Perceptrons

02-251 Spring 2019
April 9, 2019

1. Reference Reading

Two good textbooks on AI that contain this material:

- Russell, Norvig. Artificial Intelligence: A Modern Approach.
- Luger and Stubblefield, Artificial Intelligence.

Much of the material below is taken from the above two sources.

2. Neurons

2.1 A high-level view of neurons in the brain

- The neuron is the basic cell type that makes up the brain.
- Neurons have three main parts: a cell body (called the **soma**), a number of fibers branching off of the soma (called **dendrites**), and a long fiber called an **axon**.
- The dendrites and the axon connect to other neurons, forming a network of neurons.
- The neuron acts as a signal collector and emitter: input signals enter from the dendrites, are integrated by the soma and, once the total signal strength is large enough, a signal is sent down the axon. The signal sent down the axon is an electrical pulse called an action potential. This signal reaches other neurons, transmitting the signal to other neurons through connections between the axon and the dendrites of other neurons.
- The connection between an axon and a dendrite is called a **synapse**. Synapses can excite or inhibit the electric potential of the neuron to which they are connected.
- The strength of existing connections and the structure of connections (which neurons are connected) can change over time, which is what enables a collection of neurons to learn.
- Axons can be very long (centimeters to a meter).
- A neuron may be connected to thousands of other neurons.

2.2 Collections of neurons

- The human brain contains approximately 10^{11} neurons and 10^{14} synapses.
- Neurons fire on the scale of milliseconds, so relatively slowly, but operate in a massively parallel way — all the neurons are able to process information simultaneously.

- The neural network of the flatworm, *C. elegans* has been completely characterized. It has 302 neurons that are consistently connected in the same way. See `http://www.wormatlas.org/neuronalwiring.html`
- Efforts are underway to map the connections of larger brains using various techniques such as imaging.

3. The Perceptron

3.1 A simple model of a neuron

- An early model of a neuron is the perceptron.
- Perceptrons have many inputs $I_1, \ldots, I_n$ but only one output O.
- Each input I_i is associated with a weight w_i that represents the strength of the connection from the input. This models the strength of the synapse connecting to the perceptron. The weights are part of the neuron and not part of the input.
- The weight can be positive (for an excitatory connection) or negative (for an inhibitory connection)
- The output is either 1 or 0 depending on whether the total weighted input is above or below a given threshold θ. (Some people define the output as $+1$ or -1, which is equivalent.)
- In other words:

$$O = \mathbf{1}\left(\sum_i w_i I_i \geq \theta\right) = \mathbf{1}(w \cdot I \geq \theta),$$

  where w and I are vectors collecting the weights and inputs, and $\mathbf{1}(x)$ is the indicator function that returns 1 if x is true and 0 if x is false.

3.2 Functions perceptrons can represent

- Perceptrons can represent various functions (McCulloch and Pitts (1942)):

  AND output 1 if all the inputs are 1: Set each $w_i = 1$ and $\theta = n$

  OR output 1 if any of the inputs are 1: Set each $w_i = 1$ and $\theta = 1$

  NOT output 1 only if all the inputs are 0: Set each $w_i = -1$ and $\theta = -0.5$. If all the inputs are 0, the total weight will be 0, which is > -0.5, so the output will be 1. If any input is 1, then the total weight will be ≤ -1, so the output will be 0. (When $n = 1$, this is the familiar NOT function.)

  NAND Output 1 if some input is 0: Set each $w_i = -1$ and $\theta = -n + 0.5$. If all the inputs are 1, then the total weight will be $-n$, which is $< -n + 0.5$, so the output will be 0. If some input is 0, then the total weight will be at least $-(n-1)$, which is $> -n + 0.5$, so the output will be 1.

