homework6

November 27, 2024

[1]:
```
#Problem 1a
##
import pandas as pd
import numpy as np
import torch
import torch.nn as nn
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import time

# Load and preprocess the data
data = pd.read_csv('assets/Housing.csv')
X = data.drop('price', axis=1)
y = data['price']

# Convert categorical variables to numeric
X = pd.get_dummies(X, columns=['mainroad', 'guestroom', 'basement',␣
 ↪'hotwaterheating', 'airconditioning', 'prefarea', 'furnishingstatus'])

# Scale the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
y_scaled = scaler.fit_transform(y.values.reshape(-1, 1))

# Split the data
X_train, X_val, y_train, y_val = train_test_split(X_scaled, y_scaled,␣
 ↪test_size=0.2, random_state=42)

# Convert to PyTorch tensors
X_train = torch.FloatTensor(X_train)
y_train = torch.FloatTensor(y_train)
X_val = torch.FloatTensor(X_val)
y_val = torch.FloatTensor(y_val)

#send tensors to gpu
X_train = X_train.cuda()
y_train = y_train.cuda()
```

```
X_val = X_val.cuda()
y_val = y_val.cuda()

# Define the neural network
class HousingNN(nn.Module):
    def __init__(self, input_size):
        super(HousingNN, self).__init__()
        self.layer1 = nn.Linear(input_size, 8)  # Hidden layer with 8 nodes
        self.relu = nn.ReLU()
        self.layer2 = nn.Linear(8, 1)  # Output layer

    def forward(self, x):
        x = self.layer1(x)
        x = self.relu(x)
        x = self.layer2(x)
        return x

# Initialize the model
input_size = X_train.shape[1]
model = HousingNN(input_size)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

#send model to gpu
model = model.cuda()

# Training loop
num_epochs = 100
start_time = time.time()

for epoch in range(num_epochs):
    # Forward pass
    outputs = model(X_train)
    loss = criterion(outputs, y_train)

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

training_time = time.time() - start_time

# Evaluation
model.eval()
```

```
with torch.no_grad():
    # Training set predictions
    train_outputs = model(X_train)
    train_loss = criterion(train_outputs, y_train)

    # Validation set predictions
    val_outputs = model(X_val)
    val_loss = criterion(val_outputs, y_val)

    # Calculate metrics for training set
    train_pred = train_outputs.cpu().numpy()
    train_true = y_train.cpu().numpy()
    train_r2 = 1 - np.sum((train_true - train_pred) ** 2) / np.sum((train_true␣
 ↪- train_true.mean()) ** 2)
    train_mae = np.mean(np.abs(train_true - train_pred))
    train_rmse = np.sqrt(np.mean((train_true - train_pred) ** 2))

    # Calculate metrics for validation set
    val_pred = val_outputs.cpu().numpy()
    val_true = y_val.cpu().numpy()
    val_r2 = 1 - np.sum((val_true - val_pred) ** 2) / np.sum((val_true -␣
 ↪val_true.mean()) ** 2)
    val_mae = np.mean(np.abs(val_true - val_pred))
    val_rmse = np.sqrt(np.mean((val_true - val_pred) ** 2))

print(f'\nTraining Time: {training_time:.2f} seconds')
print('\nTraining Set Metrics:')
print(f'MSE Loss: {train_loss.item():.4f}')
print(f'R-squared: {train_r2:.4f}')
print(f'MAE: {train_mae:.4f}')
print(f'RMSE: {train_rmse:.4f}')

print('\nValidation Set Metrics:')
print(f'MSE Loss: {val_loss.item():.4f}')
print(f'R-squared: {val_r2:.4f}')
print(f'MAE: {val_mae:.4f}')
print(f'RMSE: {val_rmse:.4f}')
```

```
Epoch [10/100], Loss: 0.4009
Epoch [20/100], Loss: 0.3179
Epoch [30/100], Loss: 0.2617
Epoch [40/100], Loss: 0.2475
Epoch [50/100], Loss: 0.2391
Epoch [60/100], Loss: 0.2336
Epoch [70/100], Loss: 0.2288
Epoch [80/100], Loss: 0.2241
Epoch [90/100], Loss: 0.2202
```

```
Epoch [100/100], Loss: 0.2170

Training Time: 0.28 seconds

Training Set Metrics:
MSE Loss: 0.2167
R-squared: 0.7546
MAE: 0.3440
RMSE: 0.4655

Validation Set Metrics:
MSE Loss: 0.4974
R-squared: 0.6564
MAE: 0.5151
RMSE: 0.7053
```

