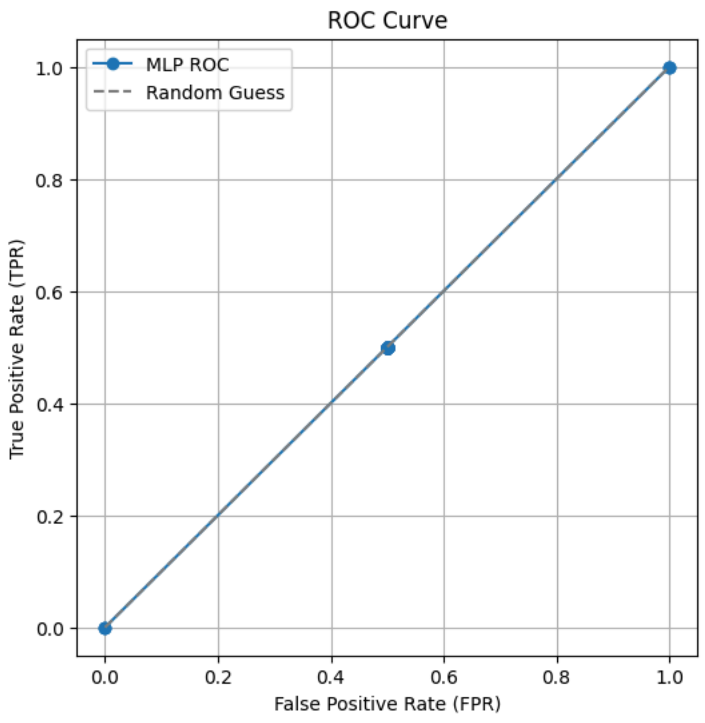
Start coding or generate with AI.

```
import numpy as np
import matplotlib.pyplot as plt

# ---------------- Activation Functions ---------------- #
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def dsigmoid(y):
    # y is already sigmoid(x)
    return y * (1 - y)

def relu(x):
    return np.maximum(0, x)

def drelu(x):
    return (x > 0).astype(float)

# ---------------- MLP Class ---------------- #
class MLP221:
    def __init__(self, act='sigmoid', seed=0):
        np.random.seed(seed)
        self.act = act
        # Weight initialization
        self.W1 = np.random.randn(2, 2) * 0.5
        self.b1 = np.zeros((1, 2))
        self.W2 = np.random.randn(2, 1) * 0.5
        self.b2 = np.zeros((1, 1))

    def forward(self, X):
        self.Z1 = X @ self.W1 + self.b1
        self.A1 = sigmoid(self.Z1) if self.act=='sigmoid' else relu(self.Z1)
        self.Z2 = self.A1 @ self.W2 + self.b2
        self.A2 = sigmoid(self.Z2)
        return self.A2

    def fit(self, X, y, lr=0.1, epochs=5000):
        y = y.reshape(-1, 1)
        for _ in range(epochs):
            A2 = self.forward(X)
            # Backprop
            dA2 = (A2 - y) * dsigmoid(A2)
            dW2 = self.A1.T @ dA2
            db2 = dA2.sum(0, keepdims=True)

            dA1 = dA2 @ self.W2.T * (dsigmoid(self.A1) if self.act=='sigmoid' else drelu(self.Z1))
            dW1 = X.T @ dA1
            db1 = dA1.sum(0, keepdims=True)

            # Gradient descent
            self.W1 -= lr * dW1
            self.b1 -= lr * db1
            self.W2 -= lr * dW2
            self.b2 -= lr * db2

    def predict(self, X):
        return (self.forward(X) >= 0.5).astype(int)

    def accuracy(self, X, y):
        return np.mean(self.predict(X).ravel() == y)

# ---------------- XOR Dataset ---------------- #
X = np.array([[0,0],[0,1],[1,0],[1,1]], float)
y = np.array([0,1,1,0], int)

# ---------------- Train and Evaluate ---------------- #
mlp = MLP221(act='sigmoid')
mlp.fit(X, y, lr=0.5, epochs=5000)

y_pred = mlp.predict(X)
y_prob = mlp.forward(X)

print("Predictions:", y_pred.ravel())
print("Accuracy:", np.mean(y_pred.ravel() == y))
# ---------------- Metrics ---------------- #
def precision(y_true, y_pred):
    tp = np.sum((y_true==1) & (y_pred==1))
    fp = np.sum((y_true==0) & (y_pred==1))
    return tp / (tp + fp + 1e-10)
```

```
def recall(y_true, y_pred):
    tp = np.sum((y_true==1) & (y_pred==1))
    fn = np.sum((y_true==1) & (y_pred==0))
    return tp / (tp + fn + 1e-10)

def f1_score(y_true, y_pred):
    p = precision(y_true, y_pred)
    r = recall(y_true, y_pred)
    return 2 * p * r / (p + r + 1e-10)

def roc_curve(y_true, y_scores, num_thresholds=100):
    thresholds = np.linspace(0,1,num_thresholds)
    tpr_list, fpr_list = [], []
    y_true = y_true.ravel()
    for t in thresholds:
        y_pred = (y_scores >= t).astype(int)
        tp = np.sum((y_true==1) & (y_pred==1))
        fp = np.sum((y_true==0) & (y_pred==1))
        fn = np.sum((y_true==1) & (y_pred==0))
        tn = np.sum((y_true==0) & (y_pred==0))
        tpr_list.append(tp / (tp + fn + 1e-10))
        fpr_list.append(fp / (fp + tn + 1e-10))
    return np.array(fpr_list), np.array(tpr_list), thresholds

print("Precision:", precision(y, y_pred))
print("Recall:", recall(y, y_pred))
print("F1-Score:", f1_score(y, y_pred))

# ---------------- Plot ROC Curve ---------------- #
fpr, tpr, thresholds = roc_curve(y, y_prob)
plt.figure(figsize=(6,6))
plt.plot(fpr, tpr, marker='o', label='MLP ROC')
plt.plot([0,1],[0,1],'--', color='gray', label='Random Guess')
plt.xlabel("False Positive Rate (FPR)")
plt.ylabel("True Positive Rate (TPR)")
plt.title("ROC Curve")
plt.legend()
plt.grid(True)
plt.show()
```

```
Predictions: [0 1 1 0]
Accuracy: 1.0
Precision: 0.49999999999375
Recall: 0.49999999999375
F1-Score: 0.49999999994375
```


ROC Curve
MLP ROC
Random Guess
True Positive Rate (TPR)
False Positive Rate (FPR)