Model Script:

```
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from torch.utils.data import DataLoader, TensorDataset

# Set random seeds for reproducibility
np.random.seed(37)
torch.manual_seed(37)
torch.cuda.manual_seed(37)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

# Load the data
X = np.load('X.npy')
Y = np.load('Y.npy')

# Convert to PyTorch tensors
X = torch.tensor(X, dtype=torch.float32)
Y = torch.tensor(Y, dtype=torch.float32)

# Split the data
X_train, X_valtest, Y_train, Y_valtest = train_test_split(X, Y, test_size=0.2, random_state=37)
X_val, X_test, Y_val, Y_test = train_test_split(X_valtest, Y_valtest, test_size=0.5,
random_state=37)

print("Shape of X_train:", X_train.shape)
print("Shape of X_test:", X_test.shape)
print("Shape of Y_train:", Y_train.shape)
print("Shape of Y_test:", Y_test.shape)

# Define the model
class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.dropout = nn.Dropout(0.6)
        self.flatten = nn.Flatten()
        self.dense = nn.Linear(hidden_size * 99, output_size)  # 99 is the sequence length
        self.softmax = nn.Softmax(dim=1)
    def forward(self, x):
        x, _ = self.lstm(x)
        x = self.dropout(x)
        x = self.flatten(x)
        x = self.dense(x)
        x = self.softmax(x)
        return x

num_rows = X_train.shape[1]
num_columns = X_train.shape[2]
num_labels = Y_train.shape[1]

model = LSTMModel(input_size=num_columns, hidden_size=416, output_size=num_labels)

# Loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0008711452772441899)

# DataLoader
train_dataset = TensorDataset(X_train, Y_train)
test_dataset = TensorDataset(X_test, Y_test)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

# Training loop
num_epochs = 29

for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels.argmax(dim=1))
        loss.backward()
        optimizer.step()
        running_loss += loss.item()

    print(f'Epoch {epoch+1}/{num_epochs}, Loss: {running_loss/len(train_loader)}')

# Evaluate the model
def evaluate(model, data_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in data_loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels.argmax(dim=1)).sum().item()
    return 100 * correct / total

train_accuracy = evaluate(model, train_loader)
print(f'Training Accuracy: {train_accuracy}%')

test_accuracy = evaluate(model, test_loader)
print(f'Testing Accuracy: {test_accuracy}%')

# Save the model's state dictionary
torch.save(model.state_dict(), 'trained_model.pth')
print("Model state dictionary saved as 'trained_model.pth'")

# JIT trace test
model.eval()
dummy_input = torch.randn(1, num_rows, num_columns)
traced_model = torch.jit.trace(model, dummy_input)
torch.jit.save(traced_model, 'traced_model.pt')
print("JIT trace test passed and model saved as 'traced_model.pt'")
```