[2]:
```python
## Problem 1b
##

import pandas as pd
import numpy as np
import torch
import torch.nn as nn
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import time
import matplotlib.pyplot as plt

# Load and preprocess the data
data = pd.read_csv('assets/Housing.csv')
X = data.drop('price', axis=1)
y = data['price']

# Convert categorical variables to numeric
X = pd.get_dummies(X, columns=['mainroad', 'guestroom', 'basement',␣
 ↪'hotwaterheating', 'airconditioning', 'prefarea', 'furnishingstatus'])

# Scale the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
y_scaled = scaler.fit_transform(y.values.reshape(-1, 1))

# Split the data
X_train, X_val, y_train, y_val = train_test_split(X_scaled, y_scaled,␣
 ↪test_size=0.2, random_state=42)

# Convert to PyTorch tensors
```

```
X_train = torch.FloatTensor(X_train)
y_train = torch.FloatTensor(y_train)
X_val = torch.FloatTensor(X_val)
y_val = torch.FloatTensor(y_val)

#send tensors to gpu
X_train = X_train.cuda()
y_train = y_train.cuda()
X_val = X_val.cuda()
y_val = y_val.cuda()

# Define the neural network
class HousingNN(nn.Module):
    def __init__(self, input_size):
        super(HousingNN, self).__init__()
        # Layers
        self.layer1 = nn.Sequential(
            nn.Linear(input_size, 32),
            nn.ReLU(),
            nn.Dropout(0.2)
        )
        self.layer2 = nn.Sequential(
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Dropout(0.2)
        )
        self.layer3 = nn.Sequential(
            nn.Linear(16, 8),
            nn.ReLU(),
            nn.Dropout(0.2)
        )
        self.output_layer = nn.Linear(8, 1)  # No activation for regression␣
 ↪output

    def forward(self, x):
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.output_layer(x)
        return x

# Initialize the model
input_size = X_train.shape[1]
model = HousingNN(input_size)
criterion = nn.MSELoss()
# Add weight decay (L2 regularization) to the optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=0.01)
```

```
#send model to gpu
model = model.cuda()

# Before training loop, add lists to store losses
train_losses = []
val_losses = []

# Training loop
num_epochs = 1000
start_time = time.time()

for epoch in range(num_epochs):
    # Forward pass
    outputs = model(X_train)

    # Calculate the main loss
    main_loss = criterion(outputs, y_train)

    # Add L2 regularization term
    l2_lambda = 0.01  # Regularization strength
    l2_reg = torch.tensor(0., requires_grad=True).cuda()
    for param in model.parameters():
        l2_reg = l2_reg + torch.norm(param, 2)

    # Combined loss with regularization
    loss = main_loss + l2_lambda * l2_reg

    # Backward pass and optimization
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    # Calculate validation loss
    with torch.no_grad():
        val_outputs = model(X_val)
        val_loss = criterion(val_outputs, y_val)

    # Store losses
    train_losses.append(loss.item())
    val_losses.append(val_loss.item())

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Train Loss: {loss.item():.4f},␣
 ↪Val Loss: {val_loss.item():.4f}')

training_time = time.time() - start_time
```

