Fabian Figueroa

Professor Stephen Pena

Applied Linear Algebra

May 12, 2025

Final Project

Written Portion:

(a). Diagram of my neural network architecture:

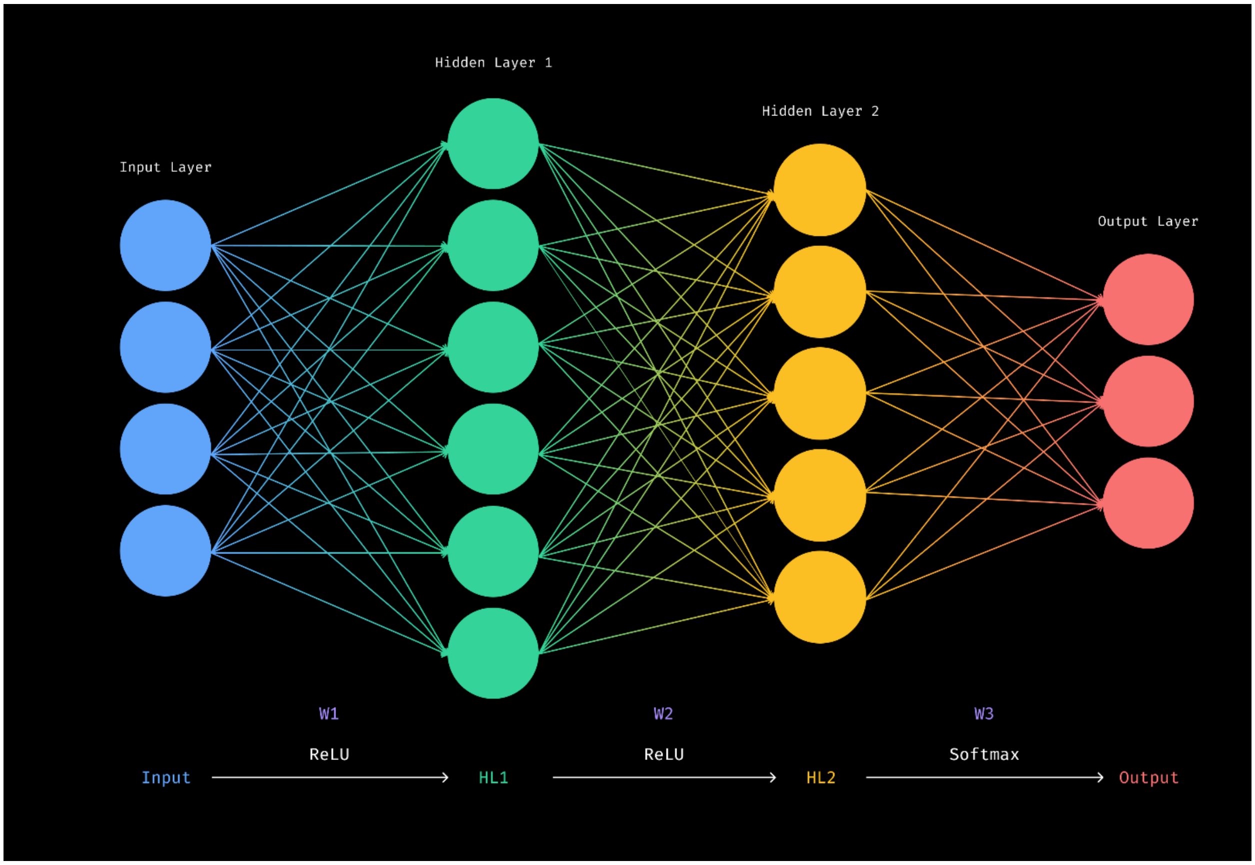

Figure 1: Network sketch created in Figma

(b) Forward Pass Computation:

Forward Pass Calculation

- $\vec{X} = [5.1,\ 3.5,\ 1.4,\ 0.2]$ – Raw data input

- Next, normalise the data, as is being done in my programme:

$$-X = \left[\frac{5.1-5.8433}{0.8253},\ \frac{3.5-3.0573}{0.4344},\ \frac{1.4-3.7580}{1.7594},\ \frac{0.2-1.1993}{0.7597}\right] \approx [-0.9006, 1.0191, -1.3402, -1.3154]$$

