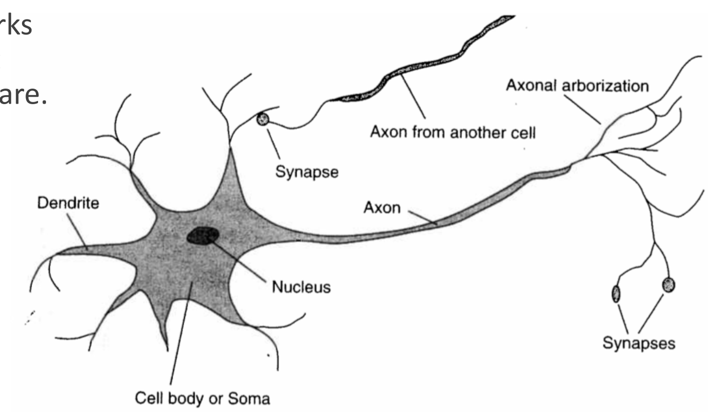
Neural Networks

MICHAEL WOLLOWSKI

Introduction

[Neural networks](NN) or more accurate Artificial Neural Networks (ANN) are software systems that are modelled on a brain's hardware.

Key element of a brain are:
- Cells and
- Synapses, which connect cells



Russell and Norvig: AIMA, 1st Ed. Figure 19.1

A Brief History of ANN

NN were first proposed by McCullough and Pitts; in 1943.

There model was non-learning.

In 1958, Frank Rosenblatt invented the perceptron, the first implemented artificial neural network.

A perceptron is a single layer NN

In 1969, Minsky and Papert wrote a very influential book, entitled: “Perceptron”

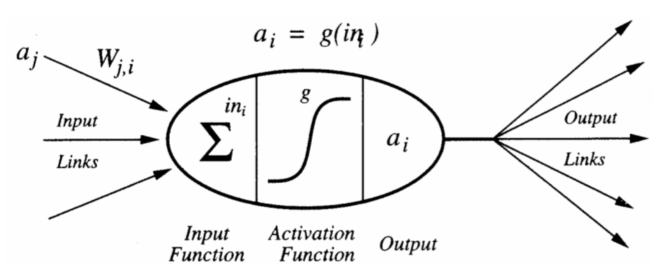
He book pointed out limitations of single layer networks.

Why nobody stood up and proposed multi-layer NN is truly puzzling.

Introduction

Mimicking NN, ANN consists of:

- Units
- Connections between units

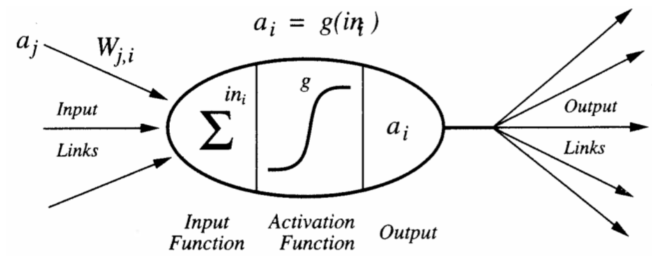


Russell and Norvig: AIMA, 1st Ed. Figure 19.4

Units

A unit consists of three parts:

1. An input function
2. An activation function
3. Output



Russell and Norvig: AIMA, 1st Ed. Figure 19.4

Input Function

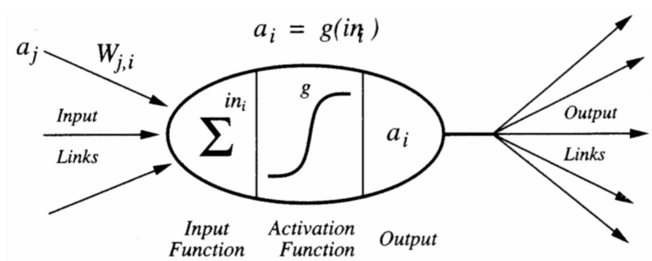
The input function simply sums each of the input values, multiplied by a weight associated with that input.

The inputs are either:

- Inputs to the NN
- Outputs from prior layers

The weights will be adjusted during training.

The weight matrix contains what might be called the knowledge of the NN



Russell and Norvig: AIMA, 1st Ed. Figure 19.4

Activation Function and Output

There are several popular activation functions:

- Step
- Sign
- Sigmoid
- tanh and
- ReLu

An activation function takes the input from the input function and produces the output value of the current unit.

The output is typically placed into vectors.

There is nothing exciting about the "Output" component of a unit.