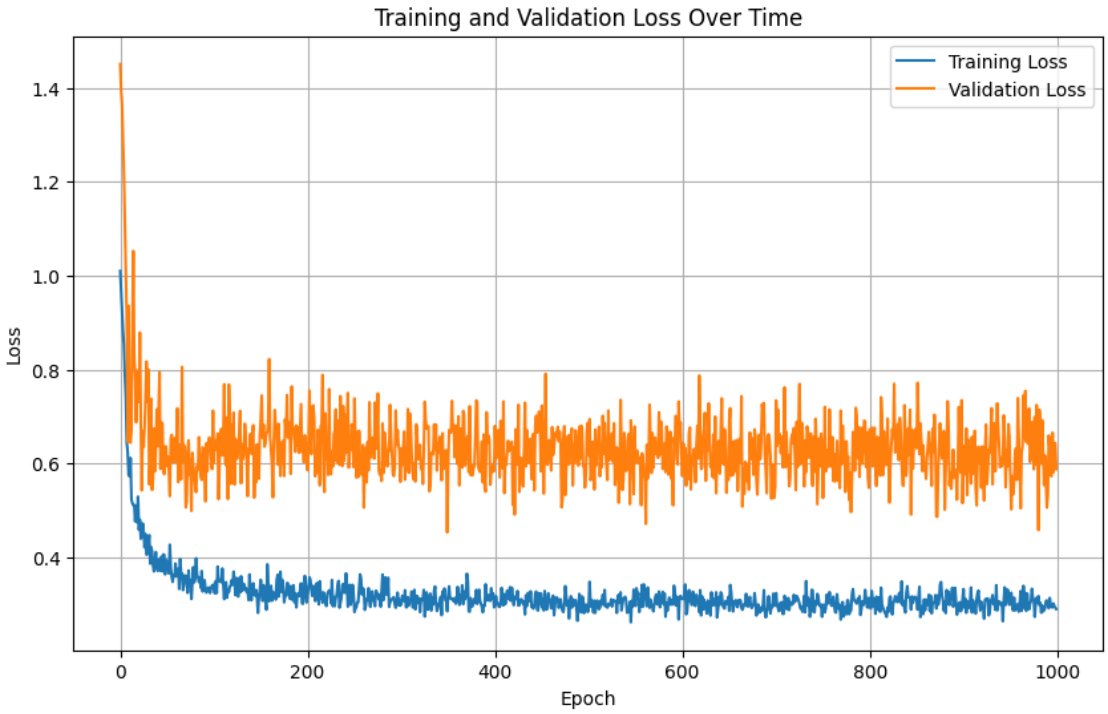
```
# Evaluation
model.eval()
with torch.no_grad():
    # Training set predictions
    train_outputs = model(X_train)
    train_loss = criterion(train_outputs, y_train)

    # Validation set predictions
    val_outputs = model(X_val)
    val_loss = criterion(val_outputs, y_val)

    # Calculate metrics for training set
    train_pred = train_outputs.cpu().numpy()
    train_true = y_train.cpu().numpy()
    train_r2 = 1 - np.sum((train_true - train_pred) ** 2) / np.sum((train_true␣
 ↪- train_true.mean()) ** 2)
    train_mae = np.mean(np.abs(train_true - train_pred))
    train_rmse = np.sqrt(np.mean((train_true - train_pred) ** 2))

    # Calculate metrics for validation set
    val_pred = val_outputs.cpu().numpy()
    val_true = y_val.cpu().numpy()
    val_r2 = 1 - np.sum((val_true - val_pred) ** 2) / np.sum((val_true -␣
 ↪val_true.mean()) ** 2)
    val_mae = np.mean(np.abs(val_true - val_pred))
    val_rmse = np.sqrt(np.mean((val_true - val_pred) ** 2))

print(f'\nTraining Time: {training_time:.2f} seconds')
print('\nTraining Set Metrics:')
print(f'MSE Loss: {train_loss.item():.4f}')
print(f'R-squared: {train_r2:.4f}')
print(f'MAE: {train_mae:.4f}')
print(f'RMSE: {train_rmse:.4f}')

print('\nValidation Set Metrics:')
print(f'MSE Loss: {val_loss.item():.4f}')
print(f'R-squared: {val_r2:.4f}')
print(f'MAE: {val_mae:.4f}')
print(f'RMSE: {val_rmse:.4f}')

# Plot training and validation losses
plt.figure(figsize=(10, 6))
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
```

```
plt.title('Training and Validation Loss Over Time')
plt.legend()
plt.grid(True)
plt.show()
```

```
Epoch [10/1000], Train Loss: 0.5757, Val Loss: 0.9357
Epoch [20/1000], Train Loss: 0.5296, Val Loss: 0.7549
Epoch [30/1000], Train Loss: 0.4073, Val Loss: 0.7314
Epoch [40/1000], Train Loss: 0.3833, Val Loss: 0.7153
Epoch [50/1000], Train Loss: 0.3948, Val Loss: 0.6362
Epoch [60/1000], Train Loss: 0.3872, Val Loss: 0.6032
Epoch [70/1000], Train Loss: 0.3620, Val Loss: 0.6250

Epoch [80/1000], Train Loss: 0.3471, Val Loss: 0.5828
Epoch [90/1000], Train Loss: 0.3472, Val Loss: 0.6295
Epoch [100/1000], Train Loss: 0.3422, Val Loss: 0.7127
Epoch [110/1000], Train Loss: 0.3767, Val Loss: 0.6176
Epoch [120/1000], Train Loss: 0.3520, Val Loss: 0.6833
Epoch [130/1000], Train Loss: 0.3602, Val Loss: 0.5580
Epoch [140/1000], Train Loss: 0.3287, Val Loss: 0.6629

Epoch [150/1000], Train Loss: 0.3308, Val Loss: 0.6590
Epoch [160/1000], Train Loss: 0.3068, Val Loss: 0.8224
Epoch [170/1000], Train Loss: 0.3169, Val Loss: 0.6845
Epoch [180/1000], Train Loss: 0.3452, Val Loss: 0.6554
Epoch [190/1000], Train Loss: 0.3187, Val Loss: 0.6388
Epoch [200/1000], Train Loss: 0.3043, Val Loss: 0.6422
Epoch [210/1000], Train Loss: 0.3292, Val Loss: 0.6160

Epoch [220/1000], Train Loss: 0.3079, Val Loss: 0.7068
Epoch [230/1000], Train Loss: 0.2967, Val Loss: 0.6704
Epoch [240/1000], Train Loss: 0.3253, Val Loss: 0.6055
Epoch [250/1000], Train Loss: 0.3230, Val Loss: 0.5924
Epoch [260/1000], Train Loss: 0.3343, Val Loss: 0.6396
Epoch [270/1000], Train Loss: 0.3280, Val Loss: 0.6549
Epoch [280/1000], Train Loss: 0.3284, Val Loss: 0.5630

Epoch [290/1000], Train Loss: 0.3080, Val Loss: 0.6617
Epoch [300/1000], Train Loss: 0.2977, Val Loss: 0.5612
Epoch [310/1000], Train Loss: 0.3071, Val Loss: 0.7068
Epoch [320/1000], Train Loss: 0.3131, Val Loss: 0.6234
Epoch [330/1000], Train Loss: 0.2946, Val Loss: 0.6794
Epoch [340/1000], Train Loss: 0.2911, Val Loss: 0.6416
Epoch [350/1000], Train Loss: 0.2857, Val Loss: 0.4536

Epoch [360/1000], Train Loss: 0.3079, Val Loss: 0.6810
Epoch [370/1000], Train Loss: 0.3195, Val Loss: 0.6182
