

Project Neural Network

Students:

Shahd Fawzy Toutou 23012087

Malak Alaa 23012115

Maram Mohamed 23012197

Dataset

path(<https://www.kaggle.com/code/wwwsalmon/simple-mnist-nn-from-scratch-numpy-no-tf-keras/input?select=train.csv>)

I. overview of the Dataset

The dataset used in this project is an image classification dataset consisting of grayscale handwritten digit images. Each instance in the dataset represents a **flattened 28×28 image** (total 784 pixels), and includes:

- **Label:** The target class, indicating the digit (0–9).
- **Features:** 784 numerical features, each corresponding to a pixel intensity (ranging from 0 to 255).

Dataset Characteristics:

- Number of examples: 70,000 (e.g., MNIST full set)
- Feature type: Numerical (pixel intensities)
- Target variable: Categorical (0–9)
- Balanced: Yes (roughly equal number of samples per class)

```
# Load dataset
data = pd.read_csv("c:/Users/Delta/Documents/datasetML/Mnist.csv")

data.head()

label pixel0 pixel1 pixel2 pixel3 pixel4 pixel5 pixel6 pixel7 pixel8 ... pixel774 pixel775 pixel776 pixel777 pixel778 pixel779 pixel780 pixel781 pixel782
0 1 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0
2 1 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0
3 4 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0 0 0 0 0 ... 0 0 0 0 0 0 0 0 0

5 rows x 785 columns

data.shape

(42000, 785)

data.columns[:10]

Index(['label', 'pixel0', 'pixel1', 'pixel2', 'pixel3', 'pixel4', 'pixel5',
       'pixel6', 'pixel7', 'pixel8'],
      dtype='object')
```

II.Clean data

1. Missing Values:

To check if there are any missing values in the dataset. Missing values can negatively impact model performance, and it is essential to handle them before feeding the data into a machine learning model

2. Duplicate Records:

To check if there are any duplicate rows in the dataset. Duplicate rows can occur due to data entry errors or data collection issues, and they can bias the model during training.

3. Dataset Summary:

To get a statistical summary of the dataset, including important metrics such as mean, standard deviation, minimum, maximum, and percentiles. This helps in understanding the general distribution of the features and detecting any extreme values or anomalies.

```
data.isnull().sum()
[58] ✓ 0.0s
...
label      0
pixel0     0
pixel1     0
pixel2     0
pixel3     0
..
pixel779   0
pixel780   0
pixel781   0
pixel782   0
pixel783   0
length: 785, dtype: int64
```

```
data.describe()
[59] ✓ 1.6s
Python
...

```

	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	...	pixel774	pixel775	pixel776	pixel777	pixel778
count	42000.000000	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	42000.0	...	42000.000000	42000.000000	42000.000000	42000.000000	42000.000000
mean	4.456643	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.219286	0.117095	0.059024	0.02019	0.017231
std	2.887730	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	6.312890	4.633819	3.274488	1.75987	1.894491
min	0.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.000000	0.000000	0.000000
25%	2.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.000000	0.000000	0.000000
50%	4.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.000000	0.000000	0.000000
75%	7.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.000000	0.000000	0.000000	0.000000	0.000000
max	9.000000	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	254.000000	254.000000	253.000000	253.000000	254.000000

```
8 rows x 785 columns

data.duplicated().any()
[60] ✓ 0.8s
Python
...
np.False_
```

