Image Recognition using Neural Networks in Machine Learning

Student: Yi Qiang Ji Zhang
Professor: Dr. Alex Ferrer Ferré
Aerospace Engineering
Polythecnical University of Catalonia

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola Superior d'Enginyeries Industrial, Aeroespacial i Audiovisual de Terrassa

10 June 2021

1 Introduction

A neuron is the basic unit of the neural network. It takes inputs parameters, then does some computation with them, and produces one output. Here's what a 2-input neuron looks like:

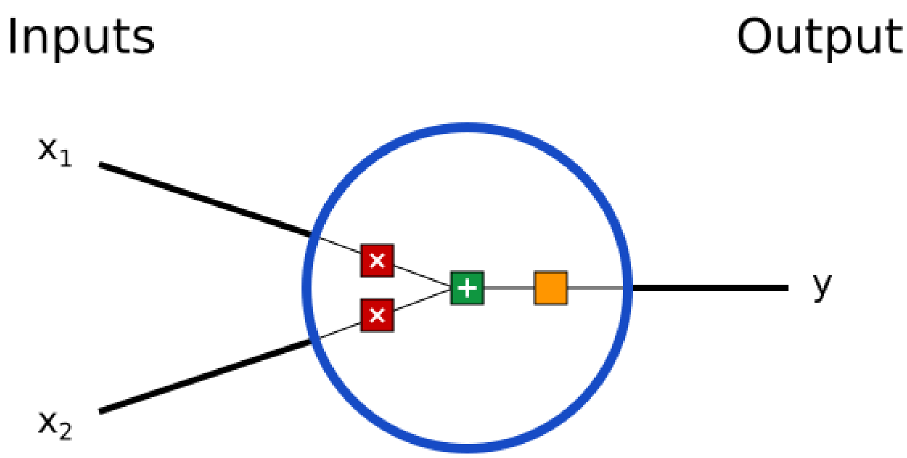


Figure 1: Schematic of a neuron. Source: Victor Zhou [1].

Each input is multiplied by a weight:

$$x_1 \longrightarrow \omega_1 \cdot x_1 \tag{1}$$

$$x_2 \longrightarrow \omega_2 \cdot x_2 \tag{2}$$

All the weight inputs are added with some bias:

$$(\omega_1 \cdot x_1) + (\omega_2 \cdot x_2) + b \tag{3}$$

And finally, the sum is passed with an activation function:

$$y = (\omega_1 \cdot x_1 + \omega_2 \cdot x_2 + b) \tag{4}$$

A commonly used activation function is the sigmoid function:

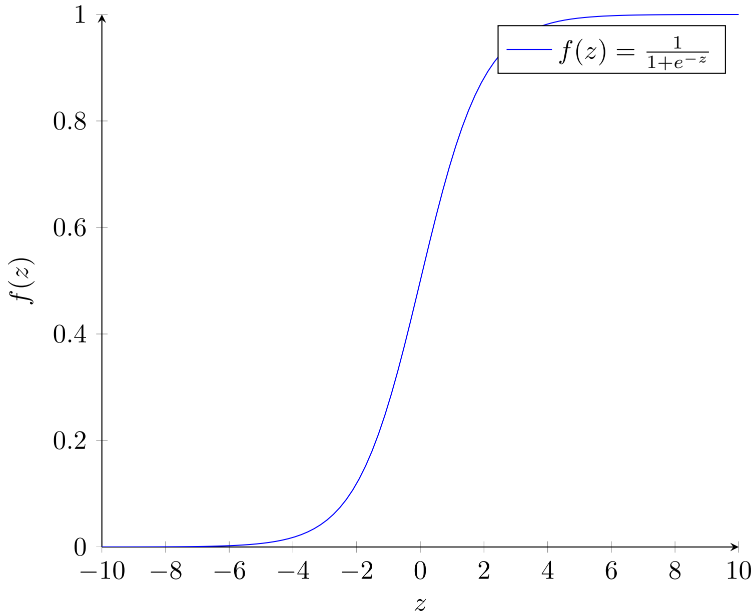


Figure 2: Sigmoid function. Source: Own.

Or the ReLU function:

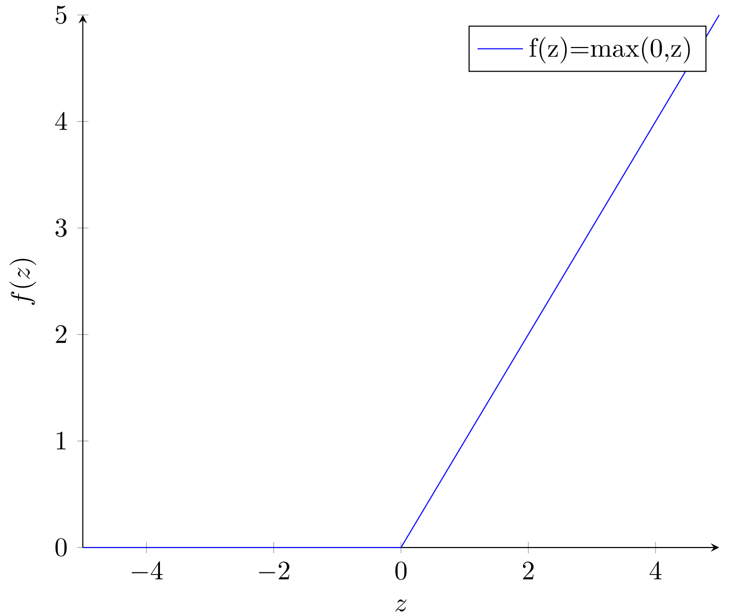


Figure 3: ReLU function. Source: Own.