Epoch [380/1000], Train Loss: 0.3037, Val Loss: 0.6555
Epoch [390/1000], Train Loss: 0.2970, Val Loss: 0.5501
Epoch [400/1000], Train Loss: 0.2941, Val Loss: 0.5839
```

```
Epoch [410/1000], Train Loss: 0.2981, Val Loss: 0.7321
Epoch [420/1000], Train Loss: 0.3184, Val Loss: 0.5121

Epoch [430/1000], Train Loss: 0.3080, Val Loss: 0.5805
Epoch [440/1000], Train Loss: 0.2947, Val Loss: 0.5667
Epoch [450/1000], Train Loss: 0.3237, Val Loss: 0.5836
Epoch [460/1000], Train Loss: 0.3107, Val Loss: 0.6223
Epoch [470/1000], Train Loss: 0.3011, Val Loss: 0.6681
Epoch [480/1000], Train Loss: 0.2702, Val Loss: 0.5984
Epoch [490/1000], Train Loss: 0.2914, Val Loss: 0.5956

Epoch [500/1000], Train Loss: 0.2840, Val Loss: 0.6863
Epoch [510/1000], Train Loss: 0.3087, Val Loss: 0.6370
Epoch [520/1000], Train Loss: 0.2962, Val Loss: 0.5922
Epoch [530/1000], Train Loss: 0.3072, Val Loss: 0.5848
Epoch [540/1000], Train Loss: 0.3166, Val Loss: 0.6491
Epoch [550/1000], Train Loss: 0.3096, Val Loss: 0.6787
Epoch [560/1000], Train Loss: 0.3427, Val Loss: 0.5856

Epoch [570/1000], Train Loss: 0.2854, Val Loss: 0.5913
Epoch [580/1000], Train Loss: 0.3271, Val Loss: 0.5886
Epoch [590/1000], Train Loss: 0.2996, Val Loss: 0.6017
Epoch [600/1000], Train Loss: 0.3118, Val Loss: 0.6187
Epoch [610/1000], Train Loss: 0.3214, Val Loss: 0.5449
Epoch [620/1000], Train Loss: 0.3306, Val Loss: 0.6860
Epoch [630/1000], Train Loss: 0.2872, Val Loss: 0.6486

Epoch [640/1000], Train Loss: 0.3161, Val Loss: 0.6484
Epoch [650/1000], Train Loss: 0.3160, Val Loss: 0.5652
Epoch [660/1000], Train Loss: 0.2946, Val Loss: 0.5416
Epoch [670/1000], Train Loss: 0.2969, Val Loss: 0.5627
Epoch [680/1000], Train Loss: 0.2766, Val Loss: 0.5633
Epoch [690/1000], Train Loss: 0.2726, Val Loss: 0.5854
Epoch [700/1000], Train Loss: 0.2981, Val Loss: 0.5497

Epoch [710/1000], Train Loss: 0.3186, Val Loss: 0.7623
Epoch [720/1000], Train Loss: 0.3208, Val Loss: 0.6757
Epoch [730/1000], Train Loss: 0.2915, Val Loss: 0.6291
Epoch [740/1000], Train Loss: 0.3304, Val Loss: 0.5526
Epoch [750/1000], Train Loss: 0.2741, Val Loss: 0.6294
Epoch [760/1000], Train Loss: 0.2974, Val Loss: 0.5935
Epoch [770/1000], Train Loss: 0.2680, Val Loss: 0.7125

Epoch [780/1000], Train Loss: 0.2876, Val Loss: 0.5801
Epoch [790/1000], Train Loss: 0.3178, Val Loss: 0.5716
Epoch [800/1000], Train Loss: 0.3142, Val Loss: 0.5485
Epoch [810/1000], Train Loss: 0.2885, Val Loss: 0.6509
Epoch [820/1000], Train Loss: 0.2938, Val Loss: 0.6958
Epoch [830/1000], Train Loss: 0.3053, Val Loss: 0.6307
Epoch [840/1000], Train Loss: 0.3256, Val Loss: 0.5740
```

```
Epoch [850/1000], Train Loss: 0.3156, Val Loss: 0.6745
Epoch [860/1000], Train Loss: 0.2876, Val Loss: 0.5999
Epoch [870/1000], Train Loss: 0.3063, Val Loss: 0.7039
Epoch [880/1000], Train Loss: 0.2870, Val Loss: 0.6692
Epoch [890/1000], Train Loss: 0.2953, Val Loss: 0.6295
Epoch [900/1000], Train Loss: 0.3098, Val Loss: 0.5158
Epoch [910/1000], Train Loss: 0.2829, Val Loss: 0.6568
Epoch [920/1000], Train Loss: 0.3061, Val Loss: 0.5486
Epoch [930/1000], Train Loss: 0.3079, Val Loss: 0.6671
Epoch [940/1000], Train Loss: 0.3035, Val Loss: 0.5920
Epoch [950/1000], Train Loss: 0.2994, Val Loss: 0.6578
Epoch [960/1000], Train Loss: 0.2847, Val Loss: 0.6582
Epoch [970/1000], Train Loss: 0.2990, Val Loss: 0.6420
Epoch [980/1000], Train Loss: 0.3043, Val Loss: 0.6580
Epoch [990/1000], Train Loss: 0.3088, Val Loss: 0.5063
Epoch [1000/1000], Train Loss: 0.2902, Val Loss: 0.5881

Training Time: 3.19 seconds

Training Set Metrics:
MSE Loss: 0.1462
R-squared: 0.8345
MAE: 0.2838
RMSE: 0.3823

Validation Set Metrics:
MSE Loss: 0.5241
R-squared: 0.6379
MAE: 0.5275
RMSE: 0.7240
```