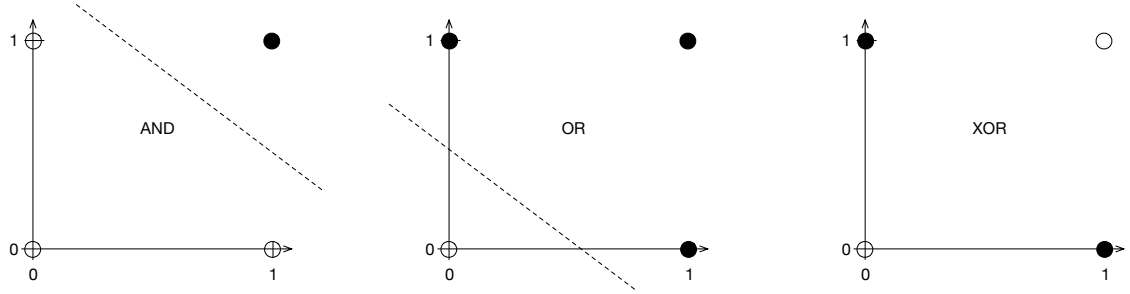
  Majority output 1 if a majority of the inputs are 1: Set each $w_i = 1$ and $\theta = n/2$

3.3 Perceptrons cannot represent all functions

- XOR: Output 1 if exactly 1 input is 1.
- The problem is that by construction the perceptron can only separate linearly separable points because the perceptron equation is an equation of a line (for 2 inputs), plane (for 3 inputs) or hyperplane (for more than 3 inputs). E.g.:

$$w_1 I_1 + w_2 I_2 + w_3 I_3 + \theta = 0$$

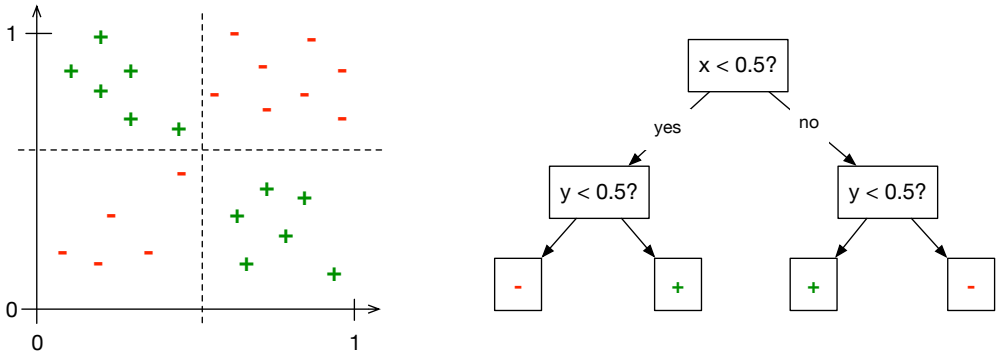
- The input $\langle I_1, \ldots, I_n \rangle$ is a point in n-dimensional space. The output is a label y of this point. To compute a function like AND, XOR, etc., the perceptron should output the right y for the given input point I.
- Points that are on one side of the perceptron hyperplane cause the output to be 0; points on the other side cause the output to be 1.



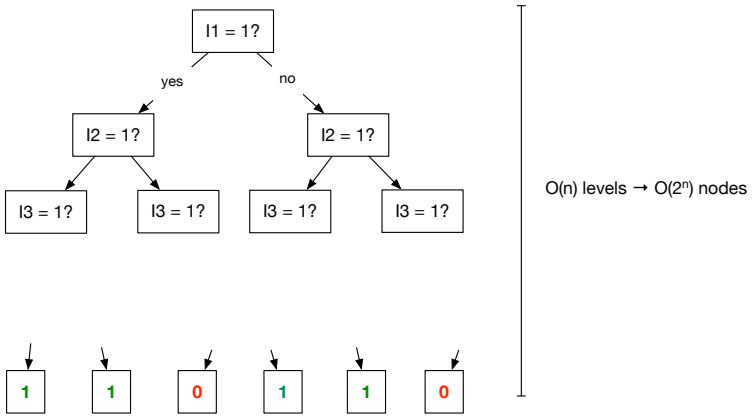
- So, perceptrons can only output the correct label for every point if the points are *linearly separable*: meaning that the points with $y = 0$ are on one side of the hyperplane, and those with $y = 1$ are on the other side.