The sigmoid function only outputs numbers in the range $(0, 1)$. You can think of it as compressing $(-\infty, +\infty)$ to $(0, 1)$, big negative numbers become ≈ 0, and big positive numbers become ≈ 1.

2 Model of Neural Network

A neural network is built by connecting several of our basic "neurons" so that the output of one can be the input of another. Here's an example of a little neural network:

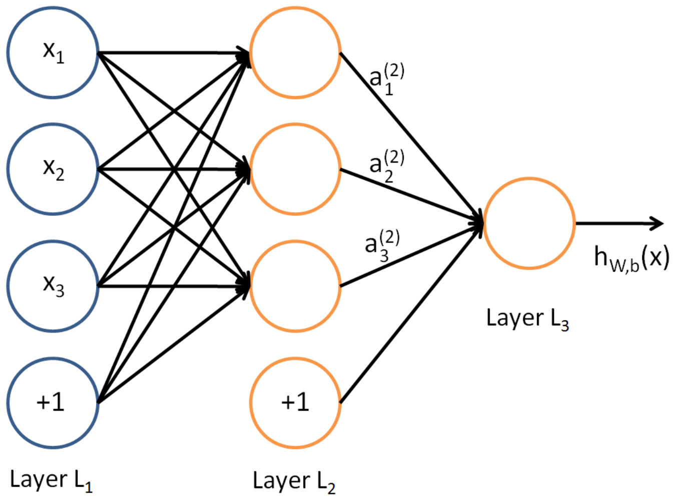


Figure 4: Neural Nerwork

In this diagram, circles represent the network's inputs. Bias units are the circles labeled " +1 " that relate to the intercept term. The network's leftmost layer is known as the input layer, while the network's rightmost layer is known as the output layer (which, in this example, has only one node). Because its values are not noticed in the training set, the intermediate layer of nodes is referred to as the hidden layer.

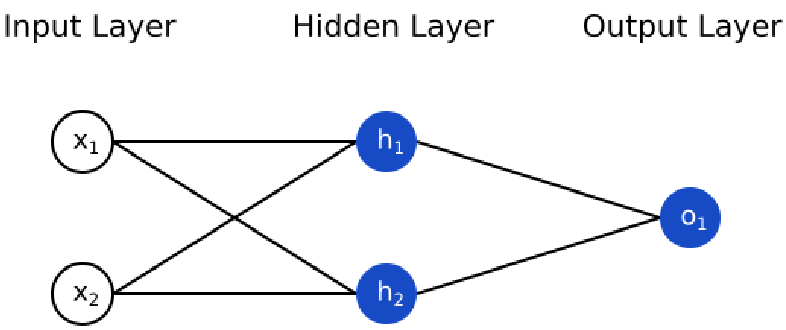


Figure 5: Neural Network layers. Source: [1]

3 Problem statement

The dataset we're using is the well-known MNIST handwritten digit dataset, which is frequently used in both ML and computer vision applications. It includes 28 × 28 grayscale photos of handwritten numerals that resemble as follows:

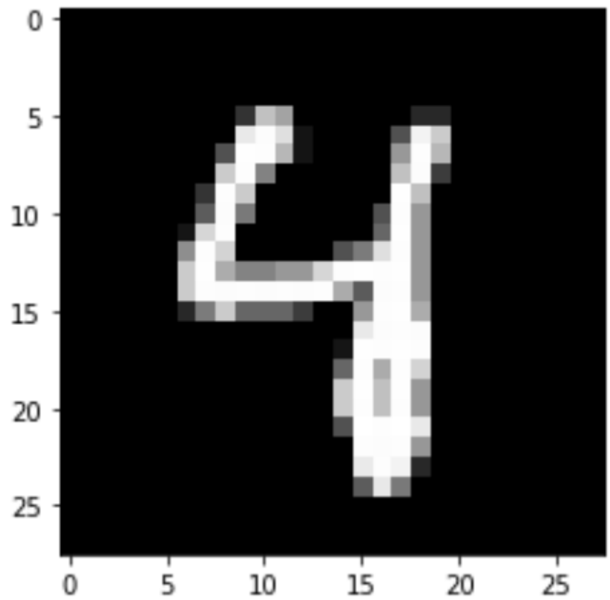


Figure 6: MNIST number example. Source: [2]

What we are seeking is to generate a neural network that is capable of recognizing the numbers from the dataset.

4 Methodology

Each image is labeled with the digit it relates to, ranging from 0 to 9. Our aim is to create a network that can take a picture like this and determine which digit is written in it.

Forward propagation refers to the act of taking an image input and putting it through a neural network to produce a prediction. The prediction generated from a particular picture is determined by the network's weights and biases, or parameters.

To train a neural network, we must update these weights and biases in order to make correct predictions. This is accomplished by a technique known as gradient descent. The main principle behind gradient descent is to work out which direction each parameter may go in to reduce error the most, then push each parameter in that direction again and over until the values with the least error and best accuracy are determined.