[3]:
```
#problem 2a
##

import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import time

# Custom Dataset class
class BreastCancerDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.FloatTensor(X)
        self.y = torch.FloatTensor(y)

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]
```

```
# Neural Network class
class BinaryClassifier(nn.Module):
    def __init__(self, input_size):
        super(BinaryClassifier, self).__init__()
        self.layer1 = nn.Linear(input_size, 32)
        self.layer2 = nn.Linear(32, 1)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.relu(self.layer1(x))
        x = self.sigmoid(self.layer2(x))
        return x

# Load and prepare data
data = load_breast_cancer()
X = data.data
y = data.target

# Split the data
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# Create datasets and dataloaders
train_dataset = BreastCancerDataset(X_train_scaled, y_train)
val_dataset = BreastCancerDataset(X_val_scaled, y_val)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32)

# Initialize model, loss function, and optimizer
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = BinaryClassifier(input_size=30).to(device)
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters())

# Training
num_epochs = 1000
start_time = time.time()

for epoch in range(num_epochs):
    model.train()
    total_loss = 0
```

```
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs.squeeze(), labels)

        # Backward pass and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    # Print progress every 10 epochs
    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {total_loss/
 ↪len(train_loader):.4f}')

training_time = time.time() - start_time

# Evaluation
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    val_loss = 0
    for inputs, labels in val_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs.squeeze(), labels)

        val_loss += loss.item()

        predicted = (outputs.squeeze() > 0.5).float()
        correct += (predicted == labels).sum().item()
        total += labels.size(0)

    val_loss /= len(val_loader)
    val_accuracy = correct / total

print(f"\nTraining Time: {training_time:.2f} seconds")
print(f"Final Training Loss: {total_loss/len(train_loader):.4f}")
print(f"Validation Accuracy: {val_accuracy:.4f}")
```

```
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import time

# Custom Dataset class
class BreastCancerDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.FloatTensor(X)
        self.y = torch.FloatTensor(y)

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

# Neural Network class
class BinaryClassifier(nn.Module):
    def __init__(self, input_size):
        super(BinaryClassifier, self).__init__()
        self.layer1 = nn.Linear(input_size, 32)
        self.layer2 = nn.Linear(32, 1)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.relu(self.layer1(x))
        x = self.sigmoid(self.layer2(x))
        return x

# Load and prepare data
data = load_breast_cancer()
X = data.data
y = data.target

# Split the data
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# Create datasets and dataloaders
train_dataset = BreastCancerDataset(X_train_scaled, y_train)
val_dataset = BreastCancerDataset(X_val_scaled, y_val)
```

