Advanced Institute for Artificial Intelligence

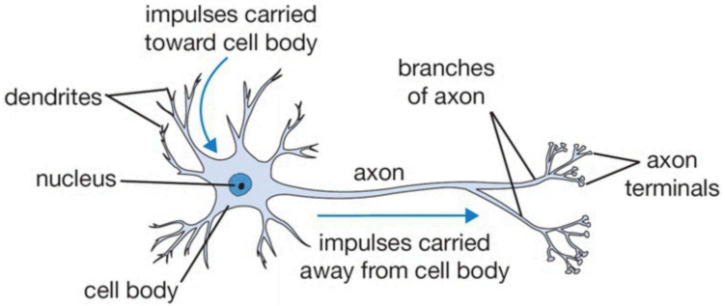
Introduction to Artificial Neural Networks

Raphael Cóbe

raphael.cobe@advancedinstitute.ai

Introduction

Neural Networks were widely used in the 1980s and 1990s aiming to mimic the functioning of the human brain. Their popularity declined in the late 1990s but came back into the spotlight with new approaches based on deep learning. But how do they work? Let's take a look first at the structure of a neuron.


impulses carried toward cell body
dendrites
branches of axon
nucleus
axon
axon terminals
impulses carried away from cell body
cell body

Neural Networks

- Neurons as structural constituents of the brain [Ramón y Cajál, 1911];
- Five to six orders of magnitude *slower than silicon logic gates*;
- In a silicon chip happen in the *nanosecond (on chip)* vs *millisecond range (neural events)*;
- A truly staggering number of neurons (nerve cells) with *massive interconnections between them*;

Neural Networks

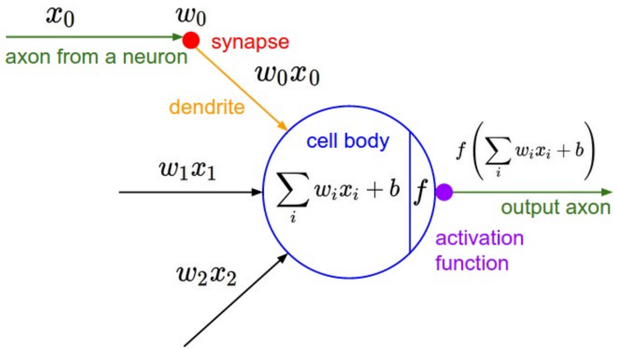
- Receive input from other units and decides whether or not to fire;
- Approximately *10 billion neurons* in the human cortex, and *60 trillion synapses* or connections [Shepherd and Koch, 1990];
- Energy efficiency of the brain is approximately 10^{-16} joules per operation per second against $\approx 10^{-8}$ in a computer;

Neurons

How do they work?

- Control the influence from one neuron on another:
  - *Excitatory* when weight is positive; or
  - *Inhibitory* when weight is negative;
- Nucleus is responsible for summing the incoming signals;
- **If the sum is above some threshold, then *fire!***

Mathematically speaking, we can represent a neuron as follows (McCulloch-Pitts model):



Artificial Neural Networks

- Model each part of the neuron and interactions;
- *Interact multiplicatively* (e.g. w_0x_0) with the dendrites of the other neuron based on the synaptic strength at that synapse (e.g. w_0);
- Learn *synapses strengths*;

Artificial Neural Networks

Function Approximation Machines

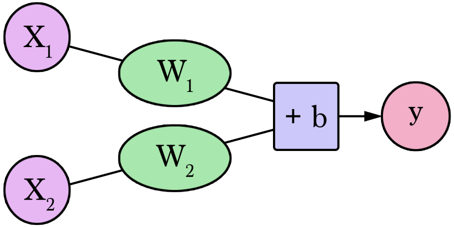
- Datasets as composite functions: $y = f^*(x)$
  - Maps x input to a category (or a value) y;
- Learn synapses weights and approximate y with $\hat{y}$:
  - $\hat{y} = f(x; w)$
  - Learn the w parameters;

Artificial Neural Networks

- Can be seen as a directed graph with units (or neurons) situated at the vertices;
    - Some are *input units*;
- Receive signal from the outside world;
- The remaining are named *computation units*;
- Each unit *produces an output*
    - Transmitted to other units along the arcs of the directed graph;

Artificial Neural Networks

- *Input*, *Output*, and *Hidden* layers;
- Hidden as in ”not defined by the output”;