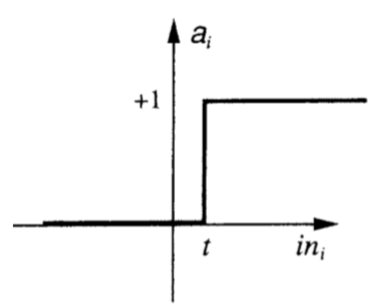
Step Activation Function

The step activation function has two output values: 0 and 1.

It produces an output of 1, if the input value is above above a certain threshold, *t*.

We will use this activation function to learn the basics of NN.

I do not think they are used in modern NN



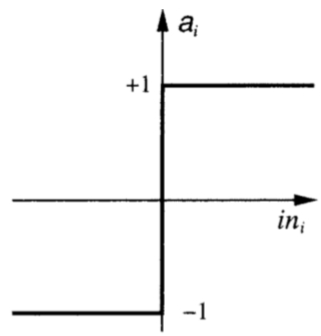
Russell and Norvig: AIMA, 1st Ed., Figure 19.5

Sign Activation Function

The sign activation function has two output values: -1 and 1.

It produces an output of 1, if the input value is above above 0.

I do not think they are used in modern NN



Russell and Norvig: AIMA, 1st Ed., Figure 19.5

Sigmoid Activation Function

The sigmoid activation function returns a value in the range of 0 to 1 with a smooth transition.

The steepness of the curve and the location of the half-way mark can be adjusted with parameters.

This function is the workhorse of ANN

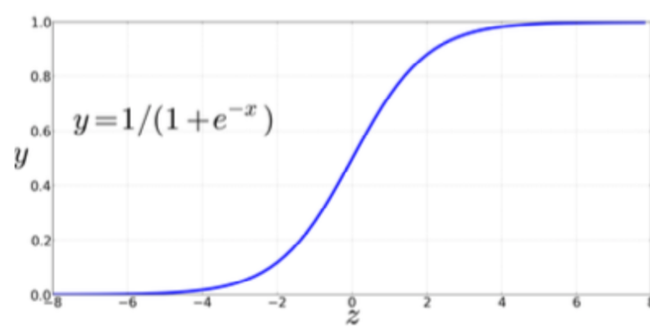


Image source: Jurafsky and Martin: Speech and Language Processing, 3rd Ed., Chpt. 7

tanh Activation Function

The tanh activation function is used in CNNs and RNNs.

It serves to normalize an input value into the range of -1 to 1.

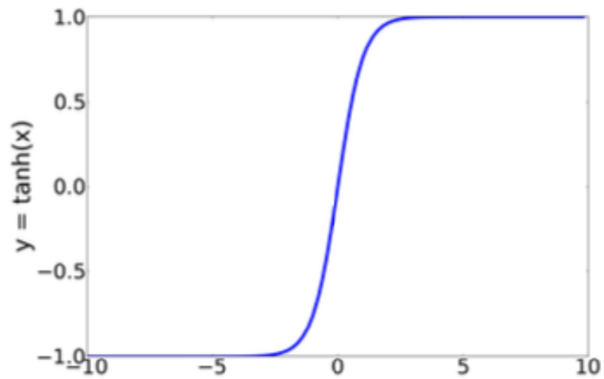


Image source: Jurafsky and Martin: Speech and Language Processing, 3rd Ed., Chpt. 7

ReLU Activation Function

The ReLU activation function converts negative inputs to 0 but keeps the positive inputs unmodified.

This function is used in CNNs, among others to add some non-linearity.

Non-linearity helps with training NN.

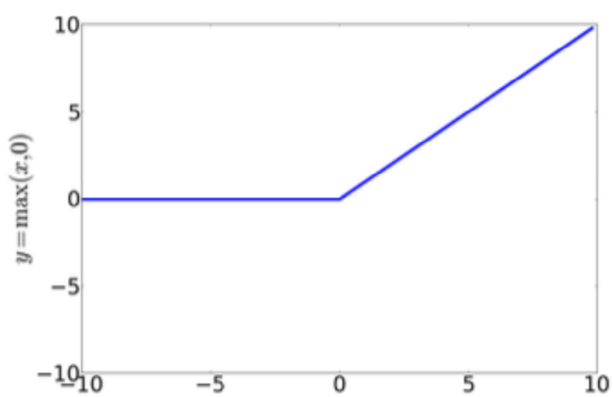


Image source: Jurafsky and Martin: Speech and Language Processing, 3rd Ed., Chpt. 7

Computational Power of NN

One of the first things researchers were concerned about after developing NN is whether NNs can compute anything a digital computer can compute.

