Task1 :-

✓Project Component & Deliverables

1. Model.py (Core Implementation)
   - Dense layer
   - Activations (ReLU, Sigmoid)
   - Loss (MSE)
   - Neural Network Wrapper

```
# model.py
import numpy as np


# ----------------------------
# Dense Layer
# ----------------------------
class Dense:
    def _init_(self, input_dim, output_dim):
        """
```

Fully connected layer:

y = xW + b

input_dim: number of input features

output_dim: number of neurons in this layer

"""

Xavier Initialization for weights

limit = np.sqrt(6 / (input_dim + output_dim))

self.W = np.random.uniform(-limit, limit, (input_dim, output_dim))

self.b = np.zeros((1, output_dim))

To store gradients and inputs for backpropagation

self.input = None

self.dW = None

self.db = None

```
  def forward(self, X):
      """
      Compute linear transformation: XW + b
      """
      self.input = X
      return X @ self.W + self.b

  def backward(self, grad_output):
      """
      Compute gradients for weights, bias, and input.
      grad_output: gradient of loss wrt layer output
      """
      # Gradient wrt weights: input^T * grad_output
      self.dW = self.input.T @ grad_output
```

```
    # Gradient wrt bias: sum along batch dimension
        self.db = np.sum(grad_output, axis=0, keepdims=True)
        # Gradient wrt input (to propagate backwards)
        return grad_output @ self.W.T


    def update_params(self, lr):
        """
        Gradient descent parameter update
        """
        self.W -= lr * self.dW
        self.b -= lr * self.db



# ---------------------------
```

```
# Activation: ReLU
# ----------------------------
class ReLU:
    def forward(self, X):
        """
        ReLU(x) = max(0, x)
        """
        self.input = X
        return np.maximum(0, X)

    def backward(self, grad_output):
        """
        Pass gradient only where input > 0
        """
        grad = grad_output.copy()
        grad[self.input <= 0] = 0
        return grad
```

```
# ----------------------------
# Activation: Sigmoid
# ----------------------------
class Sigmoid:
    def forward(self, X):
        """
        Sigmoid(x) = 1 / (1 + exp(-x))
        """
        self.output = 1 / (1 + np.exp(-X))
        return self.output


    def backward(self, grad_output):
        """
        Gradient of sigmoid: sigmoid * (1 - sigmoid)
        """
```

```
  return grad_output * self.output * (1 - self.output)



# ----------------------------
# Loss Function: MSE
# ----------------------------
class MSELoss:
    def forward(self, y_pred, y_true):
        """
        Compute Mean Squared Error
        """
        return np.mean((y_pred - y_true) ** 2)

    def backward(self, y_pred, y_true):
        """
```

```
Gradient of MSE wrt predictions: 2*(pred - true)/N
        """
        return 2 * (y_pred - y_true) / y_true.shape[0]



# ----------------------------
# Neural Network Wrapper
# ----------------------------
class NeuralNetwork:
    def _init_(self, layers):
        """
        layers: list of layers and activations
        """
        self.layers = layers

    def forward(self, X):
```

```
"""
    Sequential forward pass through layers
    """
    for layer in self.layers:
        X = layer.forward(X)
    return X

def backward(self, grad):
    """
    Sequential backward pass (reverse order)
    """
    for layer in reversed(self.layers):
        grad = layer.backward(grad)

def update(self, lr):
    """
```

Update parameters for layers that have weights

"""

for layer in self.layers:

if hasattr(layer, 'update_params'):

layer.update_ params(lr)

Explanation of model.py

- ✓ Dense Layer: Implements $y = xW + b$. Handles weight initialization, forward pass, and backpropagation (computes gradients write inputs, weights, and biases).

- ✓ ReLU Activation: Applies ReLU function; during backpropagation, only passes gradients where $input > 0$.

✓Sigmoid Activation: Applies sigmoid function; computes derivative for backpropagation.

✓MSE Loss: Computes loss and gradient write predictions.

✓Neural Network: Combines layers and handles forward/backward passes and parameter updates.