Gradient descent in a neural network is accomplished by a method known as backward propagation, or backprop. Instead of running an input image forwards through the network to get a prediction, backprop takes the previously made prediction, calculates the error of how far it was off from the actual value, and then runs this error backwards through the network to determine how much each weight and bias parameter contributed to the error. We may enhance our model by adjusting our weights and biases based on the error derivative terms. If we repeat this process enough times, we will create a neural network that can properly detect handwritten digits.

Softmax takes a column of data at a time, taking each element in the column and outputting the exponential of that element divided by the sum of the exponentials of each of the elements in the input column. The end result is a column of probabilities between 0 and 1.

The Neural Network will feature a straightforward two-layer design. The input layer $a[0]$ will have 784 units, which match to the 784 pixels in each 28x28 input picture. A hidden layer will have 10 units with ReLU activation, and the output layer will have 10 units with softmax activation corresponding to the ten digit classes.

4.1 Forward propagation

Forward propagation will be computed in the following section,

First, let's compute the unactivated values of the nodes in the first hidden layer by applying $W^{[1]}$ and $b^{[1]}$ to the input layer. We'll call the output of this operation $\boldsymbol{Z}^{[1]}$.

$$Z^{[1]} = W^{[1]}X + b^{[1]} \tag{5}$$

The bias matrix has 10×1 dimensions and in column so it is applied to all m columns of the training examples. ReLU is used for this non-linearity. It is a non-linear function which outputs the input parameter if it is above 0 or 0 if the input parameter is below 0.

$$A^{[1]} = g_{\text{ReLU}}(Z^{[1]})) \tag{6}$$

Moreover, a non-linear activation function is needed in order to build the regression model. Otherwise, the sum of the weights multiplied by the base will result in a linear combination when moving from layer to layer. Finally, the last layer yields,

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \tag{7}$$

Finally. the last activation function chosen is not the ReLU but a `softmax` since the output shall be from [0, 1] as a probability. Softmax works with a column of data at a time, taking each element in the column and dividing the exponential of that element by the total of the exponentials of all the components in the input column. The ultimate result is a column of probabilities ranging from 0 to 1:

$$A^{[2]} = g_{\text{softmax}}(Z^{[2]}) \tag{8}$$

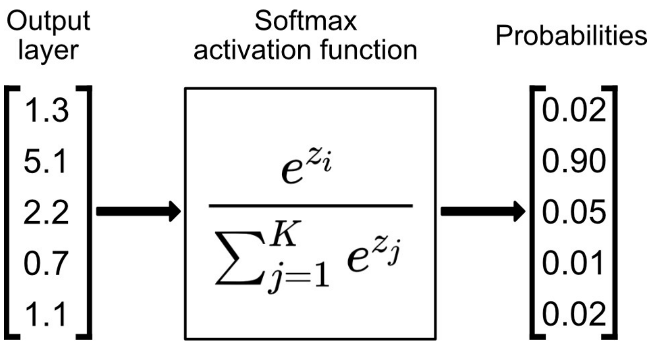


Figure 7: Softmax scheme. Source: [3].

4.2 Backward propagation

In forward propagation we compute a prediction of the result using a given set of weights and biases. However, in backward propagation we will be using the error and update the coefficients according to the labels. This is by using the following cross-entropy loss function:

$$J(\hat{y}, y) = -\sum_{i=0}^{c} y_i \log(\hat{y}_i) \tag{9}$$

Here, $\hat{y}$ is the prediction vector. It might look like this:

$$\begin{bmatrix} 0.01 \\ 0.02 \\ 0.05 \\ 0.02 \\ 0.80 \\ 0.01 \\ 0.01 \\ 0.00 \\ 0.01 \\ 0.07 \end{bmatrix} \tag{10}$$

And y is the one-hot encoding of the correct label for the training example. If the result is 5, the one-hot encoding of y would look like this:

$$\begin{matrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{matrix} \tag{11}$$

It's important to notice that the sum $\sum_{i=0}^{c} y_i \log\left(\hat{y}_i\right), y_i = 0$ for all i except the correct label. It's worth noting that the closer the prediction probability is to one, the closer the loss is to O. The loss approaches $+\infty$ as the likelihood approaches zero. We increase the accuracy of the model by reducing the cost function. Over several rounds of gradient descent, we subtract the derivative of the loss function with respect to each parameter from that parameter.

$$\begin{gathered} W^{[1]} := W^{[1]} - \alpha \frac{\delta J}{\delta W^{[1]}} \\ b^{[1]} := b^{[1]} - \alpha \frac{\delta J}{\delta b^{[1]}} \\ W^{[2]} := W^{[2]} - \alpha \frac{\delta J}{\delta W^{[2]}} \\ b^{[2]} := b^{[2]} - \alpha \frac{\delta J}{\delta b^{[2]}} \end{gathered} \tag{12}$$

The objective in backward propagation is to find $\frac{\delta J}{\delta W^{[1]}}, \frac{\delta J}{\delta b^{[1]}}, \frac{\delta J}{\delta W^{[2]}}$, and $\frac{\delta J}{\delta b^{[2]}}$.