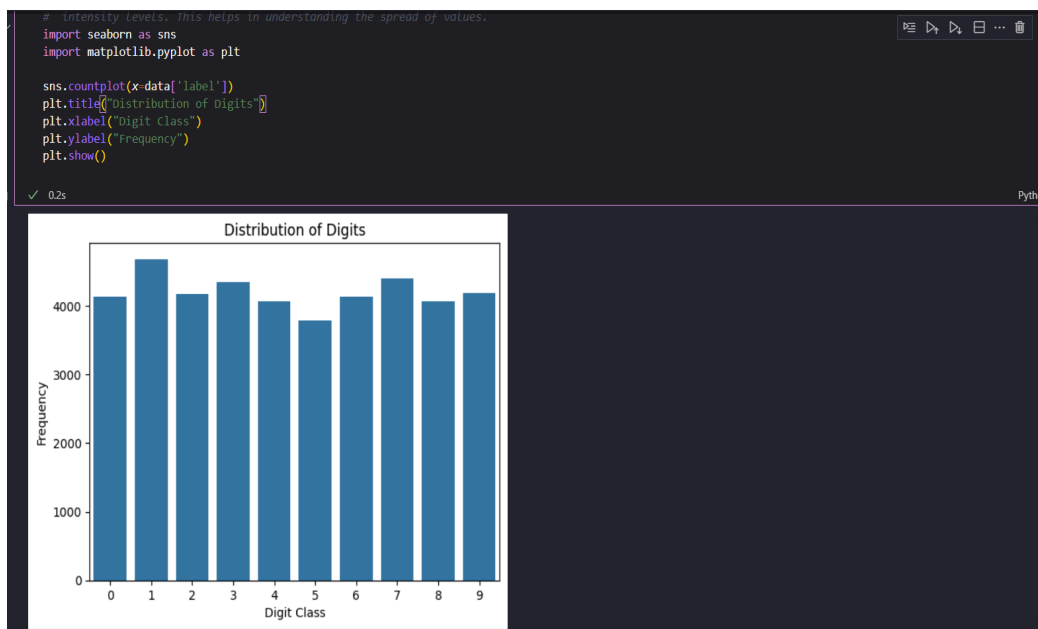
VI.Exploratory Data Analysis (EDA)

1 .Label Distribution (Class Distribution):

The goal of this visualization is to understand the distribution of the target labels (digits) in the dataset. We want to check whether the dataset is balanced or imbalanced, as an imbalanced dataset can lead to biased model training, where the model might be biased toward more frequent classes.

What it shows:

The bar plot presents the frequency of each digit (0 to 9) in the dataset. The x-axis represents the digit class, while the y-axis shows how many samples exist for each class.



