Short Introduction To Neural Networks And Deep Learning

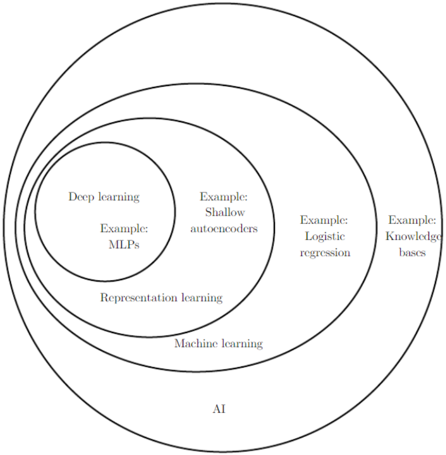
Mehadi Hassan, Shoaib Ahmed Dipu, Shemonto Das

BRAC University

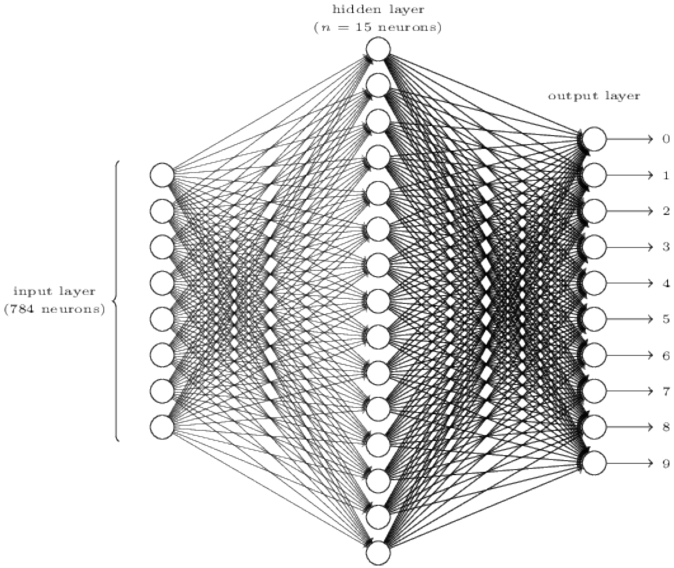
November 27, 2019

Overview

AI, ML, and Deep Learning: What's the Difference?



Neural Networks



Perceptrons

- Artificial Neurons
- A perceptron takes a several binary inputs and produces a single binary output

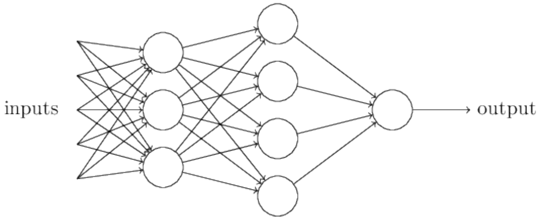
$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases} \tag{1}$$

Perceptrons

- Varying the weights and the threshold, we can get different models of decision-making

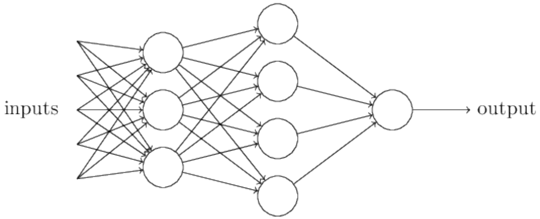
Perceptrons

- Varying the weights and the threshold, we can get different models of decision-making



Perceptrons

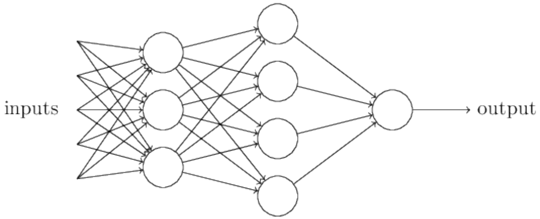
- Varying the weights and the threshold, we can get different models of decision-making



- Can weigh up different kinds of evidence in order to make decisions.

Perceptrons

- Varying the weights and the threshold, we can get different models of decision-making



- Can weigh up different kinds of evidence in order to make decisions.
- It should seem plausible that a complex network of perceptrons could make quite subtle decisions.