By using the chain rule, it is possible to obtain the cost function derivatives

$$dZ^{[2]} = A^{[2]} - Y \tag{13}$$

$$dZ^{[2]} = A^{[2]} - Y \tag{14}$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T} \tag{15}$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} .* g^{[1]\prime}(z^{[1]}) \tag{16}$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]T} \tag{17}$$

$$dB^{[1]} = \frac{1}{m} \Sigma dZ^{[1]} \tag{18}$$

4.3 Parameter updates

After finding the correpondand derivatives, we shall update the weights and biases:

$$W^{[2]} := W^{[2]} - \alpha dW^{[2]} \tag{19}$$

$$b^{[2]} := b^{[2]} - \alpha db^{[2]} \tag{20}$$

$$W^{[1]} := W^{[1]} - \alpha dW^{[1]} \tag{21}$$

$$b^{[1]} := b^{[1]} - \alpha db^{[1]} \tag{22}$$

Where α is the hyper parameter which is arbitrary. The user adjusts the parameter.

Finally, this process is done several times as for getting better more accurate version of the parameters.

4.4 Variables and parameter sizes

- $A^{[0]} = X$: 784 x m
- $Z^{[1]} \sim A^{[1]}$: 10 x m
- $W^{[1]}$: 10 x 784 (as $W^{[1]}A^{[0]} \sim Z^{[1]}$)
- $B^{[1]}$: 10 x 1
- $Z^{[2]} \sim A^{[2]}$: 10 x m
- $W^{[1]}$: 10 x 10 (as $W^{[2]}A^{[1]} \sim Z^{[2]}$)
- $B^{[2]}$: 10 x 1

Forward propagation

- $A^{[0]} = X$: 784 x m
- $Z^{[1]} \sim A^{[1]}$: 10 x m
- $W^{[1]}$: 10 x 784 (as $W^{[1]}A^{[0]} \sim Z^{[1]}$)
- $B^{[1]}$: 10 x 1
- $Z^{[2]} \sim A^{[2]}$: 10 x m
- $W^{[1]}$: 10 x 10 (as $W^{[2]}A^{[1]} \sim Z^{[2]}$)
- $B^{[2]}$: 10 x 1

Backward propagation

- $dZ^{[2]}$: 10 x m ($A^{[2]}$)
- $dW^{[2]}$: 10 x 10
- $dB^{[2]}$: 10 x 1
- $dZ^{[1]}$: 10 x m ($A^{[1]}$)
- $dW^{[1]}$: 10 x 10
- $dB^{[1]}$: 10 x 1

5 Results

```
W1, b1, W2, b2 = gradientDescent(X_train, Y_train, 0.10, 1000)
```

```
Iteration:  0
[5 1 6 ... 3 1 5] [0 8 0 ... 9 3 7]
0.11492682926829269
Iteration:  100
[0 8 0 ... 7 3 3] [0 8 0 ... 9 3 7]
0.6178780487804878
Iteration:  200
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.766609756097561
Iteration:  300
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.8168048780487804
Iteration:  400
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.8408048780487805
Iteration:  500
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.8542195121951219
Iteration:  600
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.8639756097560976
Iteration:  700
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.8710731707317073
Iteration:  800
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.8760731707317073
Iteration:  900
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.8798536585365854
```

Figure 8: Accuracy precision as a function of the iterations. Source: Own.

As the number of iteration increases, the model accuracy increases as well.