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32)

# Initialize model, loss function, and optimizer
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = BinaryClassifier(input_size=30).to(device)
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters())

# Training
num_epochs = 100
start_time = time.time()

for epoch in range(num_epochs):
    model.train()
    total_loss = 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs.squeeze(), labels)

        # Backward pass and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    # Print progress every 10 epochs
    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {total_loss/
 ↪len(train_loader):.4f}')

training_time = time.time() - start_time

# Evaluation
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    val_loss = 0
    for inputs, labels in val_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        # Forward pass
```

```
        outputs = model(inputs)
        loss = criterion(outputs.squeeze(), labels)

        val_loss += loss.item()

        predicted = (outputs.squeeze() > 0.5).float()
        correct += (predicted == labels).sum().item()
        total += labels.size(0)

    val_loss /= len(val_loader)
    val_accuracy = correct / total

print(f"\nTraining Time: {training_time:.2f} seconds")
print(f"Final Training Loss: {total_loss/len(train_loader):.4f}")
print(f"Validation Accuracy: {val_accuracy:.4f}")
```

```
Epoch [10/1000], Loss: 0.1305
Epoch [20/1000], Loss: 0.0839
Epoch [30/1000], Loss: 0.0598
Epoch [40/1000], Loss: 0.0490
Epoch [50/1000], Loss: 0.0464
Epoch [60/1000], Loss: 0.0369
Epoch [70/1000], Loss: 0.0344
Epoch [80/1000], Loss: 0.0322
Epoch [90/1000], Loss: 0.0248
Epoch [100/1000], Loss: 0.0291
Epoch [110/1000], Loss: 0.0250
Epoch [120/1000], Loss: 0.0149
Epoch [130/1000], Loss: 0.0132
Epoch [140/1000], Loss: 0.0110
Epoch [150/1000], Loss: 0.0101
Epoch [160/1000], Loss: 0.0082
Epoch [170/1000], Loss: 0.0081
Epoch [180/1000], Loss: 0.0063
Epoch [190/1000], Loss: 0.0055
Epoch [200/1000], Loss: 0.0050
Epoch [210/1000], Loss: 0.0043
Epoch [220/1000], Loss: 0.0039
Epoch [230/1000], Loss: 0.0057
Epoch [240/1000], Loss: 0.0032
Epoch [250/1000], Loss: 0.0028
Epoch [260/1000], Loss: 0.0025
```

```
Epoch [270/1000], Loss: 0.0022
Epoch [280/1000], Loss: 0.0020

Epoch [290/1000], Loss: 0.0019
Epoch [300/1000], Loss: 0.0016

Epoch [310/1000], Loss: 0.0015

Epoch [320/1000], Loss: 0.0013

Epoch [330/1000], Loss: 0.0019
Epoch [340/1000], Loss: 0.0011

Epoch [350/1000], Loss: 0.0010
Epoch [360/1000], Loss: 0.0009

Epoch [370/1000], Loss: 0.0009
Epoch [380/1000], Loss: 0.0008

Epoch [390/1000], Loss: 0.0007
Epoch [400/1000], Loss: 0.0006

Epoch [410/1000], Loss: 0.0006
Epoch [420/1000], Loss: 0.0005

Epoch [430/1000], Loss: 0.0005
Epoch [440/1000], Loss: 0.0004

Epoch [450/1000], Loss: 0.0004
Epoch [460/1000], Loss: 0.0003

Epoch [470/1000], Loss: 0.0006
Epoch [480/1000], Loss: 0.0003

Epoch [490/1000], Loss: 0.0003
Epoch [500/1000], Loss: 0.0002

Epoch [510/1000], Loss: 0.0002
Epoch [520/1000], Loss: 0.0002

Epoch [530/1000], Loss: 0.0002
Epoch [540/1000], Loss: 0.0002

Epoch [550/1000], Loss: 0.0002
Epoch [560/1000], Loss: 0.0001

Epoch [570/1000], Loss: 0.0001
Epoch [580/1000], Loss: 0.0001

Epoch [590/1000], Loss: 0.0001
Epoch [600/1000], Loss: 0.0002

Epoch [610/1000], Loss: 0.0001
Epoch [620/1000], Loss: 0.0001

Epoch [630/1000], Loss: 0.0001
Epoch [640/1000], Loss: 0.0001
```

```
Epoch [650/1000], Loss: 0.0001
Epoch [660/1000], Loss: 0.0001
Epoch [670/1000], Loss: 0.0001
Epoch [680/1000], Loss: 0.0001
Epoch [690/1000], Loss: 0.0000
Epoch [700/1000], Loss: 0.0000
Epoch [710/1000], Loss: 0.0000
Epoch [720/1000], Loss: 0.0000
Epoch [730/1000], Loss: 0.0000
Epoch [740/1000], Loss: 0.0000
Epoch [750/1000], Loss: 0.0000
Epoch [760/1000], Loss: 0.0000
Epoch [770/1000], Loss: 0.0000
Epoch [780/1000], Loss: 0.0000
Epoch [790/1000], Loss: 0.0000
Epoch [800/1000], Loss: 0.0000
Epoch [810/1000], Loss: 0.0000
Epoch [820/1000], Loss: 0.0000
Epoch [830/1000], Loss: 0.0000
Epoch [840/1000], Loss: 0.0000
Epoch [850/1000], Loss: 0.0000
Epoch [860/1000], Loss: 0.0000
Epoch [870/1000], Loss: 0.0000
Epoch [880/1000], Loss: 0.0000
Epoch [890/1000], Loss: 0.0000
Epoch [900/1000], Loss: 0.0000
Epoch [910/1000], Loss: 0.0000
Epoch [920/1000], Loss: 0.0000
Epoch [930/1000], Loss: 0.0000
Epoch [940/1000], Loss: 0.0000
Epoch [950/1000], Loss: 0.0000
Epoch [960/1000], Loss: 0.0000
Epoch [970/1000], Loss: 0.0000
Epoch [980/1000], Loss: 0.0000
Epoch [990/1000], Loss: 0.0000
```

```
Epoch [1000/1000], Loss: 0.0000

Training Time: 19.07 seconds
Final Training Loss: 0.0000
Validation Accuracy: 0.9737
Epoch [10/100], Loss: 0.1371
Epoch [20/100], Loss: 0.0756
Epoch [30/100], Loss: 0.0580
Epoch [40/100], Loss: 0.0470
Epoch [50/100], Loss: 0.0440
Epoch [60/100], Loss: 0.0354
Epoch [70/100], Loss: 0.0309
Epoch [80/100], Loss: 0.0270
Epoch [90/100], Loss: 0.0228
Epoch [100/100], Loss: 0.0215

