# Understanding Neural Networks by Building One from Scratch using NumPy

[[Mohamed Yosef](https://medium.com/@mohamedyosef101?source=post_page-----61500b9ca882--------------------------------)](https://medium.com/@mohamedyosef101?source=post_page-----61500b9ca882--------------------------------)

[Mohamed Yosef](https://medium.com/@mohamedyosef101?source=post_page-----61500b9ca882--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F122eca15ad5c&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40mohamedyosef101%2Funderstanding-neural-networks-by-building-one-from-scratch-using-numpy-61500b9ca882&user=Mohamed+Yosef&userId=122eca15ad5c&source=post_page-122eca15ad5c----61500b9ca882---------------------post_header-----------)

6 min read

·

Dec 15

4

Neural networks have become essential for modern machine learning, powering systems from image recognition to language translation. While frameworks like TensorFlow and PyTorch simplify development, building a neural network from scratch deepens understanding of how they actually work.

In this project, NumPy was used to build and train a basic neural network to recognize handwritten digits from the MNIST dataset. The goal was exposing the fundamental concepts, not maximizing performance.

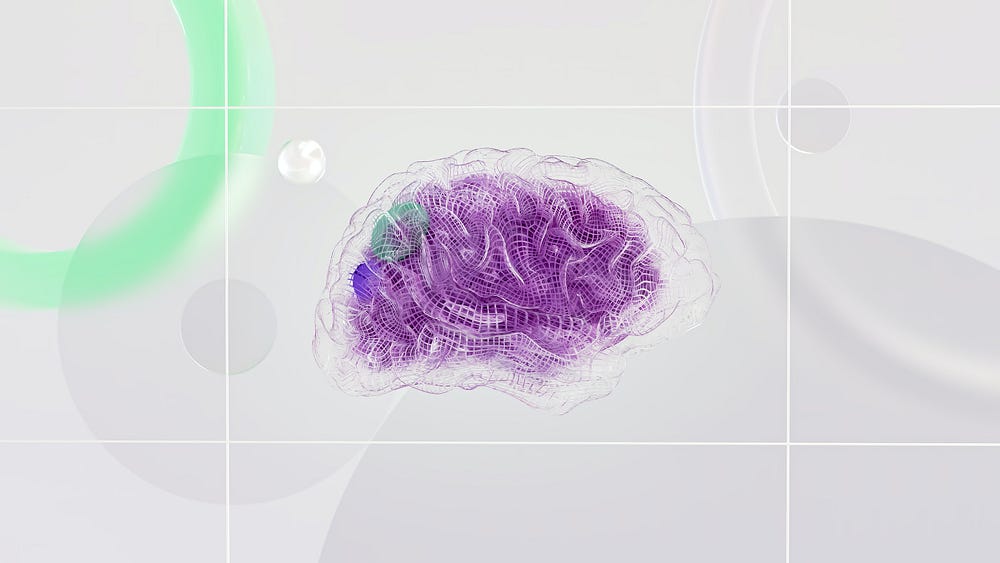


Photo by [Google DeepMind](https://unsplash.com/@googledeepmind?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral)

# Step 0. Set it up

It’s my basic step to import the libraries and load the data. Click [here](https://www.kaggle.com/competitions/digit-recognizer/data?select=train.csv) to download the dataset and start the work.

PS. You can see all of the code on [GitHub](https://github.com/mohamedyosef101/deep-learning-101/blob/3a3d3bbbaccefb28a64730c2d9d6f243b3475808/01-mnist-neural-network-with-numpy.ipynb)or [Kaggle](https://www.kaggle.com/code/mohamedyosef101/mnist-neural-network-with-numpy). Choose what you like :)

# import the libraries  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns  
  
# customize the style  
pd.options.display.float\_format = '{:.5f}'.format  
pd.options.display.max\_rows = 12  
  
# load the data  
filepath = '/kaggle/input/digit-recognizer/train.csv'  
df = pd.read\_csv(filepath)  
  
df.head()

# Step 1. Move the data to NumPy

NumPy was chosen because it provides a nimble, flexible, high-performant environment to build and experiment with neural network fundamentals rapidly. Mastering NN basics with NumPy builds transferable skills and intuition for working in any math-centric ML domain.