2. Train.ipynb (Training & visualization)

Requirenments

- Data generation
- Network creation
- Training with mini- batch SGD
- Plotting loss curves
- Plotting predictions vs ground truth

train.ipynb

```
import numpy as np
import matplotlib.py
plot as plt
from model import Dense, ReLU, Sigmoid, MSE Loss, Neural Network

# 1. Generate Synthetic Data
np.random.seed(42)

# Univariate cubic function y = x^3 + noise
X = np.linspace(-2, 2, 200).reshape(-1, 1)
y = X**3 + 0.3 * np.random.randn(*X.shape)

# 2. Build Neural Network
# ---------------------------
model = Neural Network([Dense(1, 64),
   ReLU(),
```

```
Dense(64, 64), ReLU(),
   Dense(64, 1)
])

loss_fn = MSELoss()

# 3. Training Parameters
# ----------------------------
epochs = 2000
lr = 0.01
batch_size = 32
losses = []

# 4. Training Loop
# ----------------------------
for epoch in range(epochs):
   # Shuffle data for each epoch
```

```
indices = np.random.permutation(len(X))
X_shuffled, y_shuffled = X[indices], y[indices]

# Mini-batch training
for i in range(0, len(X), batch_size):
    X_batch = X_shuffled[i:i+batch_size]
    y_batch = y_shuffled[i:i+batch_size]

    # Forward pass
    y_pred = model.forward(X_batch)

    # Compute loss
    loss = loss_fn.forward(y_pred, y_batch)
    losses.append(loss)

    # Backward pass
    grad = loss_fn.backward(y_pred, y_batch)
```

```
model.backward(grad)

    # Parameter update
    model.update(lr)

  # Print progress
  if epoch % 200 == 0:
    print(f"Epoch {epoch}, Loss: {loss:.4f}")

# 5. Visualization
# ---------------------------
# Plot loss curve
plt.figure(figsize=(8, 4))
plt.plot(losses)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training Loss Curve')
```

```
plt.show()

# Plot predictions vs ground truth
y_pred_all = model.forward(X)

plt.figure(figsize=(8, 4))
plt.scatter(X, y, label='True Data', color='blue')
plt.plot(X, y_pred_all, label='Prediction', color='red')
plt.legend()
plt.title('Model Prediction vs Ground Truth')
plt.show()
```

1. Data Generation:

- Uses cubic function with added Gaussian noise.
- Good test for non-linear regression.

2. Network Architecture:

- 1 input → Dense(64) → ReLU → Dense(64) → ReLU → Dense(1)

- ReLU prevents vanishing gradients and models non-linearity well.

3. Training Loop:

- Mini-batch SGD: Shuffles data and trains on small chunks (batch size = 32).

- Forward pass: Compute predictions.
- Loss: Compute MSE.
- Backward pass: Compute gradients.
- Update: Adjust weights using learning rate.

4. Visualization:

- Loss curve shows convergence.
- Final plot compares model predictions to true data.

3. README.md (Project Documentation)

Neural Network Regression (NumPy Only)

Overview
This project implements a fully connected neural network from scratch using NumPy for a regression task (predicting continuous values). The model is trained on a synthetic noisy cubic function.

Architecture
- Input: 1 feature (x)
- Hidden layers: Two Dense layers (64 neurons each) with ReLU activation
- Output: 1 neuron (linear activation)
- Loss: Mean Squared Error (MSE)
- Optimizer: Stochastic Gradient Descent (SGD) with mini-batches

Training
- Dataset: 200 points from $y = x^3 + noise$
- Epochs: 2000
- Learning rate: 0.01
- Batch size: 32

Results
- Loss decreases steadily during training.
- Model successfully fits cubic data with noise.
- Prediction closely matches ground truth.

Files
- model.py: Core implementation (layers, activations, loss, optimizer)
- train.ipynb: Data generation, training, visualization
- README.md: Project explanation and results

Future Improvements
- Add Adam optimizer for faster convergence.
- Add L2 regularization to avoid overfitting.
- Extend to multivariate regression tasks.

Imports

I will use NumPy for the majority of this project. However, I will also import a few other libraries to read the data and make visualizations.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

random.seed(27)
np.random.seed(27)
plt.style.use('ggplot')
```

Loading and Processing Data

```
train = pd.read_csv('input /train.csv')
```

```
X_test = pd.read_csv('input/digit-recognizer/test.csv')


print(f"Train dataset shape: {train.shape}")
print(f"Test dataset shape: {X_test.shape}")
```

Output:

```
Train dataset shape: (42000, 785)
Test dataset shape: (28000, 784)
```

```
X_train = train.drop('label', axis=1)
y_train = train['label']


# Taking a random sample and normalizing it, keeping the original for comparision.
random_index = np.random.choice(X_train.shape[0])
sample_before = X_train.iloc[random_index]
sample_after = sample_before / 255.0
```