Artificial Neural Networks

Motivation Example (taken from Jay Alammar blog post)

- Imagine that you want to forecast the price of houses at your neighborhood;
    - After some research you found that 3 people sold houses for the following values:

Area (sq ft) (x)	Price (y)
2,104	$\$399,900$
1,600	$\$329,900$
2,400	$\$369,000$

Artificial Neural Networks

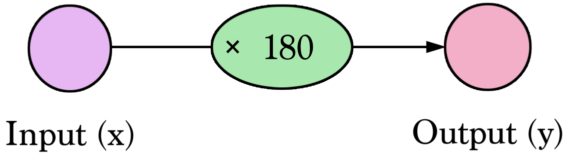
Motivation Example (taken from Jay Alammar blog post)

- If you want to sell a 2K sq ft house, how much should ask for it?
- How about finding the *average price per square feet*?
- $\$180$ *per sq ft.*

Artificial Neural Networks

Motivation Example (taken from Jay Alammar blog post)

- Our very first neural network looks like this:



Artificial Neural Networks

Motivation Example (taken from Jay Alammar blog post)

- Multiplying $2,000$ sq ft by 180 gives us $\$360,000$.
- Calculating the prediction is simple multiplication.
- ***We needed to think about the weight we'll be multiplying by.***
- That is what training means!

Area (sq ft) (x)	Price (y)	Estimated Price($\hat{y}$)
2,104	$\$399,900$	$\$378,720$
1,600	$\$329,900$	$\$288,000$
2,400	$\$369,000$	$\$432,000$

Artificial Neural Networks

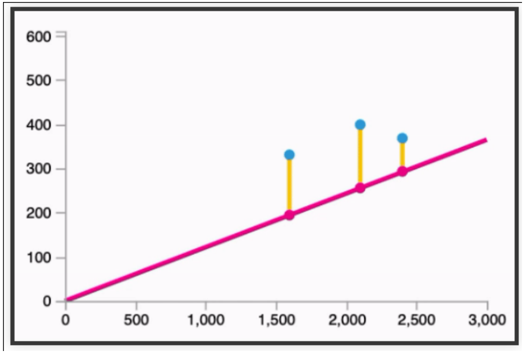
Motivation Example (taken from Jay Alammar blog post)

- How bad is our model?
  - Calculate the *Error*;
  - A better model is one that has less error;
- *Mean Square Error*: $2,058$

Area (sq ft) (x)	Price (y)	Estimated Price($\hat{y}$)	$y - \hat{y}$	$(y - \hat{y})^2$
2,104	$\$399,900$	$\$378,720$	$\$21$	449
1,600	$\$329,900$	$\$288,000$	$\$42$	1756
2,400	$\$369,000$	$\$432,000$	$\$-63$	3969

Artificial Neural Networks

- Fitting the line to our data:



Follows the equation: $\hat{y} = W * x$

Artificial Neural Networks

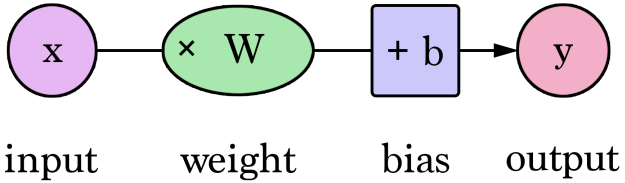
The Bias

How about adding the *Intercept*?

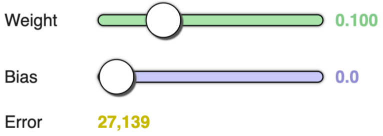
- $\hat{y} = Wx + b$

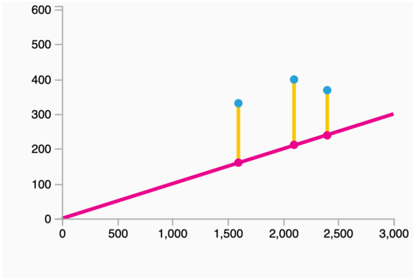
Artificial Neural Networks

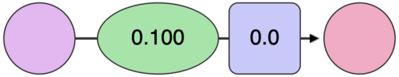
The Bias



The *ad-hoc* training







Artificial Neural Networks

How to discover the correct weights?