Below are some images for the predictions

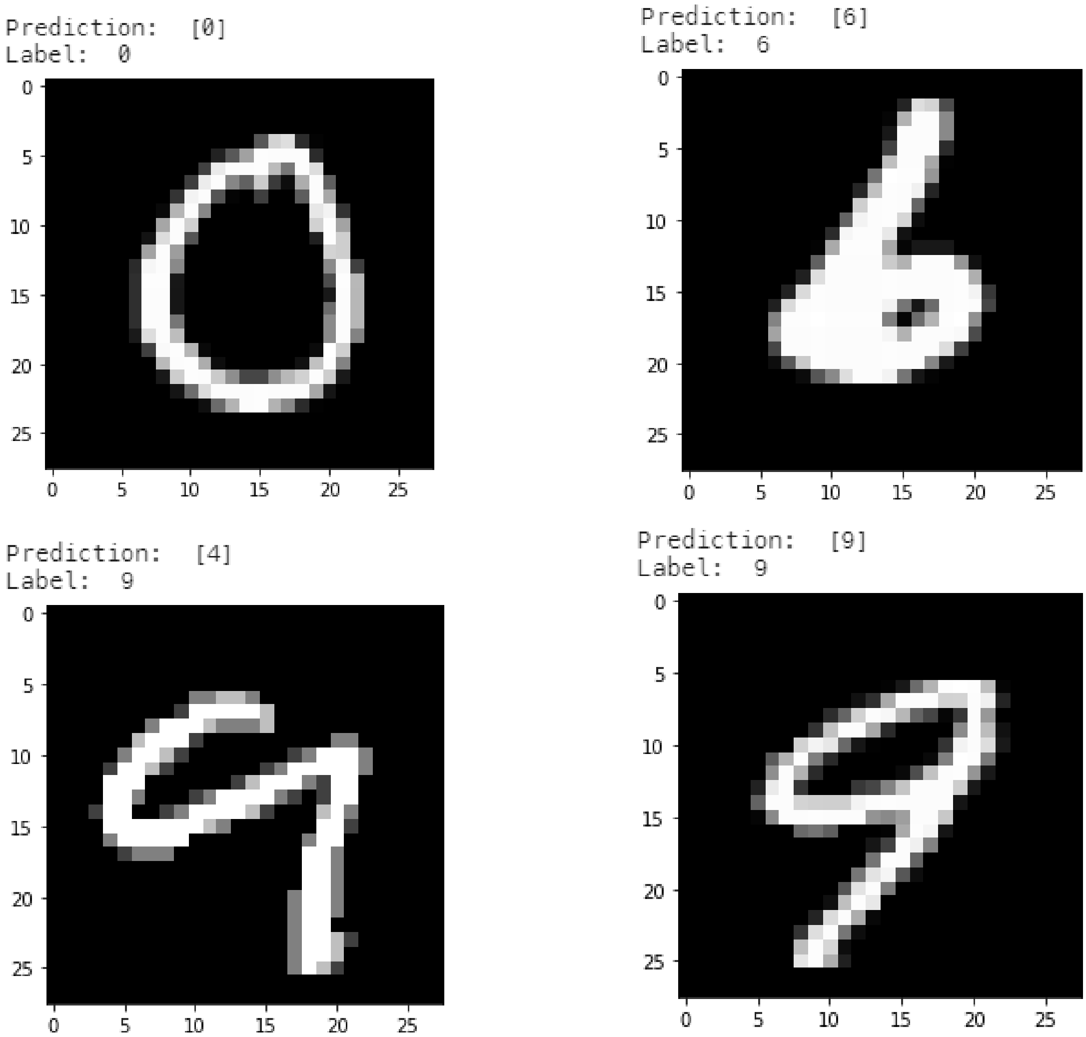


Figure 9: Train test prediction and label. Source: Own.

Also, using the above coefficients for the test set given by the dataset, the accuracy was about 88.1 % with 1000 iterations.

References

[1] Victor Zhou. *Machine Learning for Beginners: An Introduction to Neural Networks*. 2021. URL: https://towardsdatascience.com/machine-learning-for-beginners-an-introduction-to-neural-networks-d49f22d238f9.

[2] Kaggle. *Digit Recognizer*. 2021. URL: https://www.kaggle.com/c/digit-recognizer/data.

[3] Dario Radečić. *Softmax Activation Function Explained*. 2021. URL: https://towardsdatascience.com/softmax-activation-function-explained-a7e1bc3ad60.

[4] Stanford. *Multi-Layer Neural Network*. 2021. URL: http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/.

[5] Samson, Zhang. *Understanding the math behind neural networks by building one from scratch (no TF/Keras, just numpy.* 2021. URL: https://www.samsonzhang.com/2020/11/24/understanding-the-math-behind-neural-networks-by-building-one-from-scratch-no-tf-keras-just-numpy.html.

