Content:
1. Definition of a neuron
2. Neural network layers analogy
3. Weights
4. Activation functions
5. Gradient Descent

Reference: 3Blue1Brown video
Pause! What Do You See?

How do you “KNOW” it is seven? In other words, what is your proof?

But how can the computer tell? Difficulty upgrade!
Neuron
A Neuron = placeholder for a number
Network Layers
Before Going Forward...

What does these layers do???

9 = \[ \quad \]

7 = \[ \quad \]
And how to recognize patterns?

Well, smaller edges
And how to recognize patterns?

Well, smaller edges

Although this is not exactly how neural networks “learn,” this is an intuitive (or human method) way to visualize the layers.
Weights & Parameters
Ideally, we want the pixels (neurons) at the region to be highly positive and all the rest to be zero!

Even better, if we want an edge around the region, then we want those respective neurons to be negative.
Activation Functions
Standardized Output

\[ \sigma(w_1a_1 + w_2a_2 + \ldots + w_na_n + 10) \]

\[ f(x) = \frac{1}{1 + e^{-c_1(x - c_2)}} \]

Sigmoid, AKA logistic curve
Hi neuron,
Please light up only if the weighted sum is greater than ten!

$$\sigma(w_1a_1 + w_2a_2 + \ldots + w_na_n + 10)$$
And... This is just one neuron! All 784 neurons in our example have weights and biases. This resulted in 13,002 parameters!
LEARN: Find the right (most optimal) weights and biases.
Gradient Descent
Everything is CALCULUS!

Finding maximum/minimum
Cost Function: the Error

What we tried to minimize

\[ \sum_{i=0}^{n} (x_{p,i} - x_{a,i})^2 \]
How exactly does this work?

1. Prepare a training set (large!) with labels (supervised)
2. Initialize weights and biases randomly
3. Calculate the cost
4. Use gradient descent to start minimizing (decreasing the cost)

Well, it’s easy to find the minimum point of a two-dimensional function. But what do we do if it’s 13002 dimensions?
Gradient Descent

Core idea: the function decreases the fastest at the direction of the negative gradient:

$$\nabla f(x, y, z) = \langle f_x, f_y, f_z \rangle$$

AH, my multivariable class actually helps!
def simple_gd(gradient, start, learn_rate, n_iter=50, tolerance=1e-06):
    
    gradient: function with input vector and output gradient
    start: random starting point
    learn_rate: eta, controls magnitude of vector update
    n_iter: number of iterations
    tolerance: terminates the program if the difference is reached
    
    vector = start
    vector_lst = [start]
    for _ in range(n_iter):
        vector -= learn_rate * gradient(vector)
        vector_lst.append(vector)
        if abs(learn_rate * gradient(vector)) <= tolerance:
            break
    return vector, vector_lst
Stochastic Gradient Descent

More efficient!

A drunk man walking down a hill... But faster!
Next Steps

1. Calculate gradient (partial derivatives): Backpropagation
2. Understanding what actually happened in the learning process
3. Understanding stochastic gradient descent and learning rate’s role
Thank you!