- Gradient Descent:
    - Finding the *minimum of a function*;
    - Look for the best weights values, *minimizing the error*;
    - Takes steps *proportional to the negative of the gradient* of the function at the current point.
    - Gradient is a vector that is *tangent of a function* and points in the direction of greatest increase of this function.

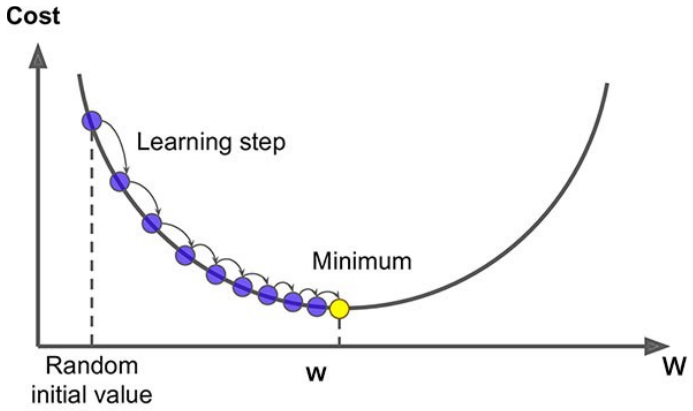
Artificial Neural Networks

Gradient Descent

- In mathematics, gradient is defined as *partial derivative for every input variable* of function;
- *Negative gradient* is a vector pointing at the *greatest decrease* of a function;
- *Minimize a function* by iteratively moving a little bit in the direction of negative gradient;

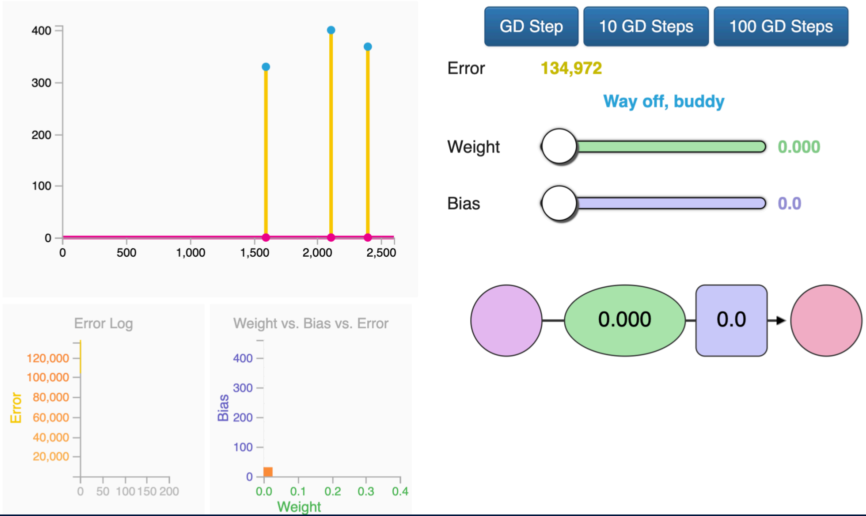
Artificial Neural Networks

Gradient Descent



Artificial Neural Networks

Gradient Descent



Error 134,972

Way off, buddy

Weight 0.000

Bias 0.0

Perceptron

The Perceptron was formally proposed by McCulloch and Pitts in the 1940s with the purpose of mathematically modeling the human neuron. Although it served as the basis for many algorithms, its discriminative power is limited, as it can only learn hyperplanes as decision functions.

Problem definition: let $\mathcal{X} = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \ldots, (\boldsymbol{x}_z, y_z)\}$ be a dataset where $\boldsymbol{x}_i \in \mathbb{R}^{n+1}$ corresponds to the input data, and $y_i \in [-1, +1]$ denotes its respective output value. Also, $\mathcal{X}$ can be **partitioned** as follows: $\mathcal{X} = \mathcal{X}^1 \cup \mathcal{X}^2$, where $\mathcal{X}^1$ and $\mathcal{X}^2$ denote the **training** and **test** data sets, respectively. Our goal is, given the training set, to learn a function $h : \mathbb{R}^{n+1} \rightarrow \{-1, +1\}$ that can correctly assign a class to a given sample.