6 Code

```
# To add a new cell, type '# %%'
# To add a new markdown cell, type '# %% [markdown]'
# %% [markdown]
# # Image recognition using Neural Network
#
# The following notebook is implemented a two-layer neural network for image recognition using Python.
#
# The training set used is the MNIST dataset.

# %%
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

# Libraries
import numpy as np  # Numpy library for numerical operations and linear algebra
# Pandas library for data science tools and data processing, CSV file I/O (e.g. pd.read_csv)
import pandas as pd
from matplotlib import pyplot as plt  # Matplotlib library for MATLAB tools

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that gets preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved outside of the current session

# %% [markdown]
# ### Read data

# %%
# Read data from data set
data = pd.read_csv('/kaggle/input/digit-recognizer/train.csv')

# %% [markdown]
# ### Preview data

# %%
data.head
```

```python


# %%
# Transform data to array
data = np.array(data)
# Get the number of rows 'm' and columns 'n'
m_original, n_original = data.shape

# Shuffle data before splitting
np.random.shuffle(data)

# %% [markdown]
# ### Split data into test and training set

# %%
# Spit data from training and test
nTest = 1000

# Test set
data_test = data[0:nTest].T
Y_test = data_test[0]
X_test = data_test[1:n_original]
X_test = X_test / 255.

# Train set
data_train = data[nTest:m_original].T
Y_train = data_train[0]
X_train = data_train[1:n_original]
X_train = X_train / 255.
_, m_train = X_train.shape

# _,m_train = X_train.shape

# Y_train = Y_train / 255. # Normalize (to avoid exp overflow)


# %%
Y_train

# %% [markdown]
# The NN will feature a straightforward two-layer design. The input layer $a[0]$ will have 784 ...
    units, which match to the 784 pixels in each 28x28 input picture. A hidden layer $a[1]$ will ...
    have 10 units with ReLU activation, and the output layer $a[2]$ will have 10 units with ...
    softmax activation corresponding to the ten digit classes.
#
# **Forward propagation**
#
# $$Z^{[1]} = W^{[1]} X + b^{[1]}$$
# $$A^{[1]} = g_{\text{ReLU}}(Z^{[1]}))$$
# $$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
# $$A^{[2]} = g_{\text{softmax}}(Z^{[2]})$$
#
# **Backward propagation**
#
# $$dZ^{[2]} = A^{[2]} - Y$$
# $$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$
# $$dB^{[2]} = \frac{1}{m} \Sigma {dZ^{[2]}}$$
# $$dZ^{[1]} = W^{[2]T} dZ^{[2]} .* g^{[1]\prime} (z^{[1]})$$
```

```
# $$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]T}$$
# $$dB^{[1]} = \frac{1}{m} \Sigma {dZ^{[1]}}$$
#
# **Parameter updates**
#
# $$W^{[2]} := W^{[2]} - \alpha dW^{[2]}$$
# $$b^{[2]} := b^{[2]} - \alpha db^{[2]}$$
# $$W^{[1]} := W^{[1]} - \alpha dW^{[1]}$$
# $$b^{[1]} := b^{[1]} - \alpha db^{[1]}$$
#
# **Vars and shapes**
#
# Forward prop
#
# - $A^{[0]} = X$: 784 x m
# - $Z^{[1]} \sim A^{[1]}$: 10 x m
# - $W^{[1]}$: 10 x 784 (as $W^{[1]} A^{[0]} \sim Z^{[1]}$)
# - $B^{[1]}$: 10 x 1
# - $Z^{[2]} \sim A^{[2]}$: 10 x m
# - $W^{[1]}$: 10 x 10 (as $W^{[2]} A^{[1]} \sim Z^{[2]}$)
# - $B^{[2]}$: 10 x 1
#
# Backprop
#
# - $dZ^{[2]}$: 10 x m ($¬A^{[2]}$)
# - $dW^{[2]}$: 10 x 10
# - $dB^{[2]}$: 10 x 1
# - $dZ^{[1]}$: 10 x m ($¬A^{[1]}$)
# - $dW^{[1]}$: 10 x 10
# - $dB^{[1]}$: 10 x 1
# %% [markdown]
# ### Functions
# Below are the list of functions that will be used

# %%
# Inititate parameters


def initateParameters():
    # Get arbitrary weights and biases
    # Substract 0.5 since randn generate values from 0 to 1
    W1 = np.random.rand(10, 784) - 0.5  # Matrix of 70 x 784
    b1 = np.random.rand(10, 1) - 0.5  # Vector of 10 x 1
    W2 = np.random.rand(10, 10) - 0.5  # Matrix of 10 x 10
    b2 = np.random.rand(10, 1) - 0.5  # Vector of 10 x 1
    return W1, b1, W2, b2

# ReLU activation function


def ReLU(Z):
    return np.maximum(Z, 0)

# ReLU derivative activation function


def ReLU_deriv(Z):
    return Z > 0
```

```
# Softmax activation function


def softmax(Z):
    A = np.exp(Z) / sum(np.exp(Z))
    return A

# Sigmoid activation function
# def Sigmoid(Z):
#     S = 1 / (1 + np.exp(-Z))
#     return S

# Sigmoid derivative activation function
# def Sigmoid_deriv(Z):
#     S_deriv = Sigmoid(Z) * (1 - Sigmoid(Z))
#     return S_deriv

# Forward propagation


def forwardPropagation(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2

# Complete Y data, return the activation


def oneHot(Y):
    # Since there are 0 - 9 numbers = 10
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
    return one_hot_Y

# Backward propagation


def backwardPropagation(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = oneHot(Y)
    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m_original * dZ2.dot(A1.T)
    db2 = 1 / m_original * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_deriv(Z1)
    dW1 = 1 / m_original * dZ1.dot(X.T)
    db1 = 1 / m_original * np.sum(dZ1)
    return dW1, db1, dW2, db2

# Update parameters


def updateParameters(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
```

```
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2


# %%
# Return prediction
def getPredictions(A2):
    # Return the index of the maximum argunment, thus, the predicted number index of the [10 x 1]...
     output vector
    return np.argmax(A2, 0)

# Prediction accuracy


def getAccuracy(predictions, Y):
    print(predictions, Y)
    # Accuracy of the predicted number
    return np.sum(predictions == Y) / Y.size

# Gradient descent function


def gradientDescent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = initateParameters()
    # Loop through the amount of iterations we set
    for i in range(iterations):
        Z1, A1, Z2, A2 = forwardPropagation(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backwardPropagation(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = updateParameters(
            W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        # For every iterations, print prediction
        if i % 100 == 0:
            print("Iteration: ", i)
            predictions = getPredictions(A2)
            print(getAccuracy(predictions, Y))
    return W1, b1, W2, b2

# %% [markdown]
# ### Execute code


# %%
W1, b1, W2, b2 = gradientDescent(X_train, Y_train, 0.10, 1000)

# %% [markdown]
# ### Test an image

# %%
# To make a singular prediction with the weights and biases calculated


def makePredictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forwardPropagation(W1, b1, W2, b2, X)
    predictions = getPredictions(A2)
    return predictions

# Test prediction

```

```python

def testPredictions(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = makePredictions(X_train[:, index, None], W1, b1, W2, b2)
    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()

# %% [markdown]
# ### Check some examples of the train set


# %%
testPredictions(0, W1, b1, W2, b2)
testPredictions(1000, W1, b1, W2, b2)
testPredictions(40123, W1, b1, W2, b2)
testPredictions(40000, W1, b1, W2, b2)

# %% [markdown]
# ### Check for the test set

# %%
test_set_predictions = makePredictions(X_test, W1, b1, W2, b2)
getAccuracy(test_set_predictions, Y_test)
```

Listing 1: Python code

7 Code with Jupyter Notebook

Image recognition using Neural Network

The following notebook is implemented a two-layer neural network for image recognition using Python.