Training Time: 2.01 seconds
Final Training Loss: 0.0215
Validation Accuracy: 0.9825
```

[4]:
```python
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
import time

# Custom Dataset class
class BreastCancerDataset(Dataset):
    def __init__(self, X, y):
        self.X = torch.FloatTensor(X)
        self.y = torch.FloatTensor(y)

    def __len__(self):
        return len(self.X)

    def __getitem__(self, idx):
        return self.X[idx], self.y[idx]

# Updated Neural Network class
class BinaryClassifier(nn.Module):
    def __init__(self, input_size):
```

```
        super(BinaryClassifier, self).__init__()
        self.layer1 = nn.Linear(input_size, 64)
        self.layer2 = nn.Linear(64, 32)
        self.layer3 = nn.Linear(32, 16)
        self.layer4 = nn.Linear(16, 8)
        self.layer5 = nn.Linear(8, 1)
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.relu(self.layer1(x))
        x = self.relu(self.layer2(x))
        x = self.relu(self.layer3(x))
        x = self.relu(self.layer4(x))
        x = self.sigmoid(self.layer5(x))
        return x

# Load and prepare data
data = load_breast_cancer()
X = data.data
y = data.target

# Split the data
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# Create datasets and dataloaders
train_dataset = BreastCancerDataset(X_train_scaled, y_train)
val_dataset = BreastCancerDataset(X_val_scaled, y_val)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=32)

# Initialize model, loss function, and optimizer
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = BinaryClassifier(input_size=30).to(device)
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters())

# Training
num_epochs = 100
start_time = time.time()
```

```
for epoch in range(num_epochs):
    model.train()
    total_loss = 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs.squeeze(), labels)

        # Backward pass and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    # Print progress every 10 epochs
    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {total_loss/
 ↪len(train_loader):.4f}')

training_time = time.time() - start_time

# Evaluation
model.eval()
with torch.no_grad():
    correct = 0
    total = 0
    val_loss = 0
    for inputs, labels in val_loader:
        inputs, labels = inputs.to(device), labels.to(device)

        # Forward pass
        outputs = model(inputs)
        loss = criterion(outputs.squeeze(), labels)

        val_loss += loss.item()

        predicted = (outputs.squeeze() > 0.5).float()
        correct += (predicted == labels).sum().item()
        total += labels.size(0)

    val_loss /= len(val_loader)
    val_accuracy = correct / total

print(f"\nTraining Time: {training_time:.2f} seconds")
```

```
print(f"Final Training Loss: {total_loss/len(train_loader):.4f}")
print(f"Validation Accuracy: {val_accuracy:.4f}")
```

```
Epoch [10/100], Loss: 0.0627
Epoch [20/100], Loss: 0.0361
Epoch [30/100], Loss: 0.0220
Epoch [40/100], Loss: 0.0101
Epoch [50/100], Loss: 0.0087
Epoch [60/100], Loss: 0.0039
Epoch [70/100], Loss: 0.0028
Epoch [80/100], Loss: 0.0015
Epoch [90/100], Loss: 0.0006
Epoch [100/100], Loss: 0.0003

Training Time: 2.73 seconds
Final Training Loss: 0.0003
Validation Accuracy: 0.9825
```

[5]:
```
#problem 3a
##

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
import time
import matplotlib.pyplot as plt

# Set random seed for reproducibility
torch.manual_seed(42)

# Define the neural network
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(3 * 32 * 32, 256)  # First and only hidden layer
        self.bn1 = nn.BatchNorm1d(256)          # Batch normalization
        self.fc2 = nn.Linear(256, 10)           # Output layer
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.2)
```

```
    def forward(self, x):
        x = self.flatten(x)
        x = self.relu(self.bn1(self.fc1(x)))
        x = self.dropout(x)
        x = self.fc2(x)
        return x

# Create separate transforms for training and testing
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load CIFAR-10 dataset
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=train_transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=test_transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64,
                                         shuffle=False, num_workers=2)

# Initialize the network and define loss function and optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
net = SimpleNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)

# Training loop
print(f"Training on {device}")
start_time = time.time()
losses = []

for epoch in range(100):
    net.train()  # Set network to training mode
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)
```

```
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    epoch_loss = running_loss / len(trainloader)
    losses.append(epoch_loss)

    # Evaluation phase
    net.eval()  # Set network to evaluation mode
    with torch.no_grad():
        correct = 0
        total = 0
        for data in testloader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    epoch_loss = running_loss / len(trainloader)
    epoch_acc = 100 * correct / total

    if (epoch + 1) % 10 == 0:
        print(f'Epoch {epoch + 1}, Loss: {epoch_loss:.3f}, Accuracy: {epoch_acc:
 ↪.2f}%')

training_time = time.time() - start_time
print(f'Finished Training. Total time: {training_time:.2f} seconds')