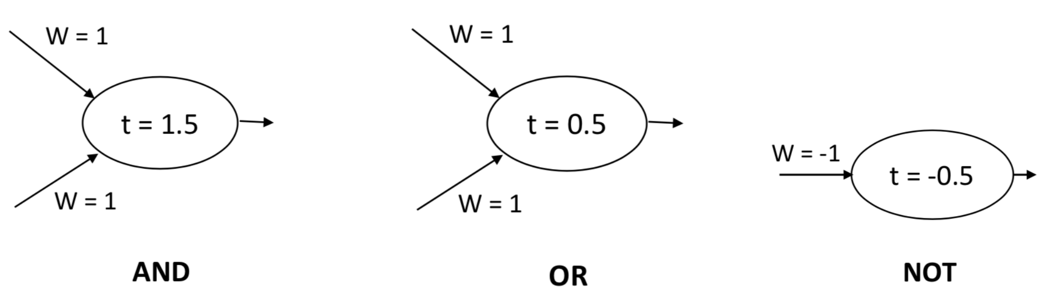
Researchers were wondering whether NN can compute the Boolean functions *and*, *or* and *not*.

As you know from your hardware courses, we can build a computer from NAND or NOR gates.

As such, if a NN can implement *and*, *or* and *not*, we could simulate a digital computer through a NN, just don't try this at home.

Computational Power of NN

Below are NN units that, using step activation functions compute the Boolean functions *and*, *or* and *not*.



Perceptron

Perceptrons were one of the first NN architectures studied.

They are single layer NNs.

They consist of an input layer and an output layer

The input layer is simply a vector of inputs

The output layer consists of the NN units which calculate an output.

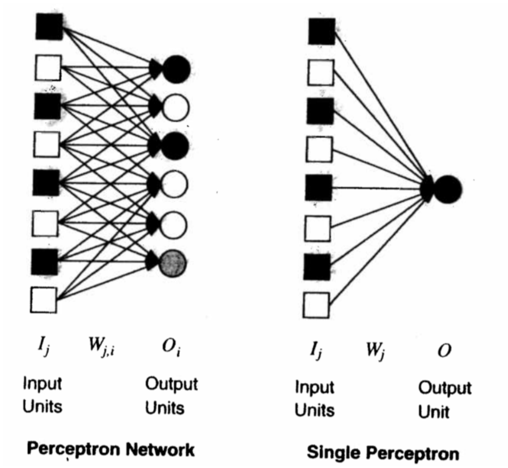


Figure source: Russell and Norvig: AIMA, 1st Ed. Figure 19.8

Knowledge in Perceptrons

As you can tell from the architecture of a perceptron, the only items that can change are the weights.

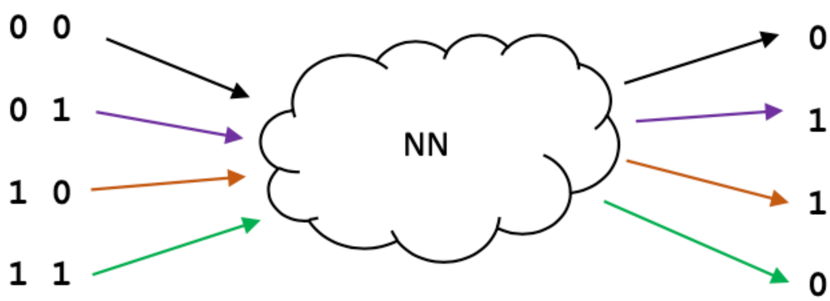
In essence, the weight matrix encodes the "knowledge" of a NN.

Learning in NNs is expensive.

In case you are wondering, learning in humans is expensive too.

In NNs, we use training data to repeatedly adjust the weights until some stopping criterion is satisfied.

Pattern Recognition in NN: XOR



Perceptron Learning

A perceptron is considered a single-layer feed-forward network.

The learning algorithm needs input vectors of data and desired output values for each output.

Let's assume a perceptron, in other words a network with just one output node.

Let n be the number of input units.

Let x = $x_1, \ldots, x_n$ be a set of input values.

Let y be the output

Let W_j, $j = 0 \ldots n$ be the weights

Let g be the activation function.

Perceptron Learning

1. The learning algorithm picks an input *e* from *examples*, the set of input vectors.
2. The algorithms calculates *in*, the input to the single neuron, the output unit. The input is the weighted sum of all the inputs.
3. The algorithm then calculates the *Error.*
   The error is the difference between the desired output *y* associated with *e* and the actual output as computed by the activation function.
   The error can be positive or negative

repeat
  for each *e* in *examples*:
    $in \leftarrow \Sigma_{j=0..n} W_j\, x_j[e]$
    $Error \leftarrow y[e] - g(in)$
    $W_j \leftarrow W_j + \alpha * Error * x_j[e]$
until *done*

Perceptron Learning

4. Next comes the adjustment of the weights.

   This is where learning happens.