2. Visualizing Sample Digits:

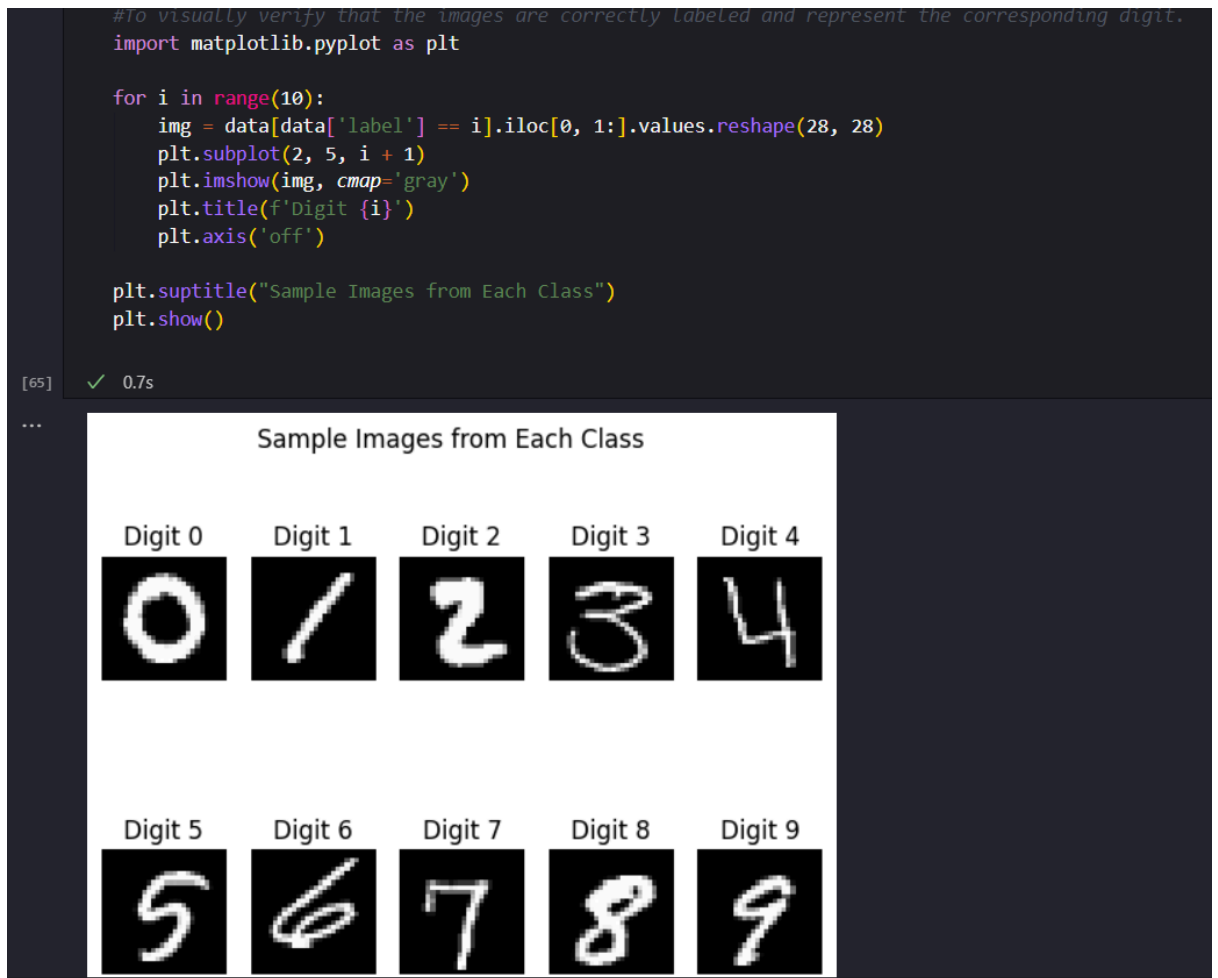
This visualization helps ensure that the images are loaded correctly and each class (digit) is visually distinguishable. It is crucial for confirming that the data labeling is correct and that the model will not learn from mislabeled data.

What it shows:

The 2x5 grid displays one sample image for each digit (0–9).

The images are shown in grayscale to match the way they will appear when fed into the model.

Each image is labeled with the corresponding digit, confirming the visual accuracy of the data.



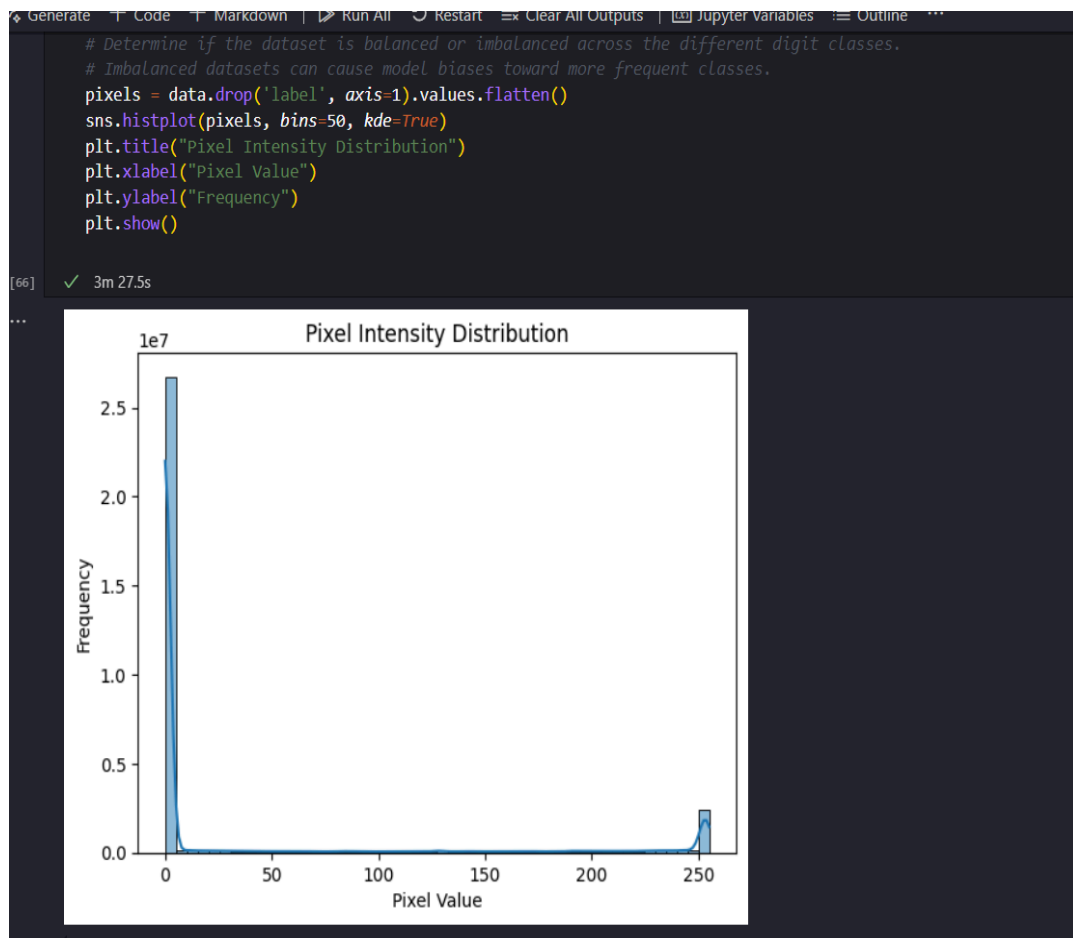
3. Pixel Value Distribution:

The purpose of this visualization is to understand the distribution of pixel intensity values across the entire dataset. Since image data is represented by numerical pixel values, knowing how these values are distributed can help us determine if normalization or scaling is necessary.

What it shows:

The histogram of pixel values, with the x-axis showing the pixel values (0 to 255) and the y-axis showing the frequency of those values across all images.

The kernel density estimation (KDE) curve provides a smooth visualization of the distribution.



VI.Data Preprocessing Pipeline:

After completing the exploratory data analysis, the next step is to preprocess the data to prepare it for training in a neural network. The key steps include:

1 .Feature/Label Separation

The dataset is split into features (X) and labels (Y).

- Features (X): All pixel values, excluding the label column.
- Labels (Y): The target digit (0–9).

```
Generate Code Markdown
# Separate the features (pixel values) and the labels (targets)
x = data.drop('label', axis=1).values # Remove the label column to get features only
y = data['label'].values              # Extract only the label column
```

✓ 0.1s

2. Data Conversion to NumPy Array:

This line converts the dataset from a Pandas DataFrame to a NumPy array, which is more suitable for numerical computations and neural network training. NumPy arrays allow efficient matrix operations like transposing and normalization, which are essential when building models from scratch. This step prepares the data for splitting, normalization, and feeding into the neural network efficiently.

3. Normalization:

We normalize the pixel values by dividing by 255 so that the values range from 0 to 1. This step ensures that the neural network can learn more efficiently and avoids issues with gradients during training.

4. Shuffling the Data:

Shuffling the data is crucial to ensure that the model does not learn patterns based on the order of the data. It helps in achieving **better generalization**.

5. Train/Test Split:

The dataset is split into:

- Training set: The majority of the data, used for training the model.
- Test set: A smaller portion, used to evaluate model performance after training.

```
# Convert to NumPy array (if not already)
data = np.array(data)      Chat (CTRL + I) / Share (CTRL + L)

# Get number of rows (examples) and columns (features + label)
m, n = data.shape
print("Data shape (with labels):", data.shape)
68] ✓ 0.0s

Data shape (with labels): (42000, 785)

# Shuffle the data randomly to avoid any order bias
np.random.shuffle(data)
69] ✓ 1.1s

# Split test data (first 1000 examples)
data_test = data[0:1000].T      # Transpose to shape: (features, examples)
Y_test = data_test[0]          # First row is Labels
X_test = data_test[1:n]        # Remaining rows are pixel data
X_test = X_test / 255.         # Normalize pixel values to [0
70] ✓ 0.0s

# Split training data (remaining examples)
data_train = data[1000:m].T    # Transpose training data
Y_train = data_train[0]        # First row is Labels
X_train = data_train[1:n]      # Remaining rows are pixel data
X_train = X_train / 255.       # Normalize training pixel values
71] ✓ 0.1s
```

Neural Network Training:

##about our trials##

we worked with another data firstly and used mini batch in learning and tried all possible learning rates and the max **acc = 0.34** then we searched for another ways and also the acc still the same so we changed the data and worked with another data -> mnist also we needed everything from the scratch so we splitted the data manually and then get `x_train` , `x_test` `y_train` , `y_test` then normalized data to be easier for the model to work with digits between `[0,1]` then wrote the algorithm we typically used matrix form as we took in the lec , the first trial was with learning rate = 0.01 and number of iterations = 500 the progress was too small and slow but still there was a progress then we tried to change the learning rate with = 0.1 (increased) and number of iterations = 1000 the progress was amazing and the acc - = 87% , but it had taken 4m and 38sec in running maybe after adding the optimizer it deals with this part (time)

Functions Explained:

1. `init_params()`

It sets the starting point for learning. Without initializing weights and biases, the network has nothing to adjust — and learning cannot begin.

2. `ReLU(Z)`

Introduces non-linearity, allowing the model to learn complex patterns. Without ReLU, the network would behave like a linear model no matter how deep it is.

3. `softmax(Z)`

Calculates the predicted output by passing the input through the network. This is the core process where the model makes its prediction.

4. `forward_prop(...)`

Calculates the predicted output by passing the input through the network. This is the core process where the model makes its prediction.

5. `ReLU_deriv(Z)`

Needed during backpropagation to correctly compute how errors flow backward through ReLU-activated layers.

6. `one_hot(Y)`

Converts class labels into a format the model can compare against softmax outputs, essential for computing the loss in classification.

7. backward_prop(...)

This is how the network learns – it calculates how much each parameter contributed to the error, so we know how to adjust them.

8. adam_update(...)

It updates weights in a way that improves stability and speed of learning by combining momentum and adaptive learning rate. Adam is more efficient than plain gradient descent.

9. get_predictions(A2)

Converts softmax probabilities to actual class predictions. This step is needed to evaluate accuracy and test performance.

10. get_accuracy(predictions, Y)

Provides a direct measure of how well the model is doing. Accuracy helps track performance during training and testing.

11. train_adam(...)

Coordinates the entire training process. It combines forward pass, backpropagation, and parameter updates over many iterations.

12. update_params(...)

Optional basic gradient descent method. Useful if Adam is not used. Shows how updates can be done manually for learning purposes.

overall Workflow of the Neural Network:

1. Initialization

The training begins by calling `init_params()`, which randomly initializes the weights and biases for both layers. These parameters will be updated during learning.

2. Training Loop (train_adam)

This function controls the entire learning process, repeating for a number of iterations.

- **In each iteration:**

- It performs forward propagation (forward_prop) to compute predictions from the current input and parameters.
- Then, it calculates the loss (difference between prediction and actual).
- It uses backpropagation (backward_prop) to compute gradients – how much each weight and bias contributed to the error.
- The gradients are used in Adam optimizer (adam_update) to update the weights and biases more efficiently than standard gradient descent.
- Every few iterations, it prints the current accuracy and loss using get_predictions and get_accuracy to track learning progress.

3. Activation Functions (ReLU, softmax)

These give the network the power to learn complex and non-linear patterns (ReLU) and output probability distributions (softmax).

4. Evaluation

After training, the final model can be used to predict new data, and its performance can be measured by comparing predictions to true labels.

Output (with out adam)

```
#first learning rate = 0.01 and the acc through 500 iteration was 0.44
# so we will increase the number of iter = 1000 and the learning rate to 0.1 to be faster and also monit
1 ✓ 1m 34.8s

Iteration: 0
[3 4 9 ... 1 1 9] [1 7 2 ... 2 4 2]
0.06619512195121952
Iteration: 100
[1 0 1 ... 2 4 2] [1 7 2 ... 2 4 2]
0.5192682926829268
Iteration: 200
[1 7 1 ... 2 4 2] [1 7 2 ... 2 4 2]
0.7301951219512195
Iteration: 300
[1 7 2 ... 2 4 2] [1 7 2 ... 2 4 2]
0.7918048780487805
Iteration: 400
[1 7 2 ... 2 4 2] [1 7 2 ... 2 4 2]
0.8217317073170731
Iteration: 500
[1 7 2 ... 2 4 2] [1 7 2 ... 2 4 2]
0.8368048780487805
Iteration: 600
[1 7 2 ... 2 4 2] [1 7 2 ... 2 4 2]
0.8480243902439024
Iteration: 700
[1 7 2 ... 2 4 2] [1 7 2 ... 2 4 2]
0.8561463414634146
Iteration: 800
...
0.8615121951219512
Iteration: 900
[1 7 2 ... 2 4 2] [1 7 2 ... 2 4 2]
0.8672439024390244

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Output: (accuracy with adam)

```
Iteration 100, Loss: 1.35716, Accuracy: 0.5548
Iteration 200, Loss: 0.70702, Accuracy: 0.7762
Iteration 300, Loss: 0.50674, Accuracy: 0.8456
Iteration 400, Loss: 0.42041, Accuracy: 0.8753
Iteration 500, Loss: 0.37213, Accuracy: 0.8913
Iteration 600, Loss: 0.34022, Accuracy: 0.9012
Iteration 700, Loss: 0.31716, Accuracy: 0.9083
Iteration 800, Loss: 0.29950, Accuracy: 0.9131
Iteration 900, Loss: 0.28558, Accuracy: 0.9174
Iteration 1000, Loss: 0.27431, Accuracy: 0.9201
```

Prediction and Testing:

make_predictions:

Uses the trained neural network to predict class labels for given input data by applying forward propagation and selecting the class with the highest probability.

test_prediction:

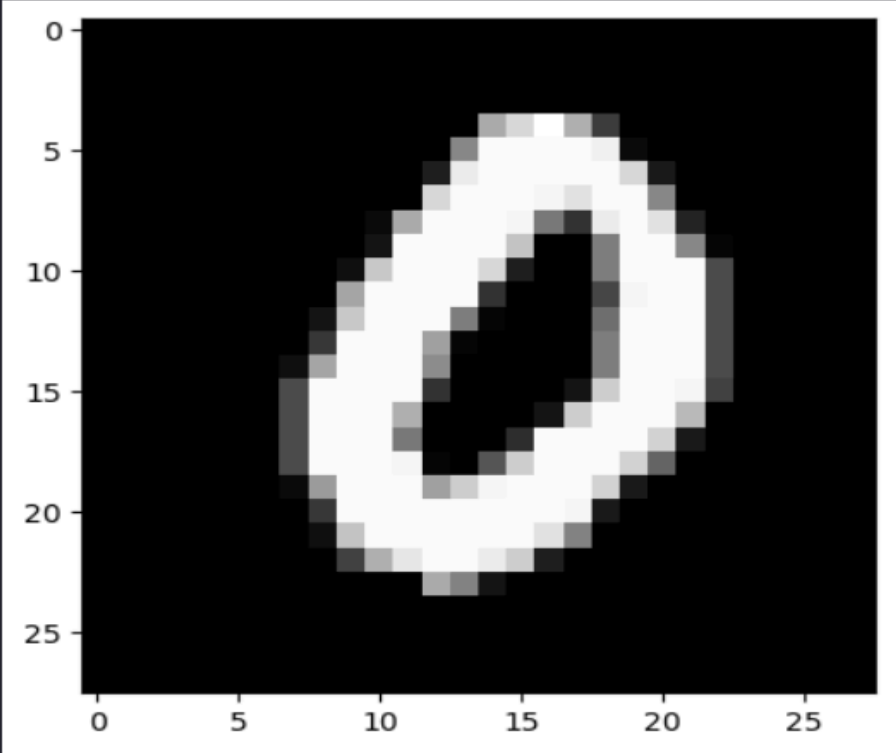
Tests and visualizes a single prediction by:

- Running the model on one image.
- Printing the predicted and actual labels.
- Displaying the image for manual inspection.

Let's look at a couple of examples:

```
test_prediction(0, w1, b1, w2, b2)
test_prediction(1, w1, b1, w2, b2)
test_prediction(2, w1, b1, w2, b2)
test_prediction(3, w1, b1, w2, b2)
```

Prediction: [0]
Label: 0



Finally, let's find the accuracy on the dev set:

```
2 0 3 1 7 8 9 0 4 5 8 2 7 9 4 1 2 2 3 4 0 2 0 9 5 7 7 8 6 9 9 1 8 2 9 0 2
3 5 1 9 2 0 7 8 6 9 4 2 8 0 1 6 7 4 9 4 3 2 9 9 5 4 9 0 9 0 1 4 3 2 6 4 6
1 4 1 0 8 1 5 6 8 8 1 9 2 0 0 4 8 0 8 0 5 5 6 4 4 4 6 5 3 3 5 6 6 3 0 9 9
4 4 8 5 6 2 1 2 1 9 8 4 0 8 5 7 4 1 3 8 8 8 2 1 0 2 2 8 4 3 5 7 4 7 2 2 3
3 8 0 9 5 0 7 3 2 0 9 5 4 2 2 7 3 7 2 7 2 5 6 3 8 8 4 6 7 6 7 7 1 8 6 2 7
3 9 9 3 3 2 0 8 8 5 1 8 7 6 4 6 3 9 4 2 8 3 9 8 2 1 8 1 4 2 7 1 1 0 2 4 5
1 9 5 9 4 3 9 8 4 1 9 4 6 7 8 4 8 8 8 6 0 9 3 8 6 3 0 0 9 2 0 6 8 2 0 4 5
7 6 4 2 8 0 1 4 4 2 8 0 7 5 5 0 1 4 7 9 6 1 9 0 0 9 0 8 0 3 6 7 6 0 2 2 0
1 4 1 6 0 0 4 9 6 6 2 6 2 0 3 0 8 2 1 3 1 5 8 7 0 2 3 8 8 7 7 7 4 2 7 0 7
9 0 1 5 7 8 2 5 0 0 5 9 5 1 3 2 0 9 5 7 8 7 0 8 8 3 3 7 8 9 5 0 3 5 9 6 7
6 6 6 0 9 4 5 3 4 6 3 1 7 6 8 9 6 8 9 7 4 1 8 0 8 2 2 2 1 2 7 1 4 9 6 9 8
0 1 4 4 3 0 6 6 2 8 0 8 3 2 9 7 1 4 1 7 1 3 1 7 6 2 1 2 1 8 4 6 9 9 2 0
5 3 5 3 9 6 9 9 6 3 6 4 5 1 4 2 0 2 7 9 1 1 5 9 9 9 0 7 5 4 8 6 8 5 5 2 6
...
5 3 5 3 9 6 9 9 6 3 6 4 5 1 4 3 0 2 7 9 1 1 5 9 9 9 0 7 5 4 8 6 8 5 5 2 6
9 3 3 4 7 6 2 7 5 4 3 7 1 9 9 1 9 2 6 0 3 7 7 8 9 2 8 1 8 9 4 0 3 4 7 4 6
9 9 3 7 6 1 3 8 6 0 5 8 2 1 5 7 8 2 0 6 6 7 3 7 6 5 0 2 9 9 9 7 1 8 3 2 2
4]
```

Output is truncated. View as a [scrollable element](#) or open in a [text editor](#). Adjust cell output [settings](#)...

0.878