```
plt.figure(figsize=(14, 8))
plt.rcParams['axes.grid'] = False


plt.subplot(2, 2, 1)
plt.hist(X_train.values.flatten(), bins=256, color='darkblue')
plt.title('Pixel value distribution')
plt.xlabel('Pixel value')
plt.ylabel('Frequency (log scale)')
plt.yscale('log')


plt.subplot(2, 2, 2)
plt.imshow(sample_before.values.reshape(28, 28), cmap='gray')
plt.title('Before Normalization')
plt.colorbar()
```

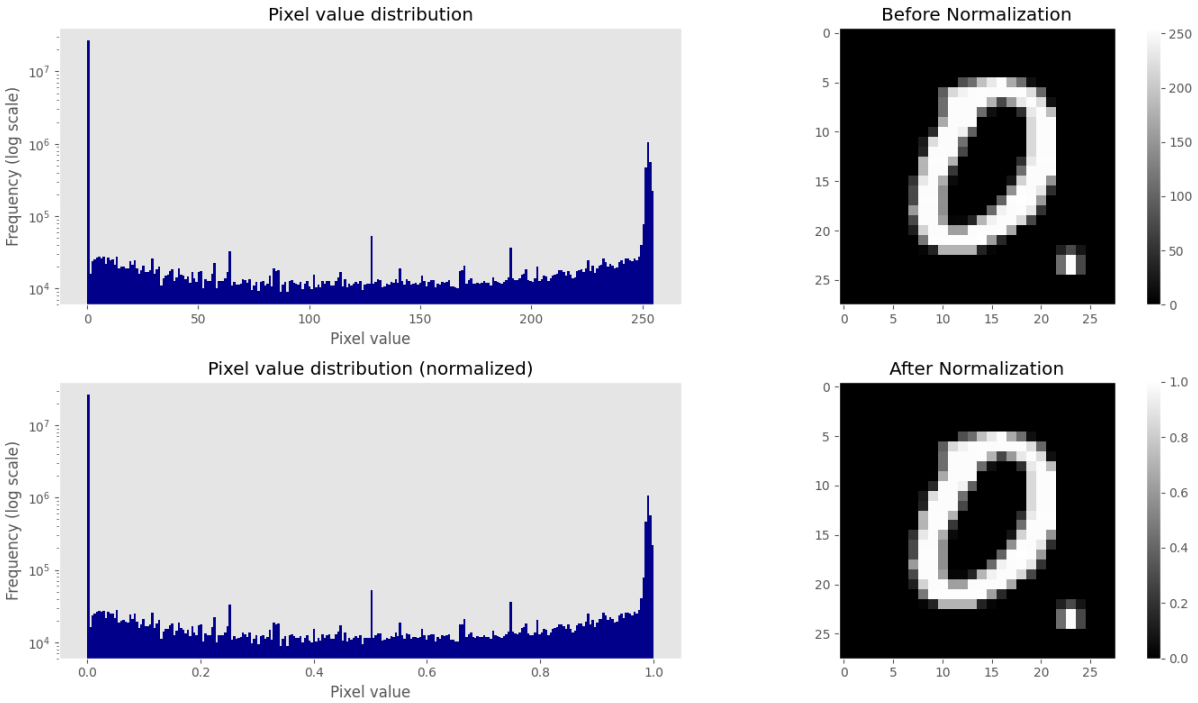
```
# Normalize the data
X_train = X_train / 255.0
X_test = X_test / 255.0


plt.subplot(2, 2, 3)
plt.hist(X_train.values.flatten(), bins=256, color='darkblue')
plt.title('Pixel value distribution (normalized)')
plt.xlabel('Pixel value')
plt.ylabel('Frequency (log scale)')
plt.yscale('log')


plt.subplot(2, 2, 4)
plt.imshow(sample_after.values.reshape(28, 28), cmap='gray')
plt.title('After Normalization')
```

```
plt.colorbar()


plt.tight_layout()
plt.show()
```




```
y_train_samples = y_train[:10].values


num_classes = y_train.max() + 1
y_train = np.eye(num_classes)[y_train]
y_train = y_train.astype(int)
```