The training set used is the MNIST dataset.

```
# This Python 3 environment comes with many helpful analytics libraries installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/docker-python
# For example, here's several helpful packages to load

# Libraries
import numpy as np # Numpy library for numerical operations and linear algebra
import pandas as pd # Pandas library for data science tools and data processing, CSV file I/O (e.g. pd.read_csv
from matplotlib import pyplot as plt  # Matplotlib library for MATLAB tools

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list all files under the input direc

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that gets preserved as output when you c
# You can also write temporary files to /kaggle/temp/, but they won't be saved outside of the current session
```

```
/kaggle/input/mnist-dataset/data.csv
/kaggle/input/digit-recognizer/sample_submission.csv
/kaggle/input/digit-recognizer/train.csv
/kaggle/input/digit-recognizer/test.csv
```

Read data

```
# Read data from data set
data = pd.read_csv('/kaggle/input/digit-recognizer/train.csv')
```

Preview data

```
data.head
```

```
<bound method NDFrame.head of        label  pixel0  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  \
0          1       0       0       0       0       0       0       0       0
1          0       0       0       0       0       0       0       0       0
2          1       0       0       0       0       0       0       0       0
3          4       0       0       0       0       0       0       0       0
4          0       0       0       0       0       0       0       0       0
...      ...     ...     ...     ...     ...     ...     ...     ...     ...
41995      0       0       0       0       0       0       0       0       0
41996      1       0       0       0       0       0       0       0       0
41997      7       0       0       0       0       0       0       0       0
41998      6       0       0       0       0       0       0       0       0
41999      9       0       0       0       0       0       0       0       0

       pixel8  ...  pixel774  pixel775  pixel776  pixel777  pixel778  \
0           0  ...         0         0         0         0         0
1           0  ...         0         0         0         0         0
2           0  ...         0         0         0         0         0
3           0  ...         0         0         0         0         0
4           0  ...         0         0         0         0         0
...       ...  ...       ...       ...       ...       ...       ...
41995       0  ...         0         0         0         0         0
41996       0  ...         0         0         0         0         0
41997       0  ...         0         0         0         0         0
41998       0  ...         0         0         0         0         0
41999       0  ...         0         0         0         0         0

       pixel779  pixel780  pixel781  pixel782  pixel783
0             0         0         0         0         0
1             0         0         0         0         0
2             0         0         0         0         0
3             0         0         0         0         0
4             0         0         0         0         0
...         ...       ...       ...       ...       ...
41995         0         0         0         0         0
41996         0         0         0         0         0
41997         0         0         0         0         0
41998         0         0         0         0         0
41999         0         0         0         0         0

[42000 rows x 785 columns]>
```

```
# Transform data to array
data = np.array(data)
# Get the number of rows 'm' and columns 'n'
m_original, n_original = data.shape

# Shuffle data before splitting
np.random.shuffle(data)
```

Split data into test and training set

```
# Spit data from training and test
nTest = 1000

# Test set
data_test = data[0:nTest].T
Y_test = data_test[0]
X_test = data_test[1:n_original]
X_test = X_test / 255.

# Train set
data_train = data[nTest:m_original].T
Y_train = data_train[0]
X_train = data_train[1:n_original]
X_train = X_train / 255.
_,m_train = X_train.shape

# _,m_train = X_train.shape

# Y_train = Y_train / 255. # Normalize (to avoid exp overflow)
```

```
Y_train
```

```
array([0, 8, 0, ..., 9, 3, 7])
```

The NN will feature a straightforward two-layer design. The input layer $a[0]$ will have 784 units, which match to the 784 pixels in each 28x28 input picture. A hidden layer $a[1]$ will have 10 units with ReLU activation, and the output layer $a[2]$ will have 10 units with softmax activation corresponding to the ten digit classes.

Forward propagation

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = g_{\text{ReLU}}(Z^{[1]}))$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = g_{\text{softmax}}(Z^{[2]})$$

Backward propagation

$$dZ^{[2]} = A^{[2]} - Y$$

$$dW^{[2]} = \frac{1}{m}dZ^{[2]}A^{[1]T}$$

$$dB^{[2]} = \frac{1}{m}\Sigma dZ^{[2]}$$

$$dZ^{[1]} = W^{[2]T}dZ^{[2]}.*g^{[1]\prime}(z^{[1]})$$

$$dW^{[1]} = \frac{1}{m}dZ^{[1]}A^{[0]T}$$

$$dB^{[1]} = \frac{1}{m}\Sigma dZ^{[1]}$$

Parameter updates

$$W^{[2]} := W^{[2]} - \alpha dW^{[2]}$$

$$b^{[2]} := b^{[2]} - \alpha db^{[2]}$$

$$W^{[1]} := W^{[1]} - \alpha dW^{[1]}$$

$$b^{[1]} := b^{[1]} - \alpha db^{[1]}$$

Vars and shapes

Forward prop

- $A^{[0]} = X$: 784 x m
- $Z^{[1]} \sim A^{[1]}$: 10 x m
- $W^{[1]}$: 10 x 784 (as $W^{[1]}A^{[0]} \sim Z^{[1]}$)
- $B^{[1]}$: 10 x 1
- $Z^{[2]} \sim A^{[2]}$: 10 x m
- $W^{[1]}$: 10 x 10 (as $W^{[2]}A^{[1]} \sim Z^{[2]}$)
- $B^{[2]}$: 10 x 1

Backprop

- $dZ^{[2]}$: 10 x m ($A^{[2]}$)
- $dW^{[2]}$: 10 x 10
- $dB^{[2]}$: 10 x 1
- $dZ^{[1]}$: 10 x m ($A^{[1]}$)
- $dW^{[1]}$: 10 x 10
- $dB^{[1]}$: 10 x 1