Perceptrons

Let's simplify the way we describe perceptrons.

$$\sum_j w_j x_j > \text{Threshold}$$

Perceptrons

Let's simplify the way we describe perceptrons.

$$\sum_j w_j x_j > \text{Threshold}$$

Simplifying is as a Dot Product,

$$\mathrm{w} \cdot \mathrm{x} \equiv \sum_j w_j x_j$$

Perceptrons

Let's simplify the way we describe perceptrons.

$$\sum_j w_j x_j > \text{Threshold}$$

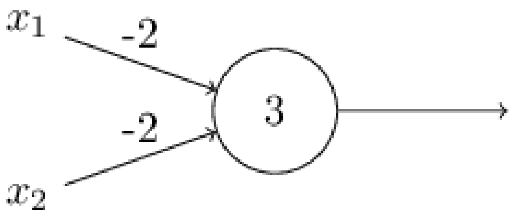
Simplifying is as a Dot Product,

$$\mathrm{w}\cdot\mathrm{x} \equiv \sum_j w_j x_j$$

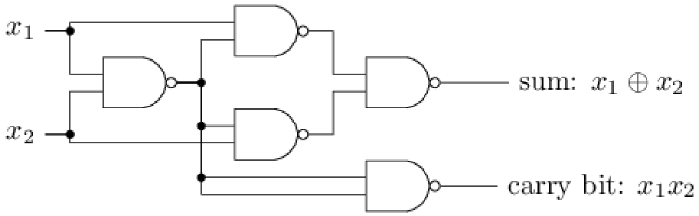
Moving the threshold to the other side of the inequality,

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \tag{2}$$

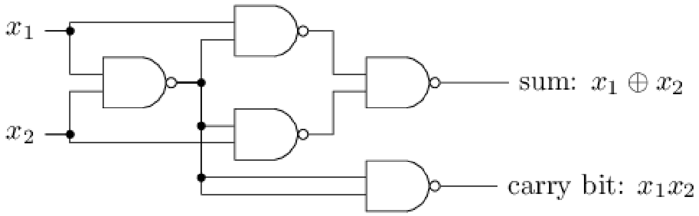
Perceptron implements a NAND gate!

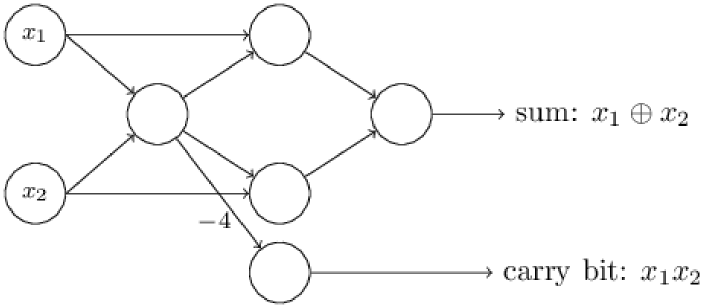


Perceptron implements a NAND gate!



Perceptron implements a NAND gate!



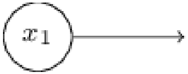


Perceptron implements a NAND gate!

This notation for input perceptrons, in which we have an output, but no inputs,

Perceptron implements a NAND gate!

This notation for input perceptrons, in which we have an output, but no inputs,



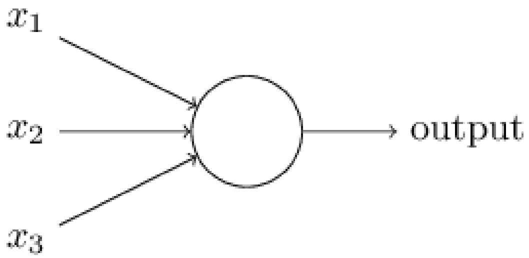
- Outputs a fixed value, not the desired value

Perceptron

- Reassuring - Can be powerful as any computing device
- Dissapointing - One may think it as a new type of NAND gate

Sigmoid Neurons

Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. That's the crucial fact which will allow a network of sigmoid neurons to learn.



Sigmoid Neurons

Here, output is σ(w*x+b), where σ is called the sigmoid function.

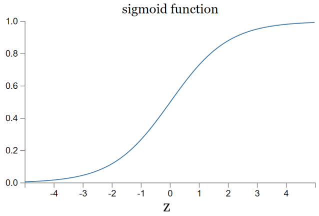
$$\sigma(z) \equiv \frac{1}{1+e^{-z}}. \tag{3}$$

To put it all a little more explicitly, the output of a sigmoid neuron with inputs x1,x2,. . . , weights w1,w2,. . . , and bias b is

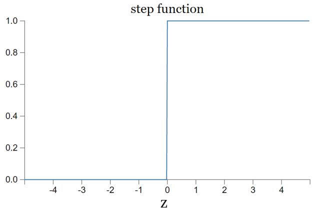
$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}. \tag{4}$$

Sigmoid Neurons

The exact form of isn't so important - what really matters is the shape of the function when plotted. Here's the shape:



This shape is a smoothed out version of a step function:

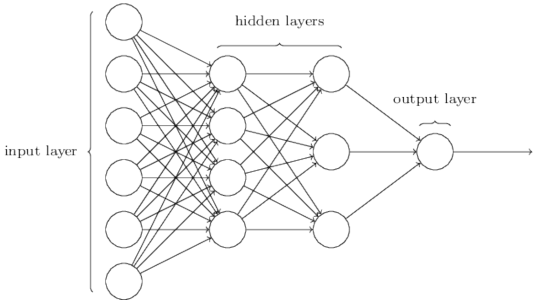


Sigmoid Neurons

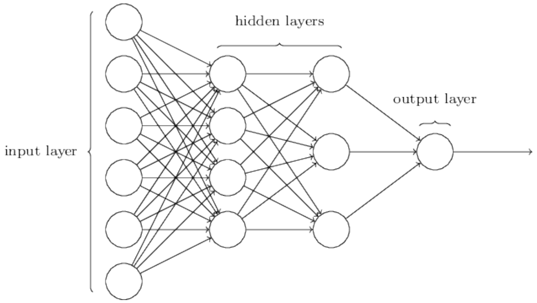
If σ had in fact been a step function, then the sigmoid neuron would be a perceptron, since the output would be 1 or 0 depending on whether w.x+b was positive or negative.
The smoothness of means that small changes Δw_j in the weights and Δb in the bias will produce a small change Δoutput in the output from the neuron. In fact, calculus tells us that Δoutput is well approximated by

$$\Delta \text{output} \approx \sum_j \frac{\partial \, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \, \text{output}}{\partial b} \Delta b, \tag{5}$$

The architecture of neural networks



The architecture of neural networks



Design of the input and output layers is straightforward but there can be quite an art to the design of the hidden layers.

The architecture of neural networks

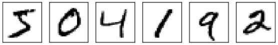
Feedforward :

- Output from one layer is used as input to the next layer
- No loops in the network
- Information is always fed forward, never fed back

Recurrent :

- Feedback loops exist
- This has been less influential than feedforward networks, in part because the learning algorithms for recurrent nets are less powerful.

A simple network to classify handwritten digits

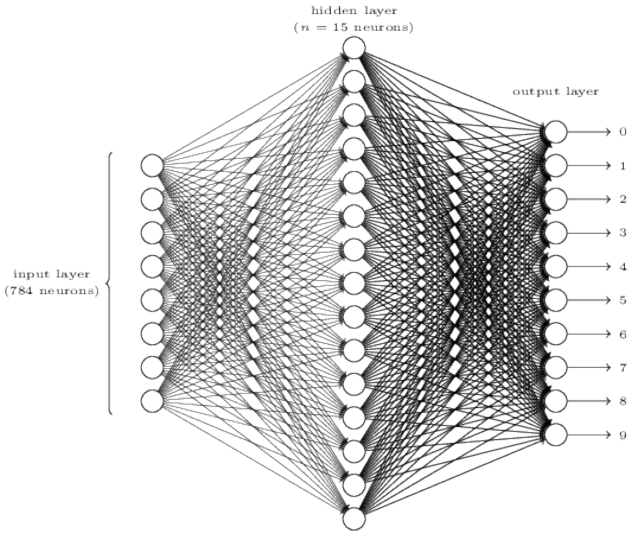


Our problem solving approach will be to solve the second problem, classifying individual digits.

One approach is to trial many different ways of segmenting the image, using the individual digit classifier to score each trial segmentation

A simple network to classify handwritten digits

To recognize individual digits we will use a three-layer neural network:



A simple network to classify handwritten digits

Input layer - Contains neurons encoding the values of the input pixels

Hidden layer - We denote the number of neurons in this hidden layer by n, and we'll experiment with different values for n.

Output layer - Contains 10 neurons. If the first neuron fires, i.e., has an output $\approx$1, then that will indicate that the network thinks the digit is a 0. And so on.

Learning with gradient descent

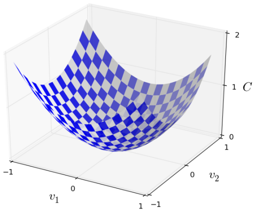
What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates y(x) for all training inputs x. To quantify how well we're achieving this goal we define a cost function.

$$C(w,b) \equiv \frac{1}{2n}\sum_{x} \|y(x) - a\|^2. \tag{6}$$

We want to find a set of weights and biases which make the cost as small as possible. We're going to develop a technique called gradient descent which can be used to solve such minimization problems. Then we'll come back to the specific function we want to minimize for neural networks.

Learning with gradient descent

Okay, let's suppose we're trying to minimize some function, C(v). This could be any real-valued function of many variables, v=v1,v2,...



