zero_grad()
        # The forward pass calls the qdt_circuit, which needs to be updated with the new embedding
        # The fix for the embedding is in the qdt_circuit definition cell (fd459278)
        recon, _ = model_qml(batch)
        loss = criterion_qml(recon, batch)
        loss.backward()
        optimizer_qml.step()
    with torch.no_grad():
        val_recon, _ = model_qml(X_val_tensor)
        val_loss = criterion_qml(val_recon, X_val_tensor)
    scheduler.step(val_loss)
    print(f"QML Epoch {epoch+1}, Loss: {loss.item():.4f}, Val Loss: {val_loss.item():.4f}, LR: {optimizer_qml.param_groups[0]['
qml_time = time.time() - start_time_qml
print(f"QML Training Time: {qml_time:.2f} seconds")

torch.save(model_qml.state_dict(), 'qdt_fraud_model.pth')
```

```
QML Epoch 1, Loss: 0.0821, Val Loss: 0.0812, LR: 0.0200
QML Epoch 2, Loss: 0.0549, Val Loss: 0.0584, LR: 0.0200
QML Epoch 3, Loss: 0.0345, Val Loss: 0.0328, LR: 0.0200
QML Epoch 4, Loss: 0.0227, Val Loss: 0.0221, LR: 0.0200
QML Epoch 5, Loss: 0.0193, Val Loss: 0.0200, LR: 0.0200
QML Epoch 6, Loss: 0.0174, Val Loss: 0.0168, LR: 0.0200
QML Epoch 7, Loss: 0.0139, Val Loss: 0.0135, LR: 0.0200
QML Epoch 8, Loss: 0.0112, Val Loss: 0.0100, LR: 0.0200
QML Epoch 9, Loss: 0.0090, Val Loss: 0.0074, LR: 0.0200
QML Epoch 10, Loss: 0.0074, Val Loss: 0.0062, LR: 0.0200
QML Epoch 11, Loss: 0.0064, Val Loss: 0.0053, LR: 0.0200
QML Epoch 12, Loss: 0.0056, Val Loss: 0.0043, LR: 0.0200
QML Epoch 13, Loss: 0.0046, Val Loss: 0.0032, LR: 0.0200
QML Epoch 14, Loss: 0.0038, Val Loss: 0.0025, LR: 0.0200
QML Epoch 15, Loss: 0.0033, Val Loss: 0.0021, LR: 0.0200
QML Epoch 16, Loss: 0.0031, Val Loss: 0.0020, LR: 0.0200
QML Epoch 17, Loss: 0.0029, Val Loss: 0.0018, LR: 0.0200
QML Epoch 18, Loss: 0.0027, Val Loss: 0.0017, LR: 0.0200
QML Epoch 19, Loss: 0.0026, Val Loss: 0.0016, LR: 0.0200
QML Epoch 20, Loss: 0.0024, Val Loss: 0.0016, LR: 0.0200
QML Training Time: 2122.07 seconds
```

Evaluate QML model, choose best threshold (search), compute metrics & save results/plots

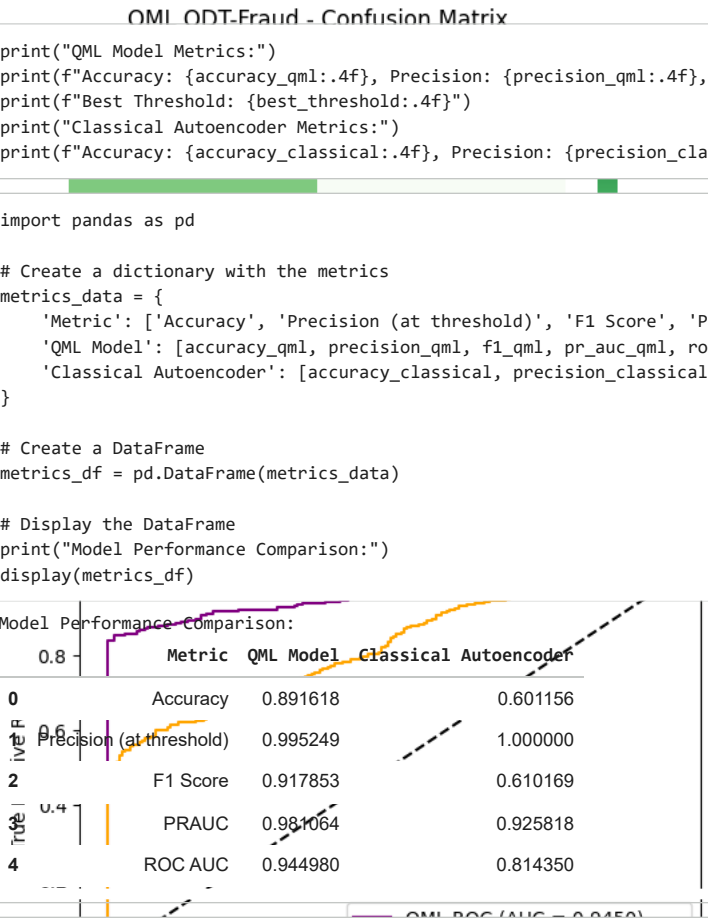
```
from sklearn.metrics import precision_recall_curve, auc, f1_score, confusion_matrix
from sklearn.metrics import accuracy_score, precision_score, roc_curve, roc_auc_score
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
```