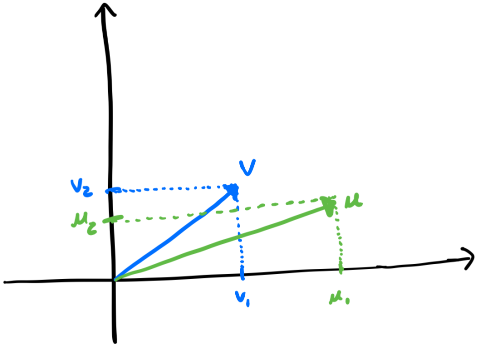
Now, let's adapt the threshold activation function so that we can create our hypothesis function:

$$h_{\boldsymbol{w}}(\boldsymbol{x}) = \begin{cases} +1 \text{ if } \boldsymbol{w}^T\boldsymbol{x} \geq \theta \\ -1 \text{ otherwise.} \end{cases} \tag{1}$$

To simplify the notation, it's usual to bring θ to the left side of the equation and assign $w_0 = \theta$. Again, we'll consider $x^0 = 1$. Thus, we have the updated hypothesis function as follows:

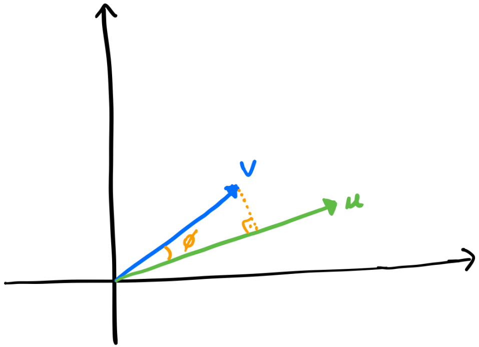
$$h_{\boldsymbol{w}}(\boldsymbol{x}) = \begin{cases} +1 \text{ if } \boldsymbol{w}^T\boldsymbol{x} \geq 0 \\ -1 \text{ otherwise.} \end{cases} \tag{2}$$

Let's revisit some concepts of Analytical Geometry. Suppose we have two vectors $\boldsymbol{u} = [u_1 \ u_2]$ and $\boldsymbol{v} = [v_1 \ v_2]$. Geometrically, we can represent them as follows:



Recalling that $\|u\| = \sqrt{u_1^2 + u_2^2}$ denotes the length (magnitude) of u.

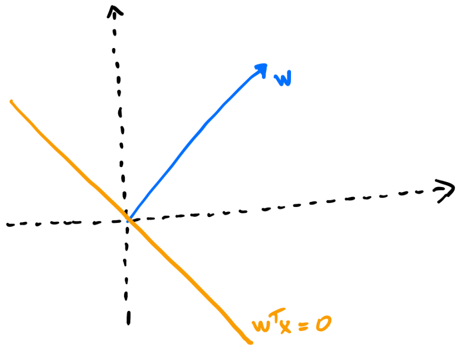
We also have the definition of **scalar projection** between two vectors, given as follows:



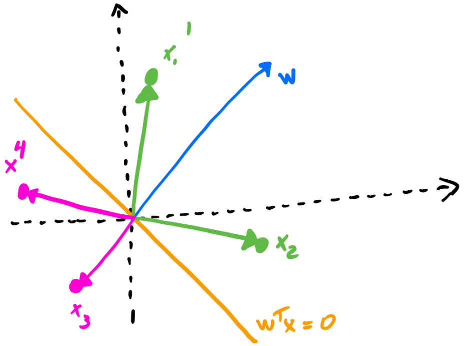
So, we can represent the scalar projection between two vectors as $\boldsymbol{w}^T\boldsymbol{x} = \|\boldsymbol{w}\|.\cos\phi$. Remembering that we have the following situations:

- $\cos\phi > 0$ when $0 < \phi < 90^\circ$.
- $\cos\phi < 0$ when $90^\circ < \phi < 270^\circ$.
- $\cos\phi = 0$ when $\phi = 90^\circ$ or $\phi = 270^\circ$

Going back to our initial problem, we have that the equation $\boldsymbol{w}^T\boldsymbol{x} = 0$ defines a hyperplane orthogonal to the weight vector $\boldsymbol{w}$ shifted by $-\theta$ (assuming $w_0 = -\theta$ and $x_0 = 1$). Let's assume $\theta = 0$, meaning the hyperplane passes through the origin.



How do we learn the weight set $\boldsymbol{w}$? The intuitive idea is to **adjust** the weight vector in a way that the samples are correctly positioned in the feature space. In the example below, we have a dataset $\mathcal{X}^1 = \{(\boldsymbol{x}_1, +1), (\boldsymbol{x}_2, +1), (\boldsymbol{x}_3, -1), (\boldsymbol{x}_4, -1)\}$, where the hyperplane $\boldsymbol{w}^T\boldsymbol{x} = 0$ is already correctly positioned.