Note that I've replaced the w and b notation by v to emphasize that this could be any function - we're not specifically thinking in the neural networks context any more. To minimize C(v) it helps to imagine C as a function of just two variables, which we'll call v1 and v2:
One way of attacking the problem is to use calculus to try to find the minimum analytically.

Learning with gradient descent

Let's think about what happens when we move the ball a small amount v1 in the v1 direction, and a small amount v2 in the v2 direction. Calculus tells us that C changes as follows:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2. \tag{7}$$

We're going to find a way of choosing v1 and v2 so as to make C negative. We denote the gradient vector by ∇C

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T. \tag{8}$$

The expression (7) for C can be rewritten as,

$$\Delta C \approx \nabla C \cdot \Delta v. \tag{9}$$

Learning with gradient descent

The real exciting about the equation is that it lets us see how to choose v so as to make C negative. In particular, suppose we choose

$$\Delta v = -\eta \nabla C, \tag{10}$$

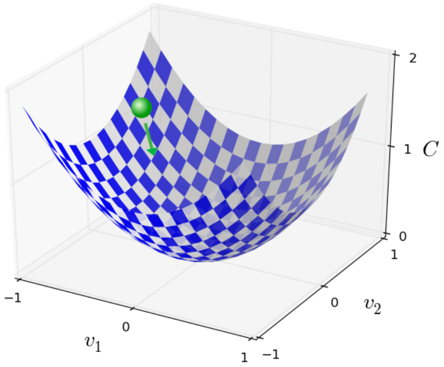
where η is a small, positive parameter (known as the learning rate). Then Equation (9) tells us that C≈−∇C·∇C=−‖∇C‖2. Because ‖∇C‖2≥0, this guarantees that C≤0, i.e., C will always decrease, never increase, if we change v according to the prescription in (10)

That is, we'll use Equation (10) to compute a value for v, then move the ball's position v by that amount:

$$v \rightarrow v' = v - \eta \nabla C. \tag{11}$$

Learning with gradient descent

Summing up, the way the gradient descent algorithm works is to repeatedly compute the gradient ∇C, and then to move in the opposite direction, "falling down" the slope of the valley. We can visualize it like this:



Learning with gradient descent

To make gradient descent work correctly, we need to choose the learning rate to be small enough that Equation (9) is a good approximation. Suppose in particular that C is a function of m variables, v1,. . . ,vm. Then the change C in C produced by a small change
$\Delta v = (\Delta v_1, \ldots, \Delta v_m)^T is$

$$\Delta C \approx \nabla C \cdot \Delta v, \tag{12}$$

where the gradient ∇C is the vector,

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \ldots, \frac{\partial C}{\partial v_m}\right)^T. \tag{13}$$

Just as for the two variable case, we can choose

$$\Delta v = -\eta \nabla C, \tag{14}$$

Learning with gradient descent

and we're guaranteed that our (approximate) expression (12) for C will be negative. This gives us a way of following the gradient to a minimum, even when C is a function of many variables, by repeatedly applying the update rule

$$v \rightarrow v' = v - \eta \nabla C. \tag{15}$$

Gradient descent can be viewed as a way of taking small steps in the direction which does the most to immediately decrease C.
The idea is to use gradient descent to find the weights wk and biases bl which minimize the cost in Equation (6)

Update Rule

Writing out the gradient descent update rule in terms of components, we have

$$
\begin{aligned}
w_k &\rightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k} \qquad (16)\\
b_l &\rightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l}. \qquad (17)
\end{aligned}
$$

Learning with gradient descent

To understand what the problem is, let's look back at the quadratic cost in Equation (6). Notice that this cost function has the form $C = \frac{1}{n}\sum_x C_x$ that is, it's an average over costs $C_x \equiv \frac{\|y(x)-a\|^2}{2}$ for individual training examples.

In practice, to compute the gradient ∇C we need to compute the gradients ∇C_x separately for each training input, x, and then average them,$\nabla C = \frac{1}{n}\sum_x \nabla C_x$

Stochastic Gradient Descent

Estimate the gradient ∇C by computing ∇Cx for a small sample of randomly chosen training inputs. By averaging over this small sample it turns out that we can quickly get a good estimate of the true gradient ∇C, and this helps speed up gradient descent, and thus learning.

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C, \tag{18}$$

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j}, \tag{19}$$

$$w_k \rightarrow w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{20}$$

$$b_l \rightarrow b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l}, \tag{21}$$

References

Michael Nielsen
Neural Networks and Deep Learning
Retrived from http://neuralnetworksanddeeplearning.com/

The End