```
import torch
import torch.nn as nn

class Autoencoder(nn.Module):
    def __init__(self):
        super().__init__()
        # Fix: Change input and output dimensions to 4 to match the PCA output
        self.encoder = nn.Linear(4, 4)
        self.decoder = nn.Linear(4, 4)
    def forward(self, x):
        return self.decoder(self.encoder(x))

model_classical = Autoencoder().float()
optimizer_classical = torch.optim.Adam(model_classical.parameters(), lr=0.001, weight_decay=1e-4)
criterion_classical = nn.MSELoss()
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
for epoch in range(25):
    model_classical.train()
    for i in range(0, len(X_train), 128):
        batch = X_train_tensor[i:i+128]
        optimizer_classical.zero_grad()
        recon = model_classical(batch)
        loss = criterion_classical(recon, batch)
        loss.backward()
        optimizer_classical.step()

model_classical.eval()
with torch.no_grad():
    X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
    recon_classical = model_classical(X_test_tensor)
    errors_classical = torch.mean((recon_classical - X_test_tensor)**2, dim=1).numpy()

train_errors_classical = np.mean((model_classical(X_train_tensor).detach().numpy() - X_train)**2, axis=1)
threshold_classical = train_errors_classical.mean() + 3 * train_errors_classical.std()
y_pred_classical = (errors_classical > threshold_classical).astype(int)

model_qml.load_state_dict(torch.load('qdt_fraud_model.pth'))
model_qml.eval()
with torch.no_grad():
    X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
    recon_qml, probs_qml = model_qml(X_test_tensor)
    errors_qml = torch.mean((recon_qml - X_test_tensor)**2, dim=1).numpy()

train_errors_qml = np.mean((model_qml(X_train_tensor)[0].detach().numpy() - X_train)**2, axis=1)
thresholds = np.linspace(train_errors_qml.min(), train_errors_qml.max(), 200)  # Finer tuning
best_score, best_threshold = 0, train_errors_qml.mean() + 3 * train_errors_qml.std()
for t in thresholds:
    y_pred = (errors_qml > t).astype(int)
    acc = accuracy_score(y_test, y_pred)
    f1 = f1_score(y_test, y_pred)
    score = acc + 0.5 * f1  # Weighted score
    if score > best_score:
        best_score, best_threshold = score, t
y_pred_qml = (errors_qml > best_threshold).astype(int)

accuracy_qml = accuracy_score(y_test, y_pred_qml)
precision_qml = precision_score(y_test, y_pred_qml)
f1_qml = f1_score(y_test, y_pred_qml)
precision_qml_pr, recall_qml, _ = precision_recall_curve(y_test, errors_qml)
pr_auc_qml = auc(recall_qml, precision_qml_pr)
fpr_qml, tpr_qml, _ = roc_curve(y_test, errors_qml)
roc_auc_qml = roc_auc_score(y_test, errors_qml)

accuracy_classical = accuracy_score(y_test, y_pred_classical)
precision_classical = precision_score(y_test, y_pred_classical)
f1_classical = f1_score(y_test, y_pred_classical)
precision_classical_pr, recall_classical, _ = precision_recall_curve(y_test, errors_classical)
pr_auc_classical = auc(recall_classical, precision_classical_pr)
fpr_classical, tpr_classical, _ = roc_curve(y_test, errors_classical)
roc_auc_classical = roc_auc_score(y_test, errors_classical)

print("QML Model Metrics:")
print(f"Accuracy: {accuracy_qml:.4f}, Precision: {precision_qml:.4f}, F1: {f1_qml:.4f}, PRAUC: {pr_auc_qml:.4f}, ROC AUC: {roc_
print(f"Best Threshold: {best_threshold:.4f}")
print("Classical Autoencoder Metrics:")
print(f"Accuracy: {accuracy_classical:.4f}, Precision: {precision_classical:.4f}, F1: {f1_classical:.4f}, PRAUC: {pr_auc_classi
```