Then, continue with calculations:

$$W_1 = \begin{bmatrix} 0.5 & -0.14 & 0.65 & 1.52 & -0.23 & -0.23 \\ 1.58 & 0.77 & -0.47 & 0.54 & -0.46 & -0.47 \\ 0.24 & -1.91 & -1.72 & -0.56 & -1.01 & 0.31 \\ -0.91 & -1.41 & 1.47 & -0.23 & 0.07 & -1.42 \end{bmatrix}_{(4\times 6)} \quad b_1 = [-0.54,\ 0.11,\ -1.15,\ 0.38,\ -0.6,\ -0.29]_{(1\times 6)}$$

$$W^{(2)} = \begin{bmatrix} -0.60 & 1.85 & -0.01 & -1.06 & 0.82 \\ -1.22 & 0.21 & -1.96 & -1.33 & 0.20 \\ 0.74 & 0.17 & -0.12 & -0.30 & -1.48 \\ -0.72 & -0.46 & 1.06 & 0.34 & -1.76 \\ 0.32 & -0.39 & -0.68 & 0.61 & 1.03 \\ 0.93 & -0.84 & -0.31 & 0.33 & 0.98 \end{bmatrix}_{(6\times 5)} \quad b^{(2)} = [-0.48,\ -0.19,\ -1.11,\ -1.20,\ 0.81]_{(1\times 5)}$$

$$W^{(3)} = \begin{bmatrix} 1.36 & -0.07 & 1.00 \\ 0.36 & -0.65 & 0.36 \\ 1.54 & -0.04 & 1.56 \\ -2.62 & 0.82 & 0.09 \\ -0.30 & 0.09 & -1.99 \end{bmatrix}_{(5\times 3)} \quad b^{(3)} = [-0.22,\ 0.36,\ 1.48]_{(1\times 3)}$$

$$Z_m^n = W^m \cdot X + b_n$$

- $Z_1^{(0)} = W^{(0)} \cdot X + b_0 = -0.4503 + 1.6102 - 0.3216 + 1.1969 - 0.54 \approx 1.4952$

$Z_1^{(1)} = 0.1261 + 0.7847 + 2.5598 + 1.8547 + 0.1100 = 5.4353$

$Z_1^{(2)} = -1.8429$; $Z_1^{(3)} = 0.6145$; $Z_1^{(4)} = 0.3999$; $Z_1^{(5)} = 0.8906$

-Then, apply ReLU, $\max(0, z_n)$; where a_1 will equal:

$a_1 = [1.4952,\ 5.4353,\ 0.000,\ 0.6145,\ 0.3999,\ 0.8906]$

Hidden layer 1

- Then do the same for HL 2, solving for a_2:

$a_2 = [0.0000,\ 2.5309,\ 0.0000,\ 0.0000,\ 3.3263]$

Hidden layer 2

- And finally compute $\vec{y}$, the output layer, using softmax:

$y = \text{softmax}([-0.3068, -0.9857, -4.2283]) = [0.6549,\ 0.3321,\ 0.0130]$

Output layer

(c) Backpropagation Computation:

Backpropagation

- The desired derivation is $\frac{\partial L}{\partial W_3[1,0]}$. What this calculates is the affect on the loss from changing the weight from neuron 1 in Hidden Layer 2 to output 0.
- With the Chain Rule, the desired relation with loss and the weight is shown as: $\frac{\partial L}{\partial \hat{y}_0} \cdot \frac{\partial \hat{y}_0}{\partial z_3^{(0)}} \cdot \frac{\partial z_3^{(0)}}{\partial W_3[1,0]}$ where each term is:

$\frac{\partial L}{\partial \hat{y}_0}$: this is cross entropy loss with softmax; $L = -\sum_{y_i} \log(\hat{y}_i) \Rightarrow \frac{\partial L}{\partial \hat{y}_0} = -\frac{y_0}{\hat{y}_0} = \frac{1}{\hat{y}_0}$

$\frac{\partial \hat{y}_0}{\partial z_3^{(0)}}$: as softmax is being used, the derivative of softmax is its own input for the correct class (0) is: $\frac{\partial \hat{y}_0}{\partial z_3^{(0)}} = \hat{y}_0(1-\hat{y}_0)$

