Artificial Intelligence for Sciences and Humanities

Homework 7: Hello Neural Networks!

DO NOT POLLUTE! AVOID PRINTING, OR PRINT 2-SIDED MULTIPAGE.

In this homework you will implement your first neural network. We will use the MNIST dataset, which contains 70,000 grayscale images of handwritten digits from 0 to 9. Each image is 28×28 pixels, so each image can be represented as a vector of length 784. The goal is to train your AI system (neural network) so that it is capable of identifying digits. To achieve this goal, you will have to:

- Load and prepare the data.
- Split the data into training, validation, and test sets.
- Construct a network (AI model) with 3 layers:
  - An input layer with 784 units.
  - One hidden layer with 64 units with *ReLU* activation functions.
  - One *softmax* output layer with 10 units.
- Define the loss (cross-entropy).
- Compute the gradients.
- Learn the optimal parameters using gradient descent.
- Evaluate the model.

You are allowed to use general libraries such as *numpy*, *matplotlib*, *sklearn.datasets*, or *sklearn.model_selection*. You are allowed to use *help* from modern AI systems. However, you are not allowed to have use an AI system to *solve* the homework for you. You are also not allowed to use neural-network libraries such as *TensorFlow*, *Keras*, or *PyTorch*. Each problem is worth 20 points.

Problem 1. Run the following code to install some useful libraries and load the MNIST dataset into (i) a $70,000 \times 784$ data matrix $\mathbf{X}$ where each row represents a vectorized digit image, with entries normalized between 0 and 1, and (ii) a $70,000 \times 1$ response vector $\mathbf{y}$ containing the corresponding labels as integer numbers.

```
mnist = fetch_openml('mnist_784', version=1, as_frame=False)

X = mnist.data
X = X.astype(np.float32) / 255.0
y = mnist.target.astype(int)

print("X shape:", X.shape)
print("y shape:", y.shape)
```

Problem 2. Run the following code to visualize a few samples

```
plt.figure(figsize=(12, 3))
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(X[i].reshape(28, 28), cmap='gray')
    plt.title(f"Label: {y[i]}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Problem 3. Run the following code to split your data into training (60%), testing (20%), and validation (20%).

```
X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.4, random_state=42
)

X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, random_state=42
)

print("X_train:", X_train.shape)
print("y_train:", y_train.shape)
print("X_val:", X_val.shape)
print("y_val:", y_val.shape)
print("X_test:", X_test.shape)
print("y_test:", y_test.shape)
```

Problem 4. Complete the following code to encode the responses as one-hot vectors:

```
def one_hot(y):
    Y = np.zeros((y.shape[0], 10))
    Y[np.arange(y.shape[0]), y] = 1
    return Y

Y_train = one_hot(___)
Y_val = one_hot(___)
Y_test = one_hot(___)
```

Problem 5. Complete the following code to specify the architecture of your neural network, as described above: 3 layers of sizes 784, 64, and 10.

```
input_size = ___
hidden_size = ___
output_size = ___
```

Problem 6. Complete the following code to initialize the parameters of your network with random numbers.

```
W1 = 0.01 * np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))

W2 = 0.01 * np.random.randn(___, ___)
b2 = np.zeros((1, ___))

print("W1:", W1.shape)
print("b1:", b1.shape)
print("W2:", W2.shape)
print("b2:", b2.shape)
```

Problem 7. Run the following code to implement your activation functions.

```
def relu(Z):
    return np.maximum(0, Z)

def relu_derivative(Z):
    return (Z > 0).astype(float)

def softmax(Z):
    Z_shifted = Z - np.max(Z, axis=1, keepdims=True)  # numerical stability
    exp_Z = np.exp(Z_shifted)
    return exp_Z / np.sum(exp_Z, axis=1, keepdims=True)
```

Problem 8. Run the following code to implement the model's function f.

```
def compute_f(X, W1, b1, W2, b2):
    Z1 = X @ W1 + b1
    A1 = relu(Z1)
    Z2 = A1 @ W2 + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2
```

Problem 9. Run the following code to implement the *cross-entropy* loss function ℓ.

```
def compute_loss(Y, A2):
    epsilon = 1e-12
    n = Y.shape[0]
    loss = -np.sum(Y * np.log(A2 + epsilon)) / n
    return loss
```

Problem 10. Run the following code to create a function that computes all gradients (derivatives).

```
def compute_gradients(X, Y, Z1, A1, A2, W2):
    n = X.shape[0]

    dZ2 = A2 - Y
    dW2 = (A1.T @ dZ2) / n
    db2 = np.sum(dZ2, axis=0, keepdims=True) / n

    dA1 = dZ2 @ W2.T
    dZ1 = dA1 * relu_derivative(Z1)
    dW1 = (X.T @ dZ1) / n
    db1 = np.sum(dZ1, axis=0, keepdims=True) / n

    return dW1, db1, dW2, db2
```

Problem 11. Complete the following function so that it does one gradient step of each parameter.

```
def gradient_step(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate):
    W1 = W1 - learning_rate * dW1
    b1 = ___
    W2 = ___
    b2 = ___
    return W1, b1, W2, b2
```

Problem 12. Complete the following code to perform the training process

```
learning_rate = ___
epochs = ___

train_losses = []
val_losses = []

for epoch in range(epochs):
    # Compute output on training data
    Z1, A1, Z2, A2 = compute_f(X_train, W1, b1, W2, b2)
    train_loss = compute_loss(Y_train, A2)

    # Compute gradients
    ___

    # Update parameters
    ___

    # Validation loss
    _, _, _, A2_val = compute_f(X_val, W1, b1, W2, b2)
    val_loss = compute_loss(Y_val, A2_val)

    train_losses.append(train_loss)
    val_losses.append(val_loss)

    if epoch % 5 == 0 or epoch == epochs - 1:
        print(f"Epoch {epoch+1}/{epochs} | Train loss: {train_loss:.4f} | Val loss: {
                                                    val_loss:.4f}")
```

Problem 13. Complete the following code to create auxiliary functions to compute accuracy.

```
def predict(X, W1, b1, W2, b2):
    _, _, _, A2 = compute_f(X, W1, b1, W2, b2)
    return np.argmax(A2, axis=1)

def accuracy(y_true, y_pred):
    return np.mean(y_true == y_pred)

train_pred = predict(X_train, W1, b1, W2, b2)
val_pred = predict(X_val, W1, b1, W2, b2)
test_pred = predict(X_test, W1, b1, W2, b2)

train_acc = accuracy(y_train, train_pred)
val_acc = accuracy(y_val, val_pred)
test_acc = accuracy(y_test, test_pred)

print("Training accuracy:", ___)
print("Validation accuracy:", ___)
print("Test accuracy:", ___)
```

Problem 14. Run the following code to plot the loss over the training process.

```
plt.figure(figsize=(8, 5))
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')
plt.legend()
plt.show()
```

Problem 15. Run the following code to use your AI system on a few test (unseen) samples.

```
sample_indices = np.arange(10)

plt.figure(figsize=(12, 3))
for i, idx in enumerate(sample_indices):
    plt.subplot(2, 5, i + 1)
    plt.imshow(X_test[idx].reshape(28, 28), cmap='gray')
    plt.title(f"T:{y_test[idx]} P:{test_pred[idx]}")
    plt.axis('off')
plt.tight_layout()
plt.show()
```

Problem 16. Did your system perform satisfactory? What proportion of images did the system classify correctly?

Problem 17. Based on the loss plot, can you tune the hyperparameters (e.g., step size or number of epochs) to improve the accuracy? What is the highest accuracy you can get?