Functions

Below are the list of functions that will be used

```
# Inititate parameters
def initateParameters():
    # Get arbitrary weights and biases
    # Substract 0.5 since randn generate values from 0 to 1
    W1 = np.random.rand(10, 784) - 0.5 # Matrix of 70 x 784
    b1 = np.random.rand(10, 1) - 0.5 # Vector of 10 x 1
    W2 = np.random.rand(10, 10) - 0.5 # Matrix of 10 x 10
    b2 = np.random.rand(10, 1) - 0.5 # Vector of 10 x 1
    return W1, b1, W2, b2

# ReLU activation function
def ReLU(Z):
    return np.maximum(Z, 0)

# ReLU derivative activation function
def ReLU_deriv(Z):
    return Z > 0

# Softmax activation function
def softmax(Z):
    A = np.exp(Z) / sum(np.exp(Z))
    return A

# Sigmoid activation function
#def Sigmoid(Z):
#    S = 1 / (1 + np.exp(-Z))
#    return S

# Sigmoid derivative activation function
# def Sigmoid_deriv(Z):
#    S_deriv = Sigmoid(Z) * (1 - Sigmoid(Z))
#    return S_deriv

# Forward propagation
def forwardPropagation(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2

# Complete Y data, return the activation
def oneHot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1)) # Since there are 0 - 9 numbers = 10
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
    return one_hot_Y

# Backward propagation
def backwardPropagation(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = oneHot(Y)
    dZ2 = A2 - one_hot_Y
    dW2 = 1 / m_original * dZ2.dot(A1.T)
    db2 = 1 / m_original * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_deriv(Z1)
    dW1 = 1 / m_original * dZ1.dot(X.T)
    db1 = 1 / m_original * np.sum(dZ1)
    return dW1, db1, dW2, db2

# Update parameters
def updateParameters(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2
```

```
# Return prediction
def getPredictions(A2):
    return np.argmax(A2, 0) # Return the index of the maximum argunment, thus, the predicted number index of th

# Prediction accuracy
def getAccuracy(predictions, Y):
    print(predictions, Y)
    return np.sum(predictions == Y) / Y.size # Accuracy of the predicted number

# Gradient descent function
def gradientDescent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = initateParameters()
    # Loop through the amount of iterations we set
    for i in range(iterations):
        Z1, A1, Z2, A2 = forwardPropagation(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backwardPropagation(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = updateParameters(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        # For every iterations, print prediction
        if i % 100 == 0:
            print("Iteration: ", i)
            predictions = getPredictions(A2)
            print(getAccuracy(predictions, Y))
    return W1, b1, W2, b2
```

Execute code

```
W1, b1, W2, b2 = gradientDescent(X_train, Y_train, 0.10, 1000)
```

```
Iteration:  0
[5 1 6 ... 3 1 5] [0 8 0 ... 9 3 7]
0.11492682926829269
Iteration:  100
[0 8 0 ... 7 3 3] [0 8 0 ... 9 3 7]
0.6178780487804878
Iteration:  200
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.766609756097561
Iteration:  300
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.8168048780487804
Iteration:  400
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.8408048780487805
Iteration:  500
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.8542195121951219
Iteration:  600
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.8639756097560976
Iteration:  700
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.8710731707317073
Iteration:  800
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.8760731707317073
Iteration:  900
[0 8 0 ... 9 3 7] [0 8 0 ... 9 3 7]
0.8798536585365854
```

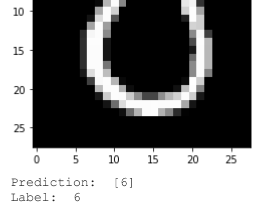
Test an image

```
# To make a singular prediction with the weights and biases calculated
def makePredictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forwardPropagation(W1, b1, W2, b2, X)
    predictions = getPredictions(A2)
    return predictions

# Test prediction
def testPredictions(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = makePredictions(X_train[:, index, None], W1, b1, W2, b2)
    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()
```

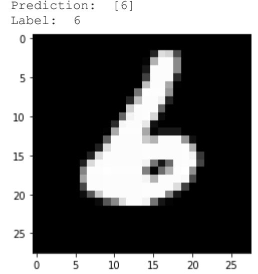
Check some examples of the train set

```
testPredictions(0, W1, b1, W2, b2)
testPredictions(1000, W1, b1, W2, b2)
testPredictions(40123, W1, b1, W2, b2)
testPredictions(40000, W1, b1, W2, b2)
```

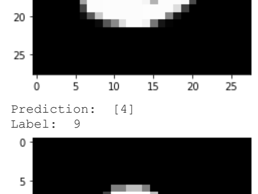
```
Prediction:  [0]
Label:  0
```



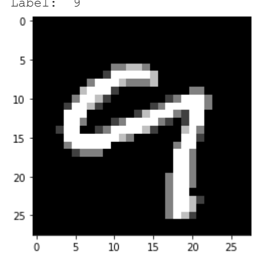

```
Prediction:  [6]
Label:  6
```




```
Prediction:  [4]
Label:  9
```




```
Prediction:  [9]
Label:  9
```



Check for the test set

```
test_set_predictions = makePredictions(X_test, W1, b1, W2, b2)
getAccuracy(test_set_predictions, Y_test)
```

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

```
0.881
```