$\frac{\partial z_3^{(0)}}{\partial W_3[1,0]}$: This is just a dot product layer: $z_3^{(0)} = \sum W_3[i,0] \cdot a_2[i] + b_3[0] \Rightarrow \frac{\partial z_3^{(0)}}{\partial W_3[1,0]} = a_2[1]$

- Altogether: $\frac{\partial L}{\partial W_3[1,0]} = \left(-\frac{1}{\hat{y}_0}\right) \cdot \hat{y}_0(1-\hat{y}_0) \cdot a_2[1] = -(1-\hat{y}_0) \cdot a_2[1]$
  - Using my values for an example: $\hat{y}_0 = 0.6549$, $a_2[1] = 2.5309$
    - $\frac{\partial L}{\partial W_3[1,0]} = -(1-0.6549) \cdot 2.5309 = -0.3451 \cdot 2.5309 = -0.8734$

Coded Portion:

(a) Neural Network Implementation:

finalNeuralNetwork.py

```python
# Initialize weights and biases
W1 = np.random.randn(4, 6)
b1 = np.random.randn(1, 6)
W2 = np.random.randn(6, 5)
b2 = np.random.randn(1, 5)
W3 = np.random.randn(5, 3)
b3 = np.random.randn(1, 3)

# Activation functions
def relu(a):
    return np.maximum(0, a)

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

def softmax(a):
    exp_a = np.exp(a - np.max(a, axis=1, keepdims=True))
    return exp_a / np.sum(exp_a, axis=1, keepdims=True)

# Loss function: cross-entropy
def cross_entropy(y_pred, y_true):
    eps = 2e-10
    return -np.mean(np.sum(y_true * np.log(y_pred + eps), axis=1))

# Gradient of loss w.r.t. logits (output preactivation)
def dL_da3(y_pred, y_true):
    return y_pred - y_true
```

(b) Training:

finalNeuralNetwork.py

```
# Training loop
for epoch in range(10000):
    # Forward pass
    z1 = X_train @ W1 + b1  # preactivation hidden layer
    a1 = relu(z1)           # activation hidden layer
    z2 = a1 @ W2 + b2
    a2 = relu(z2)
    z3 = a2 @ W3 + b3       # preactivation output layer
    y_pred = softmax(z3)    # activation output layer (softmax)

    # Compute loss
    loss = cross_entropy(y_pred, y_train)

    # Backpropagation
    da3 = dL_da3(y_pred, y_train) # dL/da3
    dW3 = a2.T @ da3
    db3 = np.sum(da3, axis=0, keepdims=True)

    dz2 = da3 @ W3.T
    da2 = dz2 * relu_derivative(a2)
    dW2 = a1.T @ da2
    db2 = np.sum(da2, axis=0, keepdims=True)

    dz1 = da2 @ W2.T # dL/dz1
    da1 = dz1 * relu_derivative(a1) # dL/da1
    dW1 = X_train.T @ da1
    db1 = np.sum(da1, axis=0, keepdims=True)

    # Update parameters
    lr = 0.01 #Learning Rate
    W3 -= lr * dW3
    b3 -= lr * db3
    #Update W2
    W2 -= lr * dW2
    #Update b2
    b2 -= lr * db2
    #Update W1
    W1 -= lr * dW1
    #update b1
    b1 -= lr * db1

    # Print loss every 10 epochs
    if epoch % 1000 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")
```

(c) Evaluation:

```
Epoch 0, Loss: 0.9393
Epoch 1000, Loss: 0.0021
Epoch 2000, Loss: 0.0007
Epoch 3000, Loss: 0.0004
Epoch 4000, Loss: 0.0003
Epoch 5000, Loss: 0.0002
Epoch 6000, Loss: 0.0002
Epoch 7000, Loss: 0.0001
Epoch 8000, Loss: 0.0001
Epoch 9000, Loss: 0.0001
Predicted classes: [1 0 2 1 1]
Actual classes: [1 0 2 1 1]

Final Predicted Output (first 5 samples):
['1.0000', '0.0000', '0.0000']
['1.0000', '0.0000', '0.0000']
['0.0000', '1.0000', '0.0000']
['1.0000', '0.0000', '0.0000']
['1.0000', '0.0000', '0.0000']

Target Output (first 5 samples):
[[0. 1. 0.]
 [1. 0. 0.]
 [0. 0. 1.]
 [0. 1. 0.]
 [0. 1. 0.]]

Test Accuracy: 96.67%
```