# put the dataset in a numpy array  
data = np.array(df)  
  
# computing the shape   
m, n = data.shape  
  
# **NOTE:**   
# m is amout of rows, n is the amount of columns  
  
print("No. of rows(m) =", m, "\nNo. of columns(n) =", n)  
  
# shuffling the data before splitting  
np.random.shuffle(data)

**Shuffling**is randomly reordering the samples in the data which reduces sampling bias and overfitting so the result will be better and fast models.

## Why did I compute the shape before shuffling?

Knowing m and n allows us to shuffle properly while retaining the ability to undo the transformation by using the original shape. Capturing data shape makes the shuffling process reversible.

# Step 2. Splitting the data

**Training Set**— Used to teach the neural network by adjusting the weights through backpropagation. The model sees and learns from these examples.

**Validation Set**— Used to evaluate how well the model performs on data it hasn’t seen before. This simulates real-world performance.

# splitting data into 75% training and 25% validation  
  
# 25% for validation  
val\_data = data[0: 10500].T  
y\_val = val\_data[0] # first column is the target  
X\_val = val\_data[1:n] # those are the features  
  
# 75% for training  
train\_data = data[10500: m].T  
y\_train = train\_data[0]  
X\_train = train\_data[1: n]  
  
# see the new shapes  
print(train\_data.shape, y\_train.shape)

You can see that there is a **T** at val\_data = data[0: 10500].T which transpose the data so that each data sample is a row, while each feature/attribute is a column

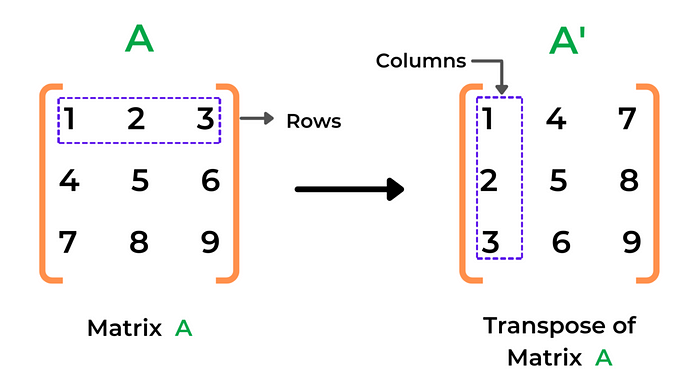


image from [allinpython.com](https://allinpython.com/transpose-of-a-matrix-in-python-with-user-input/)

## Why did I do that?

The transpose via .T creates a row-wise structure with the target variable separate from the feature values for each validation data sample. This setup enables easier access to y vs X as needed when training and evaluating the neural network.

# Step 3. Data Normalization

Neural nets learn better/faster on normalized data. Normalization makes the distribution of values consistent, meaning the network can learn more robust patterns not driven solely by outliers.

# Now the min and max values in order to apply the scale  
min\_val = data.min()  
max\_val = data.max()  
  
print(min\_val, max\_val)

This min-max scaling method preserves all original information with the simplified 0–1 range while retaining proportional differences. It’s straightforward and intuitive.

# Now let's apply the scale of 0 and 1  
X\_val = X\_val / 255  
X\_train = X\_train / 255  
  
# check if the normalisation worked  
X\_val.max()

I essentially divided the training set into training and validation sets and applied the scaling by dividing the data values by the max\_val, which is 255.

# Step 4. Model Building

The key components are:

* Input Layer: 784 nodes representing the 28x28 pixel MNIST images
* 1 Hidden Layer: 10 nodes with ReLU activation
* Output Layer: 10 nodes representing probability distribution over 10 digits using softmax.

## 4.1. Initializers

Initializes the weight matrices (W1, W2) and bias vectors (b1, b2) to random values.

def init\_params():  
 W1 = np.random.normal(size=(10, 784)) \* np.sqrt(1./(784))  
 b1 = np.random.normal(size=(10, 1)) \* np.sqrt(1./10)  
 W2 = np.random.normal(size=(10, 10)) \* np.sqrt(1./20)  
 b2 = np.random.normal(size=(10, 1)) \* np.sqrt(1./(784))  
 return W1, b1, W2, b2

## 4.2. Acitivation functions

* **ReLU** thresholds inputs at 0 to introduce nonlinearity.
* **Softmax** squashes outputs to probability-like values that sum to 1. Used for multi-class classification.

def ReLU(Z):  
 return np.maximum(Z, 0)  
  
def softmax(Z):  
 A = np.exp(Z) / sum(np.exp(Z))  
 return A

## 4.3. Forward Propagation

Performs the forward pass through the network. It takes in the parameters, input data X, calculates the linear combinations with weights/biases, applies activations, and returns activations & pre-activation values for backprop.

def forward\_prop(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

## 4.4. Backward Propagation

Performs the backward pass to calculate gradients. Uses prediction error to estimate parameter gradients.