3.4 Comparison to Decision Trees

- Decision trees are another model to represent classifiers. They are binary trees, where each node asks a YES/NO question about a point. The leaves are labeled with the predictions. For example, a node might ask "Is $I_{10} = 1$?" In the case of 0/1 labels, the leaves would each be labeled by either 0 or 1.
- Decisions trees are more powerful than perceptrons in that they can classify non-linearly separable points:



- However, sometimes that power comes at very large space and complexity cost. Consider again the **Majority** function:
  - Input = n bits $I_1, \ldots, I_n$.
  - positives = inputs with $> n/2$ bits set to 1.
  - negatives = inputs with $\leq n/2$ bits set to 1.
- We saw that a simple perception with n weights and 1 threshold can perfectly predict this set.
- Decision trees can perfectly predict it too, but look at how big the tree is:



3.5 Training Perceptrons

- Perceptrons can be trained on a set of examples: $(I^{(j)}, y^{(j)})$ where $y^{(j)}$ is the correct answer for the example — it's what we want the perceptron to output.
- To make things more uniform, we can convert the threshold θ into a weight w_0, if we assume that I_0 is always -1. Then we output 1 iff:

$$w_0(-1) + w_1 I_1 + \cdots + w_n I_n \geq 0$$
$$w_1 I_1 + \cdots + w_n I_n \geq w_0 = \theta$$

- If the output of a particular example $(\langle I_1, ..., I_n \rangle, y)$ is O, then define

$$Error = y - O.$$

- If $Error > 0$ then the output was too small, and we want to increase O.
- If $Error < 0$ then the output was too big, and we want to decrease O.
- Each term $w_i I_i$ in the perceptron contributes to the total weight. If $I_i > 0$, then increasing w_i will help us increase O; if $I_i < 0$, then decreasing w_i will help us increase O.
- So $I_i \times Error$ gives us a value with the right sign:
  - $Error > 0$: want to increase O:

    * $I_i > 0$: want to increase w_i ($I_i \times Error$ is positive)
    * $I_i < 0$: want to decrease w_i ($I_i \times Error$ is negative)
  - $Error < 0$: want to decrease O:
    * $I_i > 0$: want to decrease w_i ($I_i \times Error$ is negative)
    * $I_i < 0$: want to increase w_i ($I_i \times Error$ is positive)

- Therefore, we can update the weights to make this one example closer to correct using:

$$w_i \leftarrow w_i + \alpha \times I_i \times Error$$

- Here, α is the learning rate: a parameter we choose to decide how fast we update the weight. A bigger α will change the weights more quickly.
- Perceptron Training Algorithm:

```
Repeat:
   for each example (I, y):
      O = ComputeOutputOfPerceptron(I)
      Error = y - O
      for each weight w_i:
          w_i = w_i + alpha * I_i * Error
Until all the outputs are correct
```

- Theorem: If the input is linearly separable, the perceptron training algorithm will find weights that correctly label every point.
- Why? There is no local minimal in weight space.
- Intuition: Consider a set of separable points and a line ℓ given by the current weights. The total error of this line is $L_\ell = \sum |y_j - O_j|$. Suppose for simplicity that $O_j, y_j \in \{0, 1\}$. Then L_ℓ is the number of points that are incorrectly classified by the line. If you pick a line that has non-zero total error, there is always a direction you can move it in that will decrease the error by moving a point to the correct side, *assuming* the points are linearly separable. In other words, we can always shift ℓ towards the correct linear separating line ℓ^* in such a way that we don't increase the error. Therefore, the weight space has no local minima and the "gradient descent"-like perceptron training algorithm will find the global optimal (which is 0 by assumption).
- Note 1: if the inputs are NOT linearly separable, then nothing is guaranteed by this algorithm.
- Note 2: for linearly separable points, there are usually many possible separating lines. This algorithm can return any of them. The line of maximal margin seems like a better one to return. This is the one that has the most "wiggle room" before misclassifying a point. Support Vector Machines are a model that finds this line of maximal margin.