Entire Programme:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split

np.random.seed(42)

# Load the Iris dataset
data = load_iris()
X = data.data # shape: (150, 4)
y = data.target.reshape(-1, 1) # reshape for encoder

# One-hot encode the labels
encoder = OneHotEncoder(sparse_output=False)
y_encoded = encoder.fit_transform(y) # shape: (150, 3)

# Normalize features (optional)
X = (X - np.mean(X, axis=0)) / np.std(X, axis=0)

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y_encoded, test_size=0.2, random_state=42
)

# Set network dimensions
input_dim = 4  # number of features
hidden_dim = 6 # size of hidden layer
hidden_dim2 = 5
output_dim = 3 # number of classes

# Initialize weights and biases
W1 = np.random.randn(4, 6)
b1 = np.random.randn(1, 6)
W2 = np.random.randn(6, 5)
b2 = np.random.randn(1, 5)
W3 = np.random.randn(5, 3)
b3 = np.random.randn(1, 3)

# Activation functions
def relu(a):
    return np.maximum(0, a)

def relu_derivative(a):
    return (a > 0).astype(float)
```

```
def softmax(a):
    exp_a = np.exp(a - np.max(a, axis=1, keepdims=True))
    return exp_a / np.sum(exp_a, axis=1, keepdims=True)

# Loss function: cross-entropy
def cross_entropy(y_pred, y_true):
    eps = 2e-10
    return -np.mean(np.sum(y_true * np.log(y_pred + eps), axis=1))

# Gradient of loss w.r.t. logits (output preactivation)
def dL_da3(y_pred, y_true):
    return y_pred - y_true

# Training loop
for epoch in range(10000):
    # Forward pass
    z1 = X_train @ W1 + b1  # preactivation hidden layer
    a1 = relu(z1)           # activation hidden layer
    z2 = a1 @ W2 + b2
    a2 = relu(z2)
    z3 = a2 @ W3 + b3       # preactivation output layer
    y_pred = softmax(z3)    # activation output layer (softmax)

    # Compute loss
    loss = cross_entropy(y_pred, y_train)

    # Backpropagation
    da3 = dL_da3(y_pred, y_train) # dL/da3
    dW3 = a2.T @ da3
    db3 = np.sum(da3, axis=0, keepdims=True)

    dz2 = da3 @ W3.T
    da2 = dz2 * relu_derivative(a2)
    dW2 = a1.T @ da2
    db2 = np.sum(da2, axis=0, keepdims=True)

    dz1 = da2 @ W2.T # dL/dz1
    da1 = dz1 * relu_derivative(a1) # dL/da1
    dW1 = X_train.T @ da1
    db1 = np.sum(da1, axis=0, keepdims=True)

    # Update parameters
    lr = 0.01 #Learning Rate
    W3 -= lr * dW3
    b3 -= lr * db3
    #Update W2
    W2 -= lr * dW2
```

```
    #Update b2
    b2 -= lr * db2
    #Update W1
    W1 -= lr * dW1
    #update b1
    b1 -= lr * db1

    # Print loss every 10 epochs
    if epoch % 1000 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")

# Final predictions on test set
z1_test = X_test @ W1 + b1
a1_test = relu(z1_test)

z2_test = a1_test @ W2 + b2
a2_test = relu(z2_test)

z3_test = a2_test @ W3 + b3
y_test_pred = softmax(z3_test)

# Print predicted vs actual for comparison
print("Predicted classes:", np.argmax(y_test_pred[:5], axis=1))
print("Actual classes:", np.argmax(y_test[:5], axis=1))

print("\nFinal Predicted Output (first 5 samples):")
for row in y_pred[:5]:
    print([f"{val:.4f}" for val in row])

print("\nTarget Output (first 5 samples):")
print(y_test[:5])

correct_preds = np.argmax(y_test_pred, axis=1) == np.argmax(y_test, axis=1)
accuracy = np.mean(correct_preds)
print(f"\nTest Accuracy: {accuracy * 100:.2f}%")
```