The weight updating rule is given by the following formula:

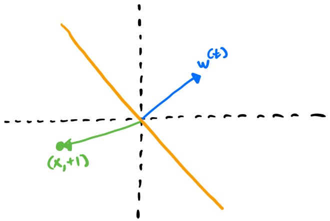
$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \alpha(y_i - h_{\boldsymbol{w}^{(t)}}(\boldsymbol{x}_i))\boldsymbol{x}_i, \qquad (3)$$

where $\forall i = 1, 2, \ldots, m$.

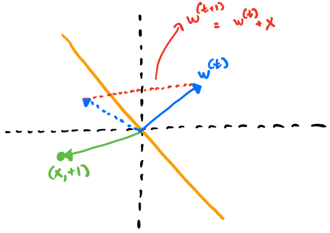
But how does it work in practice? Suppose a sample $\boldsymbol{x} \in \mathcal{X}$ such that $y = +1$ and $h_{\boldsymbol{w}}(\boldsymbol{x}) = -1$. For a value of $\alpha = 0.5$, we have:

$$\begin{aligned} \boldsymbol{w}^{(t+1)} &= \boldsymbol{w}^{(t)} + \alpha(1 - (-1))\boldsymbol{x} \\ &= \boldsymbol{w}^{(t)} + 0.5(2)\boldsymbol{x} = \boldsymbol{w}^{(t)} + \boldsymbol{x}. \end{aligned} \tag{4}$$

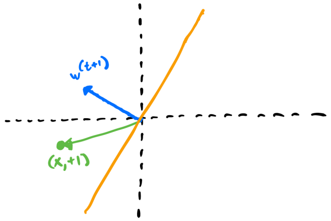
Thus, the weight vector $\boldsymbol{w}$ will be rotated so that $\boldsymbol{x}$ is positive.



Prediction



Update



Learned Hyperplane

Similarly, if the label of $\boldsymbol{x}$ is negative, i.e., $y = -1$, we have:

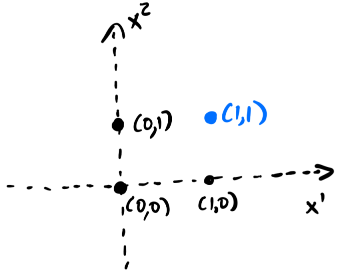
$$\begin{aligned} \boldsymbol{w}^{(t+1)} &= \boldsymbol{w}^{(t)} + \alpha(-1 - (+1))\boldsymbol{x} \\ &= \boldsymbol{w}^{(t)} + 0.5(-2)\boldsymbol{x} = \boldsymbol{w}^{(t)} - \boldsymbol{x}. \end{aligned} \tag{5}$$

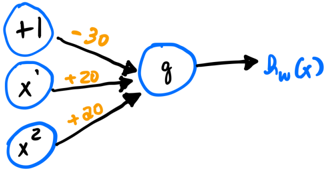
Thus, the weight vector will be rotated (through projections) to the other side. For a dataset that is **linearly separable**, it has been mathematically proven that the Perceptron algorithm has **guaranteed convergence**.

How does its algorithm work? Let's see:

1. Assign random weights to $\boldsymbol{w}$.
2. Initialize α.
3. $t = 0$.
4. For each sample $\boldsymbol{x}_i \in \mathcal{X}^1$, do:
    1. $\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \alpha(y_i - h_{\boldsymbol{w}^{(t)}}(\boldsymbol{x}_i))\boldsymbol{x}_i$
5. Repeat step 4 until some convergence criterion is established.

Let's see some examples of the Perceptron's functioning. Consider solving the logical equation: $y = x^1$ AND x^2. For two inputs, we have 2^2 possibilities of samples, i.e., our dataset is composed of the following elements: $\mathcal{X} = \{([0\ 0], 0), ([0\ 1], 0), ([1\ 0], 0), ([1\ 1], 1)\}$. Can we find the separating hyperplane?

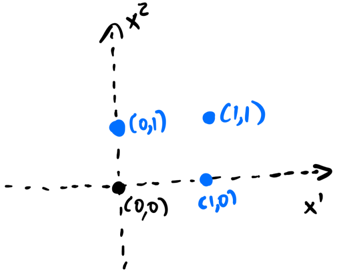


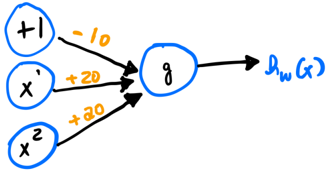


Our hypothesis function is given by $h_{\boldsymbol{w}}(\boldsymbol{x}) = g(-30 + 20x^1 + 20x^2)$. Using g as a logistic function, we have:

x^1	x^2	$h_{\boldsymbol{w}}(\boldsymbol{x})$
0	0	$g(-30) \approx 0$
0	1	$g(-10) \approx 0$
1	0	$g(-10) \approx 0$
1	1	$g(10) \approx 1$

Consider, now, the problem of solving the following logical equation: $y = x^1$ OR x^2. For two inputs, we have 2^2 possibilities of samples, i.e., our dataset is composed of the following elements: $\mathcal{X} = \{([0\ 0], 0), ([0\ 1], 1), ([1\ 0], 1), ([1\ 1], 1)\}$. Can we find the separating hyperplane?

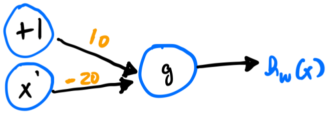




Our hypothesis function is given by $h_{\boldsymbol{w}}(\boldsymbol{x}) = g(-10 + 20x^1 + 20x^2)$. Using g as a logistic function, we have:

x^1	x^2	$h_{\boldsymbol{w}}(\boldsymbol{x})$
0	0	$g(-10) \approx 0$
0	1	$g(10) \approx 1$
1	0	$g(10) \approx 1$
1	1	$g(30) \approx 1$

Consider, now, the problem of solving the following logical equation: $y = \text{NOT } x^1$. Now, we have only one input, that is, x^1. Thus, our dataset is composed of the following elements: $\mathcal{X} = \{(0,1),(1,0)\}$. Can we find the separating hyperplane?

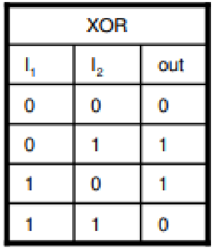


Our hypothesis function is given by $h_{\boldsymbol{w}}(\boldsymbol{x}) = g(10 - 20x^1)$. Using g as a logistic function, we have:

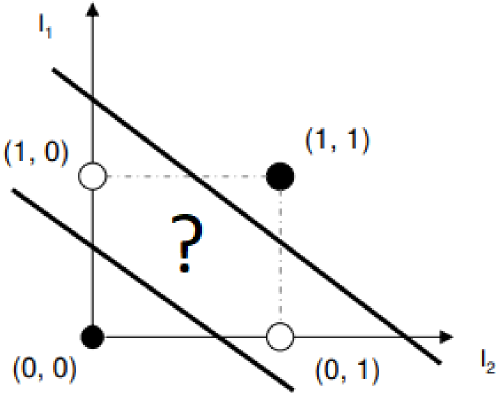
x^1	$h_{\boldsymbol{w}}(\boldsymbol{x})$
0	$g(10) \approx 1$
1	$g(-10) \approx 0$

Perceptron - What it *can't do*!

- The *XOR* function:

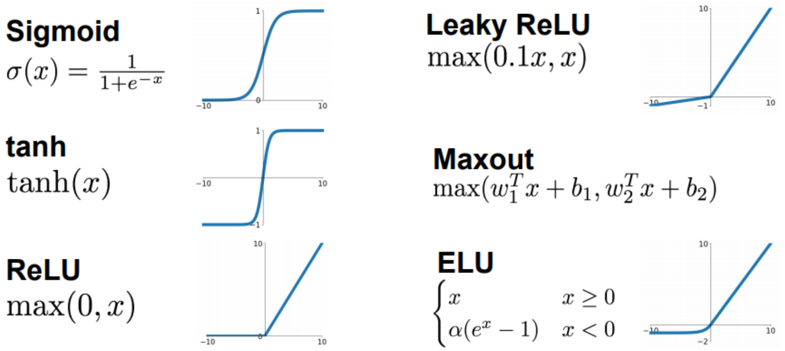


XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0



Activation Functions

- *Multiply the input* by its *weights*, *add the bias*, and *apply activation*;
- Sigmoid, Hyperbolic Tangent, Rectified Linear Unit;
- *Differentiable function* instead of the step function;



Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$

tanh
$\tanh(x)$

ReLU
$\max(0, x)$

Leaky ReLU
$\max(0.1x, x)$

Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

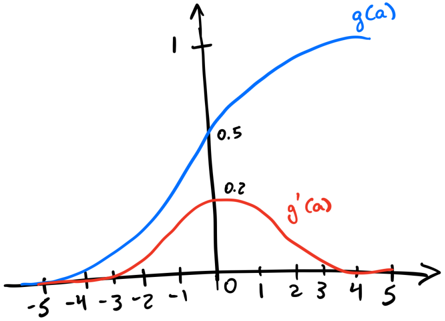
ELU
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

Adding power to the ANN

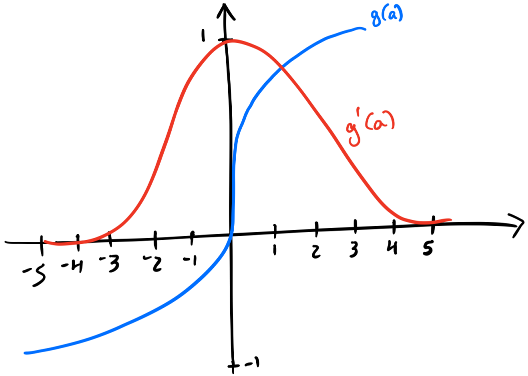
Some examples of activation functions:

- Logistic function (sigmoid): $g(a) = \dfrac{1}{1+e^{-a}}$, such that $g(a) \in [0, 1]$.
- Threshold function (step):
$$g_\theta(a) = \begin{cases} 1 \text{ if } \boldsymbol{w}^T\boldsymbol{x} \geq \theta \\ 0 \text{ otherwise.} \end{cases}$$
  such that $g(a) \in \{0, 1\}$.
- Hyperbolic tangent function: $g(a) = 2\sigma(2a) - 1$, such that $g(a) \in [-1, 1]$ and $\sigma(x)$ corresponds to the logistic function.

It's desirable for an activation function to be **differentiable**. How to choose it? It depends on its application.



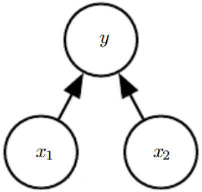
Suppose $g(a) = \dfrac{1}{1+e^{-a}}$. We have $g'(a) = g(a)(1-g(a))$. Note that $g'(a)$ saturates when $a > 5$ or $a < -5$. Furthermore, $g'(a) < 1, \forall a$. This means that for networks with many layers, the gradient tends to **vanish** during training.

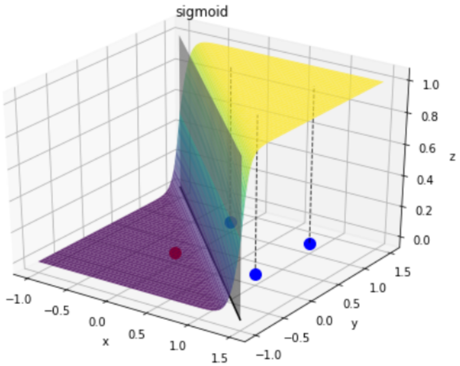


Suppose $g(a) = 2\sigma(2a) - 1$. We have $g'(a) = 1 - g^2(a)$. Although saturations occur, $g'(a)$ reaches higher values, even reaching the maximum of 1 when $a = 0$.

Perceptron - Solving the XOR problem

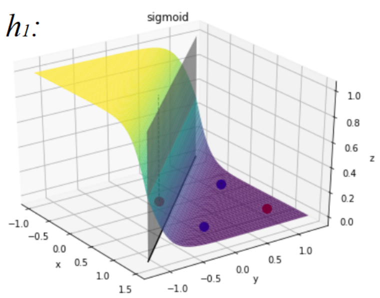
- 3D example of the solution of learning the OR function:
    - Using *Sigmoid* function;

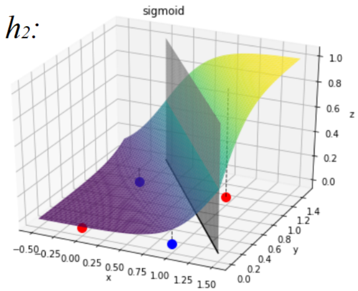




Perceptron - Solving the XOR problem

