

✓ Homework 4 - MLPs and Regularization

In this assignment, you will analyze the effect of regularization techniques (L2 regularization and dropout) on the performance of MLP models built using PyTorch. Let's get started.

Task 1: Load Dataset


We will use the Breast Cancer dataset from scikit-learn. Let's load and preprocess this dataset.

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np
import matplotlib.pyplot as plt

# Load the Breast Cancer dataset
dataset = load_breast_cancer()

# Prepare the data
X, y = dataset.data, dataset.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Example data size
print(X_train.shape, X_test.shape)
```

 (398, 30) (171, 30)

✓ Task 2: Implement a Simple MLP with PyTorch

This week we will implement MLP with two layers (one hidden layer) using PyTorch. PyTorch's implementation already supports training with and without L2 regularization and dropout.

```
class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, dropout_rate=0.0):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(dropout_rate)
        self.fc2 = nn.Linear(hidden_dim, output_dim)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu(out)
        out = self.dropout(out)
        out = self.fc2(out)
        out = self.sigmoid(out)
        return out
```

✓ Task 3: Train the MLP with PyTorch

Implement the training loop for the MLP with options for L2 regularization and dropout.

```
# Define the training function
def train_model(model, criterion, optimizer, train_loader, num_epochs=25):
    model.train() # Set the model to training mode
    for epoch in range(num_epochs):
        running_loss = 0.0
        for inputs, labels in train_loader:
```

```

inputs, labels = inputs.float(), labels.float().view(-1, 1) # Convert data to the appropriate type and shape
optimizer.zero_grad() # Zero the parameter gradients
outputs = model(inputs) # Forward pass
loss = criterion(outputs, labels) # Compute loss
loss.backward() # Backward pass
optimizer.step() # Optimize the parameters
running_loss += loss.item()
# print(f"Epoch {epoch+1}/{num_epochs}, Loss: {running_loss/len(train_loader)}")
return model

```

✓ Your Tasks

1. Complete the DataLoader creation: Use TensorDataset and DataLoader to create loaders for the training and test datasets.
2. Instantiate the MLP model: Define the model using appropriate dimensions for input, hidden, and output layers.
3. Define the loss function and optimizer: Use BCELoss for binary classification and Adam optimizer with L2 regularization.
4. Train the model with and without dropout: Modify the dropout_rate and observe the performance differences.
5. Evaluate the model: Implement an evaluation function to calculate accuracy on the test dataset.

Your code starts here

```

# 1. Create DataLoaders (0.5 pt)
# Use TensorDataset to convert the training and test data into a dataset format
# Use DataLoader to create batches of data for training and testing
batch_size = 32
train_loader = DataLoader(TensorDataset(torch.tensor(X_train), torch.tensor(y_train)), batch_size=batch_size, shuffle=True)
test_loader = DataLoader(TensorDataset(torch.tensor(X_test), torch.tensor(y_test)), batch_size=batch_size, shuffle=False)

```

```

# 2. Instantiate the model (0.5 pt)
# Define the input dimension based on the number of features in the dataset
# Define the hidden dimension and output dimension (binary classification, so output_dim = 1)
input_dim = X_train.shape[1]
hidden_dim = 50
output_dim = 1
dropout_rate = 0.50
model = MLP(input_dim, hidden_dim, output_dim, dropout_rate)

```

```

# 3. Define loss function and optimizer (0.5 pt)
# Use BCELoss (binary cross entropy) from torch.nn for binary classification
# Use Adam optimizer (instead of a fixed learning rate) and include weight_decay parameter for L2 regularization
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=0.01)

```

```

# 4. Train the model (0.5 pt)
# Train the model using the train_model function defined above
num_epochs = 25
trained_model = train_model(model, criterion, optimizer, train_loader, num_epochs)

```

```

# 5. Evaluate the model (1 pt)
# Define an evaluation function to compute the accuracy of the model on the test data
def evaluate_model(model, test_loader):
    model.eval() # Set the model to evaluation mode
    correct = 0
    total = 0
    with torch.no_grad(): # Disable gradient computation
        for inputs, labels in test_loader:
            inputs, labels = inputs.float(), labels.float().view(-1, 1) # Convert data to the appropriate type and shape
            outputs = model(inputs) # Forward pass
            predicted = (outputs > 0.5).float() # Apply a threshold to get binary predictions (0 or 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    print(f"Accuracy: {accuracy:.2f}%")

```

```

# evaluate_model(trained_model, test_loader)

```

```

# 6. Change the above code to experiment with different dropout rates in the range from 0 to 1,
# also experiment with different L2 regularization parameters (weight_decay), e.g., 0, 0.0001, 0.01, 0.1, 1, 10, etc. ,
# and summarize your findings. (1 pt)

```

```

dropout_rates = [0.0, 0.2, 0.5, 0.8]

```

```

dropout_rates = [0.0, 0.1, 0.5, 0.8]
l2_values = [0, 0.0001, 0.01, 0.1, 1, 10]

for dr in dropout_rates:
    for l2 in l2_values:
        print(f"\nTraining with Dropout={dr} and L2 Regularization={l2}")
        model = MLP(input_dim, hidden_dim, output_dim, dropout_rate=dr)
        optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=l2)

        trained_model = train_model(model, criterion, optimizer, train_loader, num_epochs)
        evaluate_model(trained_model, test_loader)

```



Training with Dropout=0.0 and L2 Regularization=0
Accuracy: 98.25%

Training with Dropout=0.0 and L2 Regularization=0.0001
Accuracy: 99.42%

Training with Dropout=0.0 and L2 Regularization=0.01
Accuracy: 98.83%

Training with Dropout=0.0 and L2 Regularization=0.1
Accuracy: 98.83%

Training with Dropout=0.0 and L2 Regularization=1
Accuracy: 63.16%

Training with Dropout=0.0 and L2 Regularization=10
Accuracy: 63.16%

Training with Dropout=0.2 and L2 Regularization=0
Accuracy: 99.42%

Training with Dropout=0.2 and L2 Regularization=0.0001
Accuracy: 99.42%

Training with Dropout=0.2 and L2 Regularization=0.01
Accuracy: 99.42%

Training with Dropout=0.2 and L2 Regularization=0.1
Accuracy: 98.83%

Training with Dropout=0.2 and L2 Regularization=1
Accuracy: 63.16%

Training with Dropout=0.2 and L2 Regularization=10
Accuracy: 63.16%

Training with Dropout=0.5 and L2 Regularization=0
Accuracy: 99.42%

Training with Dropout=0.5 and L2 Regularization=0.0001
Accuracy: 98.25%

Training with Dropout=0.5 and L2 Regularization=0.01
Accuracy: 99.42%

Training with Dropout=0.5 and L2 Regularization=0.1
Accuracy: 98.25%

Training with Dropout=0.5 and L2 Regularization=1
Accuracy: 63.16%

Training with Dropout=0.5 and L2 Regularization=10
Accuracy: 63.16%

Training with Dropout=0.8 and L2 Regularization=0
Accuracy: 99.42%

Findings

Most models had accuracy over 98%, when the weight decay was 1 or 10 then the model performed significantly worse. The high values led to underfitting. The rest of the models were overfitted, most likely because we are using a toy dataset.