```
y_train_onehot_samples = y_train[:10]

for y, y_onehot in zip(y_train_samples, y_train_onehot_samples):
    print(f"Original label: {y} | One-hot encoded label: {y_onehot}")
```

Activation Functions

- Starting with forward pass, it is important to understand what activation functions are and why they are used in neural networks.
- Activation functions are mathematical functions applied to the output of each neuron in a neural network's hidden layers. They introduce non-linearity to the network, enabling it to learn complex patterns in data. Without activation functions, neural networks would essentially reduce to linear transformations, which limits their ability to learn complex relationships in data.

I've used two different activations functions in my network architecture, Rectified Linear Unit (ReLU) and SoftMax. The former is used in the hidden layer, and the latter is applied to the output layer. ReLU is one of the most widely used activation functions. It sets all negative values to zero and leaves positive values

unchanged. For a given input value x, ReLU is defined as

ReLU(x)=max(0,x)

SoftMax is commonly used in the output layer of a neural network. It converts the raw output scores into probabilities, ensuring that the sum of the probabilities across all classes equals one. It is defined as

$$Pi = ezi / \sum nj=1 ezj$$

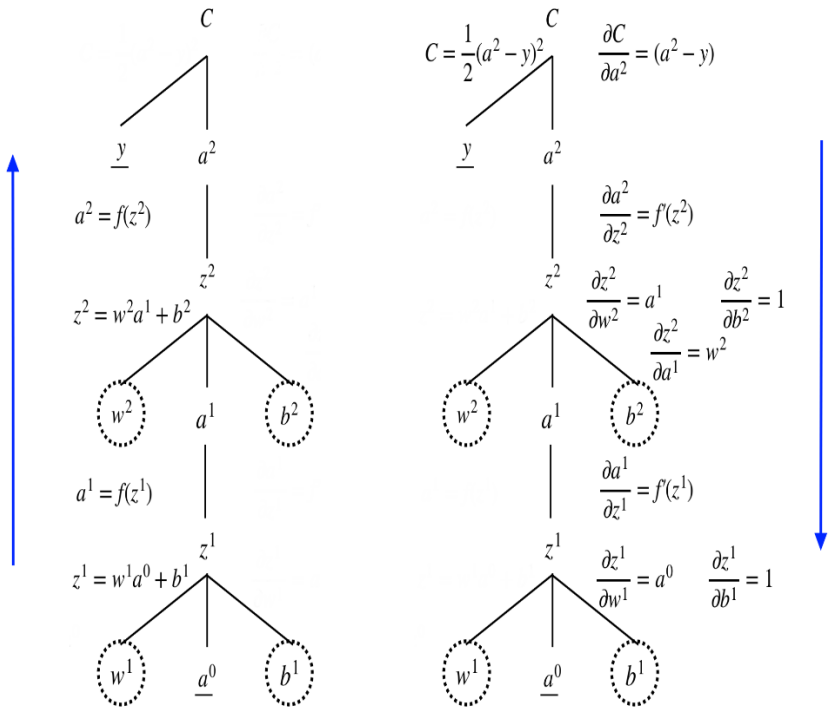
The code for these functions is shown below. I've also defined the derivative of ReLU as it is needed in backpropagation.

```
def relu(x):
    return np.maximum(x, 0)
def relu_derivative(x):
    return np.where(x > 0, 1, 0)
def softmax(x):
```

```
  exp_x = np.exp(x - np.max(x, axis=1,
keepdims=True))
    return exp_x / np.sum(exp_x, axis=1,
keepdims=True)
```

Forward pass

The tree structure above to be very useful when trying to understand the mathematical operations involved in forward and backward pass. On the left, we can see the equations of forward pass, and on the right, the equations of backpropagation.



Starting from the bottom, we have the input values, i.e. the flattened input images denoted as x=a0. We also have the weights that connect the input layer to the first and only hidden layer, w1, as well as the bias of the hidden layer denoted as b1. These values are used to calculate the first weighted sum, z1, as follows

a0=x

z1=w1a0+b1

The weighted sum is passed through an activation function f, in this case, f=ReLU.

a1=f(z1)=ReLU(z1)

z2=w2a1+b2

$$a2=SoftMax(z2)$$

```
def forward(X, W1, b1, W2, b2):
    z1 = X @ W1 + b1
    a1 = relu(z1)
    z2 = a1 @ W2 + b2
    a2 = softmax(z2)
    return z1, a1, z2, a2
```

BACKWARD PASS

output, a2=^y. This value alongside the ground truth label y is run through a loss function. Ideally, at each iteration the value of the loss function should decrease, indicating the model's improving ability to learn. I've used the Mean Squared Error function (MSE) for this task. It is defined as

$$C=1/2(a2-y)2$$

