Perceptron learning algorithm

repeat until convergence (or for some # of iterations):
for each training example \((f_1, f_2, \ldots, f_n, \text{label})\):
\[
prediction = b + \sum_{i=1}^{n} w_i f_i
\]
if \(prediction \times \text{label} \leq 0\): // they don’t agree
for each \(w_i\):
\(w_i = w_i + f_i \times \text{label}\)
\(b = b + \text{label}\)

Why is it called the “perceptron” learning algorithm if what it learns is a line? Why not “line learning” algorithm?
Our nervous system: 
the computer science view

- The human brain is a large collection of interconnected neurons
- A **neuron** is a brain cell
  - they collect, process, and disseminate electrical signals
  - they are connected via synapses
  - they **fire** depending on the conditions of the neighboring neurons

**A neuron/perceptron**

- $\sum g(in)$
  - $in = \sum w_i x_i$
  - Output $y$
  - Activation function

- How is this a linear classifier (i.e. perceptron)?

**Hard threshold = linear classifier**

- $g(in)$:
  - $\begin{cases} 
  1 & \text{if } in > b \\
  0 & \text{otherwise}
  \end{cases}$
- Output:
  - $\begin{cases} 
  1 & \text{if } \sum w_i x_i + b > 0 \\
  0 & \text{otherwise}
  \end{cases}$

**Neural Networks**

- Neural Networks try to mimic the structure and function of our nervous system

- People like biologically motivated approaches
Artificial Neural Networks

Node (Neuron/perceptron)

Edge (synapses)

our approximation

Node A

(Perceptron)

Node B

(Perceptron)

Weight \( w \)

output = \[
\begin{cases} 
1 & \text{if } \sum w_i x_i + b > 0 \\
0 & \text{otherwise}
\end{cases}
\]

\( W \) is the strength of signal sent between A and B.

If A fires and \( w \) is positive, then A stimulates B.

If A fires and \( w \) is negative, then A inhibits B.

Other activation functions

hard threshold:

\[
g(in) = \begin{cases} 
1 & \text{if } in > b \\
0 & \text{otherwise}
\end{cases}
\]

sigmoid

\[
g(x) = \frac{1}{1 + e^{-x}}
\]

tanh x

why other threshold functions?

Neural network

Individual perceptrons/neurons
Neural network

inputs

some inputs are provided/entered

Neural network

inputs
each perceptron computes and calculates an answer

Neural network

inputs

those answers become inputs for the next level

Neural network

inputs

finally get the answer after all levels compute
Activation spread

http://www.youtube.com/watch?v=Yq7d4ROvZ6I

Computation (assume 0 bias)

\[ g(\text{in}) = \begin{cases} 1 & \text{if } \text{in} > -b \\ 0 & \text{otherwise} \end{cases} \]

Computation

\[ -0.05 \times 0.02 = -0.07 \\
0.03 \\
-0.02 \\
0.01 \\
0.483 \times 0.5 + 0.495 = 0.7365 \\
0.495 \\
0.676 \\
-0.03 + 0.01 = -0.02 \]

Neural networks

Different kinds/characteristics of networks

How are these different?
Hidden units/layers

Feed forward networks

21

Hidden units/layers

Can have many layers of hidden units of differing sizes
To count the number of layers, you count all but the inputs

22

Alternate ways of visualizing

Sometimes the input layer will be drawn with nodes as well

24
Multiple outputs

Can be used to model multiclass datasets or more interesting predictors, e.g. images

Neural networks

Recurrent network
Output is fed back to input
Can support memory!
Good for temporal/sequential data

What does the decision boundary of a perceptron look like?
Line (linear set of weights)
What does the decision boundary of a 2-layer network look like? Is it linear?
What types of things can and can’t it model?
What does the decision boundary look like?

<table>
<thead>
<tr>
<th>X₁</th>
<th>X₂</th>
<th>X₁ XOR X₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Let $x_2 = 0$, then:

$-x_1 - 0.5 = 0$

$x_1 = -0.5$

What does this perceptron's decision boundary look like?
What does the decision boundary look like?

Let $x_2 = 0$, then:

$$x_1 - 0.5 = 0$$

Thus, $x_1 = 0.5$

Let $x_2 = 0$, then:

$$x_1 - 0.5 = 0$$

Thus, $x_1 = 0.5$

What operation does this perceptron perform on the result?

Fill in the truth table
What does the decision boundary look like?
What does the decision boundary look like?

Input $x_1$ 

Input $x_2$ 

Output $= x_1 \lor x_2$

linear splits of the feature space

combination of these linear spaces

This decision boundary?

Output

This decision boundary?

output = \begin{cases} 1 & \text{if } \sum w_i x_i + b > 0 \\ 0 & \text{otherwise} \end{cases}

This decision boundary?

output = \begin{cases} 1 & \text{if } \sum w_i x_i + b > 0 \\ 0 & \text{otherwise} \end{cases}

This decision boundary?

output = \begin{cases} 1 & \text{if } \sum w_i x_i + b > 0 \\ 0 & \text{otherwise} \end{cases}
What does the decision boundary look like?

Three hidden nodes
NN decision boundaries

Theorem 9 (Two-Layer Networks are Universal Function Approximators). Let $F$ be a continuous function on a bounded subset of $D$-dimensional space. Then there exists a two-layer neural network $\hat{F}$ with a finite number of hidden units that approximate $F$ arbitrarily well. Namely, for all $x$ in the domain of $F$, $|F(x) - \hat{F}(x)| < \epsilon$.

Put simply: two-layer networks can approximate any function

For DT, as the tree gets larger, the model gets more complex.

The same is true for neural networks:

more hidden nodes = more complexity

Adding more layers adds even more complexity (and much more quickly).

Good rule of thumb:

number of 2-layer hidden nodes $\leq$ \frac{number of examples}{number of dimensions}

Training

How do we learn the weights?

Training multilayer networks

perceptron learning: if the perceptron's output is different than the expected output, update the weights

gradient descent: compare output to label and adjust based on loss function

Any other problem with these for general NNs?

perceptron/linear model

neural network
Learning in multilayer networks

**Challenge**: for multilayer networks, we don't know what the expected output/error is for the internal nodes!

- perceptron/linear model
- neural network

how do we learn these weights?

Backpropagation: intuition

**Gradient descent method for learning weights by optimizing a loss function**

1. calculate output of all nodes
2. calculate the weights for the output layer based on the error
3. “backpropagate” errors through hidden layers

Key idea: propagate the error back to this layer

We can calculate the actual error here
Backpropagation: intuition

“backpropagate” the error:
Assume all of these nodes were responsible for some of the error
How can we figure out how much they were responsible for?

Backpropagation: the details

Gradient descent method for learning weights by optimizing a loss function
1. calculate output of all nodes
2. calculate the updates directly for the output layer
3. “backpropagate” errors through hidden layers

What loss function?
Backpropagation: the details

Gradient descent method for learning weights by optimizing a loss function

1. calculate output of all nodes
2. calculate the updates directly for the output layer
3. “backpropagate” errors through hidden layers

\[
\text{loss} = \frac{1}{2} \sum (y - \hat{y})^2 \quad \text{squared error}
\]