Reading 9

☆ Video: What is a neural network

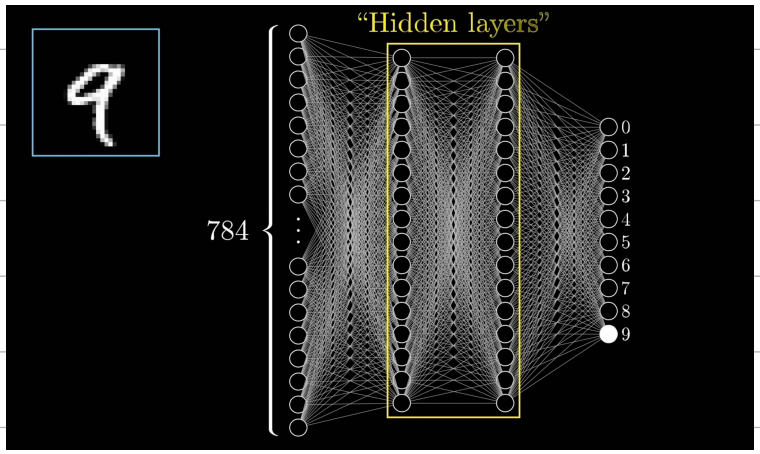
- Putting together a neural network to recognize hand written digits.
- The plain vanilla neural network is the multilayer perceptron

- A neuron is a thing that holds a number between 0 and 1.
  - ↳ This number is called the activation.
  - ↳ A neuron is "lit up" when activation is high.

- We have a 28×28 grid of pixels for our images.
  - ↳ Each pixel is a neuron.
  - ↳ So we have 784 neurons in our first layer.

- The last layer will have 10 neurons (one for each digit 0-9).
  - ↳ The activations in these neurons represents how much the system thinks that a given image corresponds to a given digit.

- There are also the "hidden" layers in between.

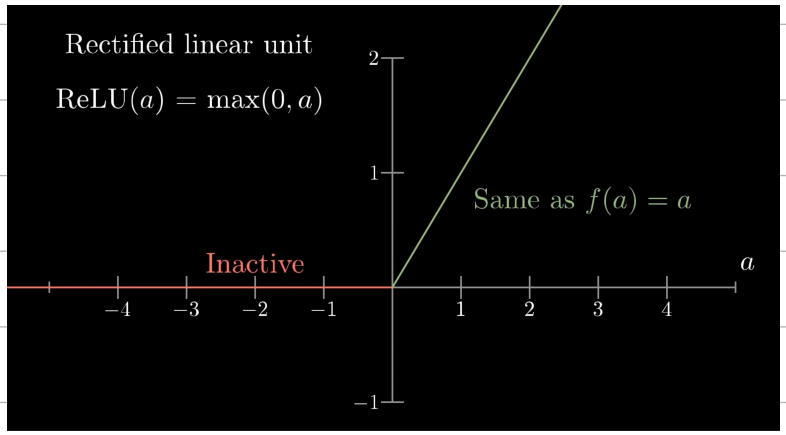


- Activations in one layer impact the activations of neurons in the next layer.

- Why expect layers?
  - ↳ Maybe the neurons in the second to last layer each represent a subpart of a digit.
    - ↳ i.e. the loop at the top of 9 and 8
    - ↳ or the longest vertical line in 4.

- Each connection from neuron in present layer to neuron in next layer has a <u>weight</u>.
  - ↳ We sum $\text{weight}_i \cdot \text{activation}_i$ for each neuron i in the present layer to determine the activation of <u>one</u> of the neurons in the next layer.
    - ↳ $\text{activation} = w_1 a_1 + \ldots + w_n a_n$
  - ↳ This weighted sum could be any number
    - ↳ so we use a sigmoid function to map the weighted sum $\in \mathbb{R}$ to an activation $\in [0, 1]$
    - ↳ you can also add a <u>bias</u> into the weighted sum to affect when the neuron "lights up"
      - ↳ (ex): maybe you only want the neuron to "light up" if weighted sum is larger than 10.

- Note: Sigmoid is kind of outdated, modern networks use ReLU to serve the same purpose since it is easier to train



↳ So determining activation of a neuron in next layer:

$$\sigma(w_1 a_1 + w_2 a_2 + \dots w_n a_n + b)$$

Sigmoid — weighted sum — bias

↳ This equation is more commonly seen as the following:

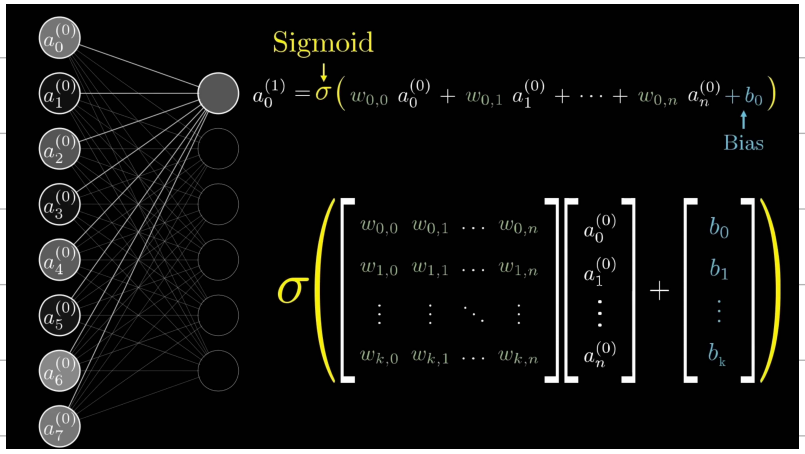
↳ organize all activations in present layer into a column vector.

↳ organize all weights between layers into a matrix

↳ each row corresponds to the connections between present layer and a particular neuron in the next layer.