# The derivative of ReLU (to be used in backpropagation).  
def ReLU\_deriv(Z):  
 return Z > 0  
  
# Converts a vector of class index labels Y   
# to a one-hot encoded matrix.  
def one\_hot(Y):  
 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  
  
# The BACKWARD propagation  
def backward\_prop(Z1, A1, Z2, A2, W1, W2, X, Y):  
 one\_hot\_Y = one\_hot(Y)  
 dZ2 = A2 - one\_hot\_Y  
 dW2 = 1 / m \* dZ2.dot(A1.T)  
 db2 = 1 / m \* np.sum(dZ2)  
 dZ1 = W2.T.dot(dZ2) \* ReLU\_deriv(Z1)  
 dW1 = 1 / m \* dZ1.dot(X.T)  
 db1 = 1 / m \* np.sum(dZ1)  
 return dW1, db1, dW2, db2

# 4.5. Final touches

1. **Updating the parameters** by applying the gradients, using a learning rate alpha.
2. **Getting class predictions** by taking the argmax of outputs.
3. **Comparing predictions** to true labels to calculate classification accuracy.
4. Finally, **put it all together** to iteratively train the network with gradient descent. Calculates the gradients, updates parameters, and repeats for num iterations.

# 1. Updating the parameters  
def update\_params(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  
  
# 2. Class Predictions  
def get\_predictions(A2):  
 return np.argmax(A2, 0)  
  
# 3. Accuracy for evaluation  
def get\_accuracy(predictions, Y):  
 print(predictions, Y)  
 return np.sum(predictions == Y) / Y.size  
  
# 4. Put it all together  
def gradient\_descent(X, Y, alpha, iterations):  
 W1, b1, W2, b2 = init\_params()  
 for i in range(iterations):  
 Z1, A1, Z2, A2 = forward\_prop(W1, b1, W2, b2, X)  
 dW1, db1, dW2, db2 = backward\_prop(Z1, A1, Z2, A2, W1, W2, X, Y)  
 W1, b1, W2, b2 = update\_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)  
 if i % 10 == 0:  
 print("Iteration: ", i)  
 predictions = get\_predictions(A2)  
 print(get\_accuracy(predictions, Y))  
 return W1, b1, W2, b2

# Step 5. Starting training and evaluating

After the model parameters have been learned, we want to assess performance on data that has been completely held out.

# Put the functions on action  
W1, b1, W2, b2 = gradient\_descent(X\_train, y\_train, 0.10, 500)

*~ 90% accuracy*

If you are looking for a higher score checkout my notebook about[**how to build your first neural network with Keras**](https://www.kaggle.com/code/mohamedyosef101/build-your-first-neural-network/edit/run/153871112)**.**

# Useful Resources

## For using NumPy

* Samson Zhang. 2020. [Simple MNIST NN from scratch with numpy](https://www.kaggle.com/code/wwsalmon/simple-mnist-nn-from-scratch-numpy-no-tf-keras/notebook). Kaggle

## Tutorials on Deep Learning

* Misra Turp. 2023. [50 Days of Deep Learning](https://youtube.com/playlist?list=PLM8lYG2MzHmQn55ii0duXdO9QSoDF5myF&si=s1pe9cRtFjKCPqR5). YouTube.
* Grant Sanderson. 2017. [Neural Networks, Deep Learning](https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi). YouTube.

## The math behind neural networks

* Grant Sanderson. 2016. [The Essence of Linear Algebra](https://youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab&si=ieGLDzRU2Ln9L0RO). 3Blue1Brown. YouTube.
* Kimberly Brehm. 2019. [Linear Algebra (Entire Course)](https://youtube.com/playlist?list=PLl-gb0E4MII03hiCrZa7YqxUMEeEPmZqK&si=TT-bemenvZWQIGG2). YouTube.
* Grant Sanderson. 2016. [Multivariable Calculus](https://www.khanacademy.org/math/multivariable-calculus). Khan Academy.
* [Statistics and Probability](https://www.khanacademy.org/math/statistics-probability). 2008. Khan Academy.

Thanks for your time!