Neural Network for Digit Classification

This project implements a neural network from scratch to classify handwritten digits from the `sklearn` Digits dataset. The network is trained using backpropagation and gradient descent.

Dataset

The dataset used is the `load_digits` dataset from `sklearn`. It contains 8x8 grayscale images of digits (0-9) with 64 features (pixel intensities) and corresponding labels.

Optical Recognition of Handwritten Digits Dataset

Data Set Characteristics:

- **Number of Instances:** 1797
- **Number of Attributes:** 64
- **Attribute Information:** 8x8 image of integer pixels in the range 0..16.
- **Missing Attribute Values:** None
- **Creator:** E. Alpaydin (alpaydin '@' boun.edu.tr)
- **Date:** July 1998

Preprocessing

1. **Normalization**: The input features are normalized using `Normalizer` to ensure all values are on the same scale.
2. **One-Hot Encoding**: The target labels are one-hot encoded to match the output layer's format.
3. **Train-Test Split**: The dataset is split into training (80%) and testing (20%) sets.

Neural Network Architecture

The neural network consists of **3 layers**:

Layer 1 (Input Layer)

- **Input Features**: 64 (plus 1 bias term, making it 65)
- **Neurons**: 16
- **Weights Dimensions**: `65 x 16`

Layer 2 (Hidden Layer)

- **Input Features**: 16 (plus 1 bias term, making it 17)
- **Neurons**: 16
- **Weights Dimensions**: `17 x 16`

Layer 3 (Output Layer)

- **Input Features**: 16 (plus 1 bias term, making it 17)
- **Neurons**: 10 (corresponding to the 10 digit classes)
- **Weights Dimensions**: `17 x 10`

Activation Functions

- **ReLU**: Used in the hidden layers to introduce non-linearity.

$$\text{ReLU}(x) = \max(0, x)$$

- **Softmax**: Used in the output layer to convert logits into probabilities.
  The softmax function is defined as:

$$\text{Softmax}(z * i) = \frac{e^{z_i}}{\sum * j = 1^n e^{z_j}}$$

  where z_i is the i-th logit, and n is the total number of logits.
  This ensures that the output probabilities sum to 1.

Forward Propagation

1. **Layer 1**:

$$z^{(1)} = X \cdot W^{(1)} + b^{(1)}$$

$$a^{(1)} = \text{ReLU}(z^{(1)})$$

2. **Layer 2**:

$$z^{(2)} = a^{(1)} \cdot W^{(2)} + b^{(2)}$$

$$a^{(2)} = \text{ReLU}(z^{(2)})$$

3. **Layer 3 (Output Layer)**:

$$z^{(3)} = a^{(2)} \cdot W^{(3)} + b^{(3)}$$

$$\hat{Y} = a^{(3)} = \text{Softmax}(z^{(3)})$$

Loss Function

The loss function used is **Cross-Entropy Loss**:

$$J = -\frac{1}{n} \sum \left(Y \cdot \log(\hat{Y}) \right)$$

Backpropagation

The gradients are computed for each layer using the chain rule:

1. **Output Layer**:
    - Gradients for weights and biases are computed as:

$$\frac{\partial J}{\partial W^{(3)}} = \frac{1}{n} \sum (a^{(3)} - Y) \cdot a^{(2)T}$$

$$\frac{\partial J}{\partial b^{(3)}} = \frac{1}{n}\sum(a^{(3)} - Y)$$

2. **Hidden Layers**:
   - For Layer 2:

$$\delta^{(2)} = \left(\delta^{(3)} \cdot W^{(3)}\right) \odot \mathrm{ReLU}'(z^{(2)})$$

$$\frac{\partial J}{\partial W^{(2)}} = \frac{1}{n}\sum \delta^{(2)} \cdot a^{(1)T}$$

$$\frac{\partial J}{\partial b^{(2)}} = \frac{1}{n}\sum \delta^{(2)}$$

   - For Layer 1:

$$\delta^{(1)} = \left(\delta^{(2)} \cdot W^{(2)}\right) \odot \mathrm{ReLU}'(z^{(1)})$$

$$\frac{\partial J}{\partial W^{(1)}} = \frac{1}{n}\sum \delta^{(1)} \cdot X^{T}$$

$$\frac{\partial J}{\partial b^{(1)}} = \frac{1}{n}\sum \delta^{(1)}$$

   - Here, $\delta^{(i)}$ represents the error term for layer i, and the derivatives of the activation functions are as follows:
     - **ReLU**:

$$\mathrm{ReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$

     - **Softmax**:
       The derivative of the softmax function for a single output i is:

$$\frac{\partial \mathrm{Softmax}(z_i)}{\partial z_j} = \mathrm{Softmax}(z_i) \cdot (\delta_{ij} - \mathrm{Softmax}(z_j))$$

where δ_ij is the Kronecker delta, equal to 1 if $i = j$, and 0 otherwise.

Training

- **Optimizer**: Gradient Descent
- **Learning Rate**: 0.01
- **Batch Size**: 32
- **Epochs**: 500

The weights are updated using:

$$W^{(i)} \leftarrow W^{(i)} - \alpha \cdot \frac{\partial J}{\partial W^{(i)}}$$

$$b^{(i)} \leftarrow b^{(i)} - \alpha \cdot \frac{\partial J}{\partial b^{(i)}}$$

Results

After training, the model is evaluated on the test set. The accuracy is computed as:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Predictions}}$$

Results on the training set

```
Epoch 0, Loss: 2.3802869928898365
Epoch 50, Loss: 1.206206512585289
Epoch 100, Loss: 0.4039758421433619
Epoch 150, Loss: 0.2563353337566877
Epoch 200, Loss: 0.19490139962663064
Epoch 250, Loss: 0.15863091100372181
Epoch 300, Loss: 0.13382645965174142
Epoch 350, Loss: 0.11549642915155892
Epoch 400, Loss: 0.10236851909048866
Epoch 450, Loss: 0.0910746108913426
```

Results on the test set

```
Test Accuracy: 96.94%
```

Notes

The accuracy of the model can be significantly enhanced by using more layers in the neural network and training for more epochs. These improvements would allow the model to better capture complex patterns in the data.

How to Run

1. Ensure all dependencies are installed:
    - `numpy`
    - `scikit-learn`
2. Run the Jupyter Notebook `ann.ipynb` to train and evaluate the model.

File Structure

- `ann.ipynb` : Contains the implementation of the neural network.
- `utils.py` : Contains helper functions.