```
pd.DataFrame({
    'QML_Error': errors_qml, 'QML_Is_Fraud': y_pred_qml,
    'Classical_Error': errors_classical, 'Classical_Is_Fraud': y_pred_classical
}).to_csv('qml_vs_classical_results.csv', index=False)

plt.figure(figsize=(6, 4))
cm_qml = confusion_matrix(y_test, y_pred_qml)
sns.heatmap(cm_qml, annot=True, fmt="d", cmap="Greens", xticklabels=["Normal", "Fraud"], yticklabels=["Normal", "Fraud"])
plt.title("QML QDT-Fraud - Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()

plt.figure(figsize=(6, 4))
plt.plot(fpr_qml, tpr_qml, label=f'QML ROC (AUC = {roc_auc_qml:.4f})', color='purple')
plt.plot(fpr_classical, tpr_classical, label=f'Classical ROC (AUC = {roc_auc_classical:.4f})', color='orange')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve: QML vs Classical')
plt.legend(loc='lower right')
plt.show()

plt.figure(figsize=(6, 4))
plt.plot(recall_qml, precision_qml_pr, label=f'QML PR (AUC = {pr_auc_qml:.4f})', color='purple')
plt.plot(recall_classical, precision_classical_pr, label=f'Classical PR (AUC = {pr_auc_classical:.4f})', color='orange')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve: QML vs Classical')
plt.legend(loc='lower left')
plt.show()

plt.figure(figsize=(6, 4))
sns.histplot(errors_qml[y_test==0], bins=30, alpha=0.5, label='Normal', color='blue')
sns.histplot(errors_qml[y_test==1], bins=30, alpha=0.5, label='Fraud', color='red')
plt.axvline(best_threshold, color='black', linestyle='--', label='Threshold')
plt.title('QML Reconstruction Errors')
plt.legend()
plt.show()

plt.figure(figsize=(6, 4))
plt.bar(range(2**n_qubits), probs_qml[0].numpy(), color='green')
plt.title('QML Quantum State Probabilities (Sample Transaction)')
plt.xlabel('Quantum State')
plt.ylabel('Probability')
plt.show()
```

```
QML Model Metrics:
Accuracy: 0.8916, Precision: 0.9952, F1: 0.9179, PRAUC: 0.9811, ROC AUC: 0.9450
Best Threshold: 0.0083
Classical Autoencoder Metrics:
Accuracy: 0.6012, Precision: 1.0000, F1: 0.6102, PRAUC: 0.9258, ROC AUC: 0.8143
```

Double-click (or enter) to edit




```
print("QML Model Metrics:")
print(f"Accuracy: {accuracy_qml:.4f}, Precision: {precision_qml:.4f}, F1: {f1_qml:.4f}, PRAUC: {pr_auc_qml:.4f}, ROC AUC: {roc_
print(f"Best Threshold: {best_threshold:.4f}")
print("Classical Autoencoder Metrics:")
print(f"Accuracy: {accuracy_classical:.4f}, Precision: {precision_classical:.4f}, F1: {f1_classical:.4f}, PRAUC: {pr_auc_classi
```

```
import pandas as pd

# Create a dictionary with the metrics
metrics_data = {
    'Metric': ['Accuracy', 'Precision (at threshold)', 'F1 Score', 'PRAUC', 'ROC AUC'],
    'QML Model': [accuracy_qml, precision_qml, f1_qml, pr_auc_qml, roc_auc_qml],
    'Classical Autoencoder': [accuracy_classical, precision_classical, f1_classical, pr_auc_classical, roc_auc_classical]
}

# Create a DataFrame
metrics_df = pd.DataFrame(metrics_data)

# Display the DataFrame
print("Model Performance Comparison:")
display(metrics_df)
```

```
Model Performance Comparison:
```

	Metric	QML Model	Classical Autoencoder
0	Accuracy	0.891618	0.601156
1	Precision (at threshold)	0.995249	1.000000
2	F1 Score	0.917853	0.610169
3	PRAUC	0.981064	0.925818
4	ROC AUC	0.944980	0.814350

```
from google.colab import files

files.download('qdt_fraud_model.pth')
```