# Evaluation
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
print(f'Accuracy on test set: {accuracy:.2f}%')
```

```
# Plot training loss
plt.figure(figsize=(10, 5))
plt.plot(losses)
plt.title('Training Loss Over Time')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)
plt.savefig('training_loss.png')
plt.close()

# Save results to a file
with open('results.txt', 'w') as f:
    f.write(f'Training time: {training_time:.2f} seconds\n')
    f.write(f'Final training loss: {losses[-1]:.4f}\n')
    f.write(f'Test accuracy: {accuracy:.2f}%\n')
```

```
Files already downloaded and verified

Files already downloaded and verified

Training on cuda

Epoch 10, Loss: 1.366, Accuracy: 54.39%

Epoch 20, Loss: 1.285, Accuracy: 55.87%

Epoch 30, Loss: 1.249, Accuracy: 56.86%

Epoch 40, Loss: 1.206, Accuracy: 58.04%

Epoch 50, Loss: 1.190, Accuracy: 57.98%

Epoch 60, Loss: 1.170, Accuracy: 58.60%

Epoch 70, Loss: 1.156, Accuracy: 58.57%

Epoch 80, Loss: 1.145, Accuracy: 58.80%

Epoch 90, Loss: 1.126, Accuracy: 59.14%

Epoch 100, Loss: 1.116, Accuracy: 58.96%
Finished Training. Total time: 628.40 seconds

Accuracy on test set: 58.96%
```

[6]:
```
#problem 3b
##

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
```

```
import time
import matplotlib.pyplot as plt

# Set random seed for reproducibility
torch.manual_seed(42)

# Define the neural network
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(3 * 32 * 32, 512)  # Increased first layer size
        self.bn1 = nn.BatchNorm1d(512)         # Add batch normalization
        self.fc2 = nn.Linear(512, 256)           # New hidden layer
        self.bn2 = nn.BatchNorm1d(256)
        self.fc3 = nn.Linear(256, 128)           # New hidden layer
        self.bn3 = nn.BatchNorm1d(128)
        self.fc4 = nn.Linear(128, 10)            # Output layer
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.2)           # Added dropout to combat␣
 ↪overfitting

    def forward(self, x):
        x = self.flatten(x)
        x = self.relu(self.bn1(self.fc1(x)))
        x = self.dropout(x)
        x = self.relu(self.bn2(self.fc2(x)))
        x = self.dropout(x)
        x = self.relu(self.bn3(self.fc3(x)))
        x = self.dropout(x)
        x = self.fc4(x)
        return x

# Create separate transforms for training and testing
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load CIFAR-10 dataset
```

```
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=train_transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=64,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=test_transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=64,
                                         shuffle=False, num_workers=2)

# Initialize the network and define loss function and optimizer
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
net = SimpleNN().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(net.parameters(), lr=0.001)

# Training loop
print(f"Training on {device}")
start_time = time.time()
losses = []

for epoch in range(100):
    net.train()  # Set network to training mode
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data[0].to(device), data[1].to(device)

        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    epoch_loss = running_loss / len(trainloader)
    losses.append(epoch_loss)

    # Evaluation phase
    net.eval()  # Set network to evaluation mode
    with torch.no_grad():
        correct = 0
        total = 0
        for data in testloader:
            images, labels = data[0].to(device), data[1].to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
```

```
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    epoch_loss = running_loss / len(trainloader)
    epoch_acc = 100 * correct / total

    if (epoch + 1) % 10 == 0:
        print(f'Epoch {epoch + 1}, Loss: {epoch_loss:.3f}, Accuracy: {epoch_acc:
 ↪.2f}%')

training_time = time.time() - start_time
print(f'Finished Training. Total time: {training_time:.2f} seconds')

# Evaluation
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data[0].to(device), data[1].to(device)
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
print(f'Accuracy on test set: {accuracy:.2f}%')

# Plot training loss
plt.figure(figsize=(10, 5))
plt.plot(losses)
plt.title('Training Loss Over Time')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)
plt.savefig('training_loss.png')
plt.close()

# Save results to a file
with open('results.txt', 'w') as f:
    f.write(f'Training time: {training_time:.2f} seconds\n')
    f.write(f'Final training loss: {losses[-1]:.4f}\n')
    f.write(f'Test accuracy: {accuracy:.2f}%\n')
```

```
Files already downloaded and verified

Files already downloaded and verified

Training on cuda
```

```
Epoch 10, Loss: 1.292, Accuracy: 56.94%
Epoch 20, Loss: 1.166, Accuracy: 59.21%
Epoch 30, Loss: 1.096, Accuracy: 60.43%
Epoch 40, Loss: 1.039, Accuracy: 61.56%
Epoch 50, Loss: 1.003, Accuracy: 61.66%
Epoch 60, Loss: 0.962, Accuracy: 62.22%
Epoch 70, Loss: 0.933, Accuracy: 62.35%
Epoch 80, Loss: 0.908, Accuracy: 62.63%
Epoch 90, Loss: 0.890, Accuracy: 62.90%
Epoch 100, Loss: 0.878, Accuracy: 63.36%
Finished Training. Total time: 635.82 seconds
Accuracy on test set: 63.36%
```