Enhancement 5: Choose a new dataset from the list below. Search the Internet and download your chosen dataset (many of them could be available on kaggle). Adapt your model to your dataset. Train your model and record your results.

- cancer_dataset - Breast cancer dataset.
- crab_dataset - Crab gender dataset.
- glass_dataset - Glass chemical dataset.
- iris_dataset - Iris flower dataset.
- ovarian_dataset - Ovarian cancer dataset.
- thyroid_dataset - Thyroid function dataset.

Summary of Model Performance:

- **Test Loss**: 0.0059 — The test loss is very low, indicating that the model's predictions are extremely close to the actual values on the test set.
- **Accuracy**: 1.0000 (100%) — The model correctly predicted the labels for all test samples, meaning it made zero classification errors.
- **Precision**: 1.0000 (100%) — Of all the positive predictions made by the model, 100% were correct. There were no false positives.
- **Recall**: 1.0000 (100%) — The model identified all actual positive instances correctly. There were no false negatives.
- **F1-Score**: 1.0000 (100%) — Since both precision and recall are perfect, the F1-score, which balances both, is also 100%.

Conclusion:

- The model performed extremely well on the Iris dataset, achieving **perfect test accuracy** and very low loss values.
- There is no sign of overfitting, as the model performs equally well on both the training and test sets.

[29]
```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import torch
import torch.nn as nn
import torch.optim as optim

# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Labels

# Split into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Scale the features (standardization)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Convert to PyTorch tensors
X_train_tensor = torch.FloatTensor(X_train)
y_train_tensor = torch.LongTensor(y_train)
X_test_tensor = torch.FloatTensor(X_test)
y_test_tensor = torch.LongTensor(y_test)

# Create TensorDatasets and DataLoaders
train_dataset = torch.utils.data.TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = torch.utils.data.TensorDataset(X_test_tensor, y_test_tensor)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32)
```

[26]
```
class IrisModel(nn.Module):
    def __init__(self):
        super(IrisModel, self).__init__()
        # Input size is 4 (features of Iris dataset), output size is 3 (classes)
        self.fc1 = nn.Linear(4, 128)  # 4 input features
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 32)
        self.fc4 = nn.Linear(32, 3)  # 3 output classes

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        x = self.fc4(x)
        return x
```

```python
[27] # Initialize model, optimizer, and loss function
     model = IrisModel()
     optimizer = optim.Adam(model.parameters(), lr=1e-3)
     criterion = nn.CrossEntropyLoss()

     # Training loop
     num_epochs = 100

     for epoch in range(num_epochs):
         model.train()
         epoch_loss = 0
         epoch_acc = 0
         for X_batch, y_batch in train_loader:
             optimizer.zero_grad()
             outputs = model(X_batch)
             loss = criterion(outputs, y_batch)
             loss.backward()
             optimizer.step()

             epoch_loss += loss.item()

             # Calculate accuracy
             _, preds = torch.max(outputs, 1)
             acc = torch.sum(preds == y_batch).item() / len(y_batch)
             epoch_acc += acc

         print(f'Epoch {epoch+1}/{num_epochs} | Loss: {epoch_loss/len(train_loader):.4f} | Accuracy: {epoch_acc/len(train_loader):.4f}')
```

```python
# Evaluate on test data
model.eval()
test_loss = 0
test_acc = 0
with torch.no_grad():
    for X_batch, y_batch in test_loader:
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
        test_loss += loss.item()

        # Calculate accuracy
        _, preds = torch.max(outputs, 1)
        acc = torch.sum(preds == y_batch).item() / len(y_batch)
        test_acc += acc

print(f'Test Loss: {test_loss/len(test_loader):.4f} | Test Accuracy: {test_acc/len(test_loader):.4f}')
```

[31]
```python
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score
```

```python
# Evaluate on the test data
model.eval()  # Set the model to evaluation mode
test_loss = 0
all_preds = []
all_labels = []

with torch.no_grad():  # Disable gradient calculation for faster evaluation
    for X_batch, y_batch in test_loader:
        outputs = model(X_batch)
        loss = criterion(outputs, y_batch)
        test_loss += loss.item()

        # Get the predicted class labels
        _, preds = torch.max(outputs, 1)

        # Store the predictions and true labels for metric calculations
        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(y_batch.cpu().numpy())

# Compute accuracy, precision, recall, and F1-score using the collected predictions and true labels
accuracy = accuracy_score(all_labels, all_preds)
precision = precision_score(all_labels, all_preds, average='weighted')
recall = recall_score(all_labels, all_preds, average='weighted')
f1 = f1_score(all_labels, all_preds, average='weighted')

print(f'Test Loss: {test_loss/len(test_loader):.4f}')
print(f'Accuracy: {accuracy:.4f}')
print(f'Precision: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1-Score: {f1:.4f}')
```

```
Test Loss: 0.0059
Accuracy: 1.0000
Precision: 1.0000
Recall: 1.0000
F1-Score: 1.0000
```