↳ put all biases into a column vector

↳ obtain the following:



- So final equation is! $a^{(1)} = \sigma(Wa^{(0)} + b)$

$a^{(1)}, a^{(0)} \in \mathbb{R}^n$, $W \in M_{m \times n}$

$b \in \mathbb{R}$, $\sigma : \mathbb{R} \to [0,1]$

- So a neuron is more accurately a function that takes in the outputs of all the neurons in the previous layer and spits out a number between 0 and 1.

- So you can think of our network as a function that takes in 784 (numbers as) inputs and spits out 10 (numbers as) outputs.
  - ↳ It has 13,002 parameters (weights and biases)

★ Video: Gradient Descent, how neural networks learn:

Note: Same example as above (recognizing handwritten digits)

- First initialize all weights and biases randomely.
  - ↳ Get results (10 activations in final layer)
    - ↳ Compute "cost"
      - ↳ $\sum (\hat{y} - y)^2$, $\hat{y}$ is output activation, y is desired activation
        - ↳ summing the squared difference of all 10 output neurons
      - ↳ this is the "cost" of a single training example.
        - ↳ small when model is confident
        - ↳ large when model is unsure
      - ↳ consider average cost across all training examples
      - ↳ the cost function takes in all weights and biases as input and outputs 1 number which is the cost. The parameters are all the training examples.
  - ↳ We want to minimize cost function, a.k.a. find the minimum
    - ↳ So we use gradient descent
      - ↳ Algorithm for gradient descent:
        1. Compute ∇C where C is the cost function
        2. Take small step in $-\nabla C$ direction
        3. Repeat

  - ↳ Each component of the gradient vector tells us 2 things:
    - ↳ The sign tells us whether the corresponding component of the input vector should be bumped or down
    - ↳ The relative magnitude of the components tell you which changes matter more.
      - ↳ "Some weights simply matter more"

- Remember you ideally want the global minimum of the cost function but in practice this is very difficult!
  - ↳ Gradient descent only gaurantees you will find a minimum, that minimum could be local or global.
    - ↳ finding a local minimum is much more likely
      - ↳ In practice, these local minimums work very well.

- In this network, the network can recognize digits but has NO idea how to draw them. (talking about MLP here)
  - ↳ This can be explained by the training set as well
    - ↳ The network thinks everything is a digit
      - ↳ So if you pass in random grid, it will still give output.

☆ Video: Backpropogation, intuitively

- Backpropogation is the algorithm for determining how a single training example would like to nudge the weights and biases.
  - ↳ Not just what should go up or down, but what relative proportions to those changes causes the most rapid decrease to the cost.

- Remember the components of our gradient vector tells us 2 things:
  - ↳ the magnitude of each component tells us how sensitive the cost function is to each weight and bias.
    - ↳ If one component has value 3.20 and another has value 0.10
      - ↳ then changing the weight/bias corresponding to the component with value 3.20 has 32 times greater difference on cost function then changing the weight/bias associated with 0.10

- Intuitive Walkthrough:

  - In determing the activation for a specific neuron remember how we do $\sigma\left(\sum w_i a_i + b\right)$.

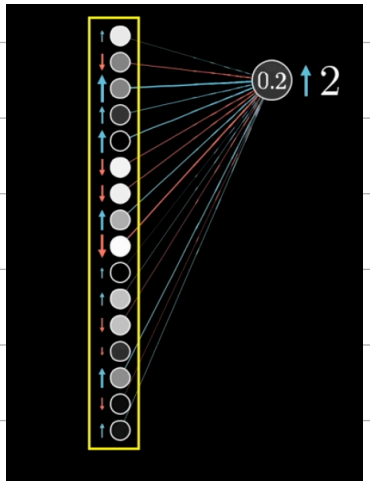
    ↳ The weights associated with the connections between the "brightest" neurons in the previous layer and the specific neuron we are considering in the present layer have more impact on the cost function.

- To make a specific neuron fire you have 3 options:

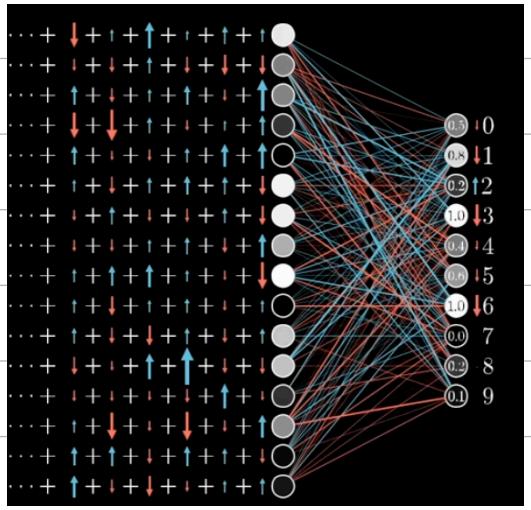
  1. Alter b
  2. Increase w_i in proportion with a_i
  3. Change a_i in proportion to w_i

- Then you sum the results of whatever method you pick from above with the results of all the other connections.

- Ex: Single Neuron



Ex: Entire layer



- You recursively apply the same process to all layers.

- Remember that is only how one training example impacts weights and biases.

- You do the same process for every other training example and record the changes they want to make to all weights and biases.
  - ↳ You then average all the results.

- This loosely speaking gives you the negative gradient of the cost function needed for gradient descent.

- This is computationally expensive, so stochastic gradient descent is used instead:
  - ↳ randomely shuffle your training data
  - ↳ divide it into a lot of mini-batches.
  - ↳ then you compute a "step" according to the mini-batch.
  - ↳ then you use a different mini-batch to compute the next step
  - ↳ this looks like the "drunken man's walk to the minimum"