   There are three components to the adjustment made to the weights.
   1) α is the learning rate.
      A typical value is 0.05
      It determines how quickly a network converges
   2) Error was calculated in step (3)
      Notice that if the error is negative, then the entire term is negative
      The effect of a negative term is that the weight will decrease
   3) x_j, the value of this particular input.
      If the value is high, then it must be important and there is a larger adjustment to the weight.

repeat
  for each *e* in *examples*:
    $in \leftarrow \Sigma_{j=0..n} W_j\, x_j[e]$
    $Error \leftarrow y[e] - g(in)$
    $W_j \leftarrow W_j + \alpha * Error * x_j[e]$
until *done*

Perceptron Learning

5. Finally, there is the stopping criteria.

   There are basically two options:

   1) Terminate after a set number of iterations.
   2) Terminate when the overall error falls below a certain threshold.

   We will explore both of them

repeat
  for each e in *examples*:
    $in \leftarrow \Sigma_{j=0..n} W_j\, x_j[e]$
    $Error \leftarrow y[e] - g(in)$
    $W_j \leftarrow W_j + \alpha * Error * x_j[e]$
until *done*

Perceptron Learning

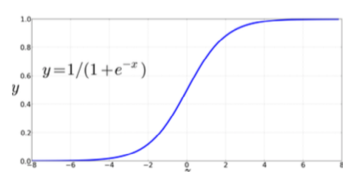
If the network has a differentiable activation function, then we add $g'(in)$ to the to the weight update calculation.

Suppose we have a sigmoid activation function.

When g outputs ½, then the differentiated value is 1.

When g outputs 0 or 1 then the differentiated value is 0.

Using $g'(in)$ modifies the weights so that the output moves away from the center of the range to the edges, i.e. 0 or 1.



repeat
  for each e in *examples*:
    $in \leftarrow \Sigma_{j=0..n} W_j\, x_j[e]$
    $Error \leftarrow y[e] - g(in)$
    $W_j \leftarrow W_j + \alpha * Error * g'(in) * x_j[e]$
until *done*

Linear Separability, AND

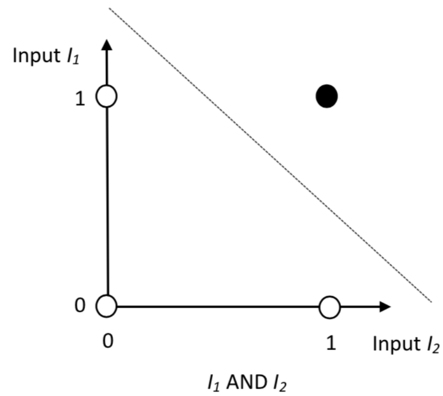
Look at the following graph of AND.

I_1 and I_2 represent the two inputs to AND

The hollow circles represent the value 0 of AND on the inputs of 0 and 1.

The filled in circles represent a value of 1.

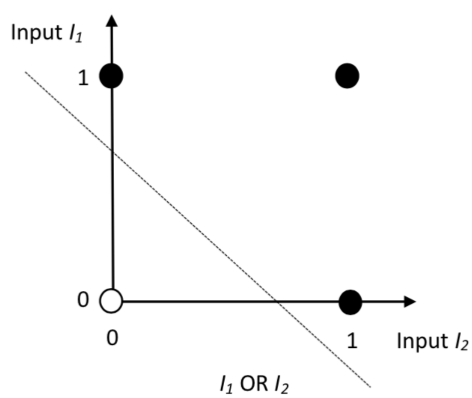
The dashed line represents the fact that we can separate the four different outputs of AND into two separate regions.



Linear Separability, OR

Look at the following graph of OR.

We again see ways of linearly separating the different outputs of OR.



XOR

Look at the following graph for XOR.

There is no way to linearly separate the

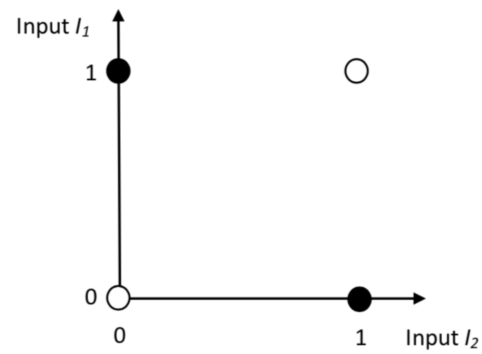
two classes of outputs.

We would need to draw a ridge.

Perceptrons, or single layer feed-forward networks cannot learn the XOR functions.

To learn XOR, we need a feed-forward network with two layers, and output and a hidden layer.

We will cover them next.



I_1 XOR I_2