4. Other Activation Functions

- This analysis follows Luger and Stubblefield, Artificial Intelligence, pp 673–675.

- So far we have seen the standard perceptron that uses a simple threshold to determine the output. A downside of this is that if my total weight was $\theta - \epsilon$ — that is just below the activation threshold — then my observed error will be the same as if my total weight was $-\infty$.
- It makes sense then to replace the hard threshold with a softer threshold.
- A typical choice is the logistic function:

$$f(x) = \frac{1}{1 + e^{-\lambda x}}$$

- Here λ is a parameter that controls how sharp the transition is: when λ is large, the transition is very sharp; when it is small, the transition is smoother. Asymptotically, at very low x the function approaches 0 and at high x the function approaches 1.
- Why this function? It's nicely parameterized by λ, it's continuous, and it's differentiable. There are other possible functions that serve the same purpose.
- So now the output of the network is $f(\sum_i w_i I_i)$.
- Let's derive the update rule for the perceptron learning algorithm for an activation function f that is continuous and differentiable:

4.1 Update rule for other activation functions*

This section is slightly more advanced. You don't need to know it in detail, but if you work through it, it will give you some more intuition about the perceptron weight update rule.

- Define $SError = (1/2)(y_i - O_i)^2$ as the squared error of the perceptron's output O_i compared with the desired output y_i on a particular example $\langle I, y_i \rangle$. (The 1/2 is for mathematical convenience; you'll see why later.)
- We want to compute $\frac{\delta SError}{\delta w_k}$. That is, what is the rate of change in the error as we change a weight w_k?
- By the chain rule, we have:

$$\frac{\delta SError}{\delta w_k} = \frac{\delta SError}{\delta O_i} \times \frac{\delta O_i}{\delta w_k} \tag{1}$$

  We can compute the first term directly:

$$\frac{\delta SError}{\delta O_i} = \frac{\delta (1/2)(y_i - O_i)^2}{\delta O_i} = -(y_i - O_i) \tag{2}$$

  We get the negative because the derivative of $y_i - O_i$ with respect to O_i is -1, and the $1/2$ disappears by bringing the power 2 down.
- Now we want to compute $\frac{\delta O_i}{\delta w_k} = \frac{\delta f(\sum_i w_i I_i)}{\delta w_k}$. Again by the chain rule, where the derivative of $f(g(x))$ is $g'(x)f'(g(x))$, we have:

$$\frac{\delta f(\sum_i w_i I_i)}{\delta w_k} = I_k f' \left(\sum_i w_i I_k \right) \tag{3}$$

since I_k is the derivative of the inner function $\sum_i w_i I_k$. So, putting together (2) and (3) into (1), we have:

$$\frac{\delta SError}{\delta w_k} = -(y_i - O_i) I_k f' \left(\sum_i w_i I_k \right)$$

- We want to decrease the error, so we want to move in the direction opposite of this gradient, so we want to change w_k by some amount:

  $$-\alpha \frac{\delta SError}{\delta w_k} = \alpha (y_i - O_i) I_k f' \left(\sum_i w_i I_k \right)$$

  This leads to the update rule:

  $$w_k \leftarrow w_k + \alpha (y_i - O_i) I_k f' \left(\sum_i w_i I_k \right)$$

  Compare this to our original update rule: $w_k \leftarrow +\alpha \times Error \times I_k$. Since $Error = y_i - O_i$, this new update rule is the same, except we multiply by the derivative of the activation function!