```
def backward(X, y, z1, a1, z2, a2, W1, W2, b1, b2):
    m = X.shape[0]

    delta_2 = a2 - y
    dW2 = a1.T @ delta_2 / m
    db2 = np.sum(delta_2 * 1, axis=0) / m

    delta_1 = delta_2 @ W2.T * relu_derivative(z1)
    dW1 = X.T @ delta_1 / m
    db1 = np.sum(delta_1 * 1, axis=0) / m

    return dW1, db1, dW2, db2
```

PARAMETERS

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

the parameters, i.e. the weights and biases, we first need to define a learning rate, α. The learning rate is a hyperparameter that determines the step size at which the model's parameters are updated during the training process. We'll use the optimization algorithms gradient descent to minimize the loss function. The learning rate controls the size of the steps taken in the direction opposite to the gradient of the loss function. The generalized algorithm is

θ new=θ old−α∇C

where θnew and θold denote the updated parameters and the parameters from the previous iteration respectively. Applying this to our use case, we get

w2 new = w2 old – a dc/dw2

Training

```
def train(X, y, W1, b1, W2, b2, learning_rate, epochs, batch_size):
    histories = {
        "epoch": [],
        "step": [],
        "loss": [],
        "accuracy": []
    }
```

```
    step = 1
    loop = tqdm(range(epochs))
    for epoch in loop:
        for X_batch, y_batch in get_batch(X, y, batch_size):
            z1, a1, z2, a2 = forward(X_batch, W1, b1, W2, b2)
            dW1, db1, dW2, db2 = backward(X_batch, y_batch, z1, a1, z2, a2, W1, W2, b1, b2)
            W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate)


            if step % 100 == 0:
                loss = mean_squared_error(y_batch, a2)
                acc = accuracy(y_batch, a2)
                histories["epoch"].append(epoch + 1)
                histories["step"].append(step)
```

```
    histories["loss"].append(loss)
            histories["accuracy"].append(acc)
            loop.set_postfix(loss=loss, accuracy=acc)
        step += 1

    return W1, b1, W2, b2, histories
def mean_squared_error(y_true, y_pred):
    return np.sum((y_pred - y_true) ** 2) / y_true.shape[0]
def accuracy(y_true, y_pred):
    return np.mean(np.argmax(y_true, axis=1) == np.argmax(y_pred, axis=1))
input_size = X_train.shape[1]
hidden_size = 128
output_size = num_classes
learning_rate = 0.01
```

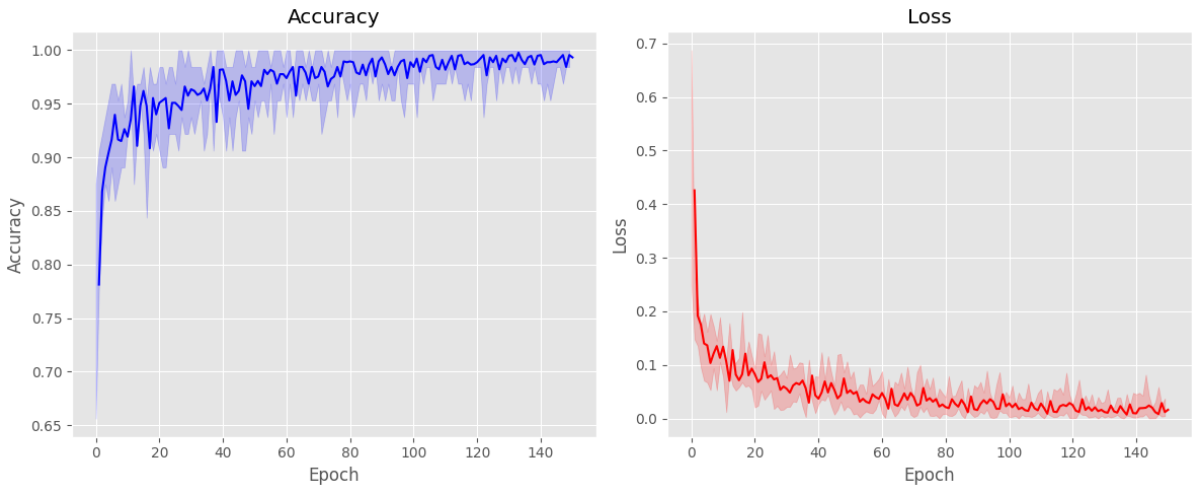
epochs = 150

batch_size = 64

W1, b1, W2, b2 = init_weights(input_size, hidden_size, output_size)

W1, b1, W2, b2, histories = train(X_train.values, y_train, W1, b1, W2, b2, learning_rate, epochs, batch_size)

100%|[redacted]| 150/150 [03:33<00:00, 1.42s/it, accuracy=0.984, loss=0.0182]



Neural Network Training Notebook

This notebook demonstrates training neural networks from scratch using NumPy.

It includes data generation, model training, gradient checking, and comprehensive

Visualizations for regression tasks.