

Homework 5 Python Code

```
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 sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score
import pandas as pd
import matplotlib.pyplot as plt
import time
```

Problem 1: Nonlinear Temperature Prediction

Data

```
t_c = np.array([0.5, 14.0, 15.0, 28.0, 11.0, 8.0, 3.0, -4.0, 6.0, 13.0, 21.0])
t_u = np.array([35.7, 55.9, 58.2, 81.9, 56.3, 48.9, 33.9, 21.8, 48.4, 60.4, 68.4])
t_un = 0.1 * t_u # Normalized
```

1.a Modified training loop for nonlinear model: $w_2 * t_u^{**2} + w_1 * t_u + b$

```
def model_nonlinear(t_u, w1, w2, b):
    return w2 * t_u ** 2 + w1 * t_u + b
```

```
def loss_fn(t_p, t_c):
    return ((t_p - t_c) ** 2).mean()
```

```
def train_nonlinear(n_epochs, lr, t_u, t_c):
    w1 = 0.0
    w2 = 0.0
    b = 0.0
    losses = []
    for epoch in range(1, n_epochs + 1):
        t_p = model_nonlinear(t_u, w1, w2, b)
        loss = loss_fn(t_p, t_c)
        losses.append(loss)
        grad_w1 = 2.0 * (t_p - t_c).dot(t_u) / t_u.size
        grad_w2 = 2.0 * (t_p - t_c).dot(t_u ** 2) / t_u.size
        grad_b = 2.0 * (t_p - t_c).mean()
        w1 -= lr * grad_w1
        w2 -= lr * grad_w2
        b -= lr * grad_b
    if epoch % 500 == 0:
```

```

print(f"Epoch {epoch}, Loss: {loss:.4f}")
return w1, w2, b, losses

# 1.b Train with different learning rates
learning_rates = [0.1, 0.01, 0.001, 0.0001]
for lr in learning_rates:
    print(f"\nTraining with LR = {lr}")
    train_nonlinear(5000, lr, t_un, t_c) # Use t_un for better convergence

# For 1.c, assuming best is lr=0.0001, compare with linear
# Linear model function
def model_linear(t_u, w, b):
    return w * t_u + b

def train_linear(n_epochs, lr, t_u, t_c):
    w = 0.0
    b = 0.0
    for epoch in range(1, n_epochs + 1):
        t_p = model_linear(t_u, w, b)
        loss = loss_fn(t_p, t_c)
        grad_w = 2.0 * (t_p - t_c).dot(t_u) / t_u.size
        grad_b = 2.0 * (t_p - t_c).mean()
        w -= lr * grad_w
        b -= lr * grad_b
    return w, b, loss

# Train linear with suitable lr (e.g., 0.01)
w_lin, b_lin, loss_lin = train_linear(5000, 0.01, t_un, t_c)
print(f"Linear final loss: {loss_lin:.4f}")

# Assuming nonlinear trained with lr=0.0001
w1_nl, w2_nl, b_nl, _ = train_nonlinear(5000, 0.0001, t_un, t_c)
loss_nl = loss_fn(model_nonlinear(t_un, w1_nl, w2_nl, b_nl), t_c)
print(f"Nonlinear final loss: {loss_nl:.4f}")

# Visualization
plt.scatter(t_u, t_c, label='Data')
t_us = np.linspace(t_u.min(), t_u.max(), 100)
t_uns = 0.1 * t_us
plt.plot(t_us, model_linear(t_uns, w_lin, b_lin), label='Linear')
plt.plot(t_us, model_nonlinear(t_uns, w1_nl, w2_nl, b_nl), label='Nonlinear')
plt.legend()
plt.show()

```

```

# Problem 2: Linear Regression on Housing Dataset

# 2.a Preprocessing and training
# Assume housing.csv with columns: price, area, bedrooms, bathrooms, stories, parking
df = pd.read_csv('housing.csv') # Replace with actual path
features = ['area', 'bedrooms', 'bathrooms', 'stories', 'parking']
X = df[features].values
y = df['price'].values

X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)

scaler_X = StandardScaler()
scaler_y = StandardScaler()
X_train = scaler_X.fit_transform(X_train)
X_val = scaler_X.transform(X_val)
y_train = scaler_y.fit_transform(y_train.reshape(-1, 1)).flatten()
y_val = scaler_y.transform(y_val.reshape(-1, 1)).flatten()

def model_lr(X, params):
    return np.dot(X, params[:-1]) + params[-1]

def loss_mse(y_pred, y_true):
    return np.mean((y_pred - y_true) ** 2)

def train_lr(n_epochs, lr, X_train, y_train, X_val, y_val):
    n_features = X_train.shape[1]
    params = np.zeros(n_features + 1) # W1 to W5, B
    for epoch in range(1, n_epochs + 1):
        y_pred_train = model_lr(X_train, params)
        loss_train = loss_mse(y_pred_train, y_train)
        grad = 2 * np.dot(X_train.T, (y_pred_train - y_train)) / len(y_train)
        grad_b = 2 * np.mean(y_pred_train - y_train)
        params[:-1] -= lr * grad
        params[-1] -= lr * grad_b
        if epoch % 500 == 0:
            y_pred_val = model_lr(X_val, params)
            loss_val = loss_mse(y_pred_val, y_val)
            r2_val = r2_score(y_val, y_pred_val)
            print(f"Epoch {epoch}, Train Loss: {loss_train:.4f}, Val R2: {r2_val:.4f}")
    return params

```

```

# 2.b Train with different LR
for lr in learning_rates:
    print(f"\nTraining LR with LR = {lr}")
    train_lr(5000, lr, X_train, y_train, X_val, y_val)

# Problem 3: Neural Networks on Housing

# 3.a One hidden layer NN
class NN1Hidden(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(NN1Hidden, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, 1)
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
    return x

def train_nn(model, n_epochs, lr, X_train, y_train, X_val, y_val):
    optimizer = optim.SGD(model.parameters(), lr=lr)
    criterion = nn.MSELoss()
    X_train_t = torch.tensor(X_train, dtype=torch.float32)
    y_train_t = torch.tensor(y_train, dtype=torch.float32).unsqueeze(1)
    X_val_t = torch.tensor(X_val, dtype=torch.float32)
    y_val_t = torch.tensor(y_val, dtype=torch.float32).unsqueeze(1)
    start_time = time.time()
    for epoch in range(n_epochs):
        model.train()
        optimizer.zero_grad()
        outputs = model(X_train_t)
        loss = criterion(outputs, y_train_t)
        loss.backward()
        optimizer.step()
        training_time = time.time() - start_time
        model.eval()
        with torch.no_grad():
            y_pred_train = model(X_train_t)
            train_loss = criterion(y_pred_train, y_train_t).item()
            y_pred_val = model(X_val_t)
            val_r2 = r2_score(y_val_t.numpy(), y_pred_val.numpy())
    print(f"Training time: {training_time:.2f}s, Train Loss: {train_loss:.4f}, Val R2: {val_r2:.4f}")
    return train_loss, val_r2

```

```
input_size = X_train.shape[1]
hidden_size = 8
model_1h = NN1Hidden(input_size, hidden_size)
train_nn(model_1h, 200, 0.01, X_train, y_train, X_val, y_val)
```

3.b Three hidden layers NN

```
class NN3Hidden(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(NN3Hidden, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, hidden_size)
        self.fc4 = nn.Linear(hidden_size, 1)
    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
```

```
model_3h = NN3Hidden(input_size, hidden_size)
train_nn(model_3h, 200, 0.01, X_train, y_train, X_val, y_val)
```