```
# Import packages:
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Load and Read the training-set
data = pd.read_csv('/kaggle/input/digit-recognizer/train.csv')

# Display the shape of the training set
print(f"Shape of train data before Transposing: {data.shape}")

# Extract the target from the features
y_train = data['label'].values

# Drop the target from the feature set
x_train = data.drop('label', axis=1).values

# Write a function that splits the data into two train and validation sets (80/20 split)
def train_test_split(X, y, test_size=0.2):
    samples_number = X.shape[0]
    validation_samples_number = int(test_size * samples_number)

    index = np.random.permutation(samples_number)

    validation_indices = index[:validation_samples_number]
    train_indices = index[validation_samples_number:]

    x_train, x_validation = X[train_indices], X[validation_indices]
    y_train, y_validation = y[train_indices], y[validation_indices]

    return x_train, x_validation, y_train, y_validation

# Split the data into training and validation sets (80/20 split)
x_train, x_validation, y_train, y_validation = train_test_split(x_train, y_train, test_size=0.2)

# Check the shapes of the resulting datasets
print("X_train shape:", x_train.shape)
print("X_validation shape:", x_validation.shape)
print("y_train shape:", y_train.shape)
print("y_validation shape:", y_validation.shape)

# Write a function for data preprocessing
def preprocess_data(x_train, y_train, x_validation, y_validation):
    x_train_scaled = x_train.T / 255.0
    x_validation_scaled = x_validation.T / 255.0
    y_train_encoded = np.eye(10)[y_train].T    # One-hot encode training labels
    y_validation_encoded = np.eye(10)[y_validation].T    # One-hot encode validation labels

    return x_train_scaled, y_train_encoded, x_validation_scaled, y_validation_encoded

# Write the activation functions
def ReLU(z):
    return np.maximum(z, 0)

def ReLU_derivative(z):
    return np.where(z > 0, 1, 0)
```

```
def Softmax(z):
    exp_z = np.exp(z - np.max(z))    # Avoid overflow
    return exp_z / np.sum(exp_z, axis=0)

# Write a function to initialize the parameters for forward propagation
def initialize_parameters(input_size, hidden_size, output_size):
    np.random.seed(0)    # Set seed(0) get the same random numbers.

    # Initialize the Weights, and Biases
    w1 = np.random.randn(hidden_size, input_size) * 0.01    # Multiply weights by a small
alpha (0.001 OR 0.01) to make the weights as small as possible
    b1 = np.zeros((hidden_size, 1))
    w2 = np.random.randn(output_size, hidden_size) * 0.01
    b2 = np.zeros((output_size, 1))

    parameters = {'w1': w1,
                  'b1': b1,
                  'w2': w2,
                  'b2': b2}

    return parameters

# Feed the parameters through the forward process
def forward_propagation(x, parameters):
    # Extract the parameters
    w1 = parameters['w1']
    b1 = parameters['b1']
    w2 = parameters['w2']
    b2 = parameters['b2']

    # Calculate the forward equations
    z1 = np.dot(w1, x) + b1
    a1 = ReLU(z1)
    z2 = np.dot(w2, a1) + b2
    a2 = Softmax(z2)

    forward_cache = {'z1': z1,
                     'a1': a1,
                     'z2': z2,
                     'a2': a2}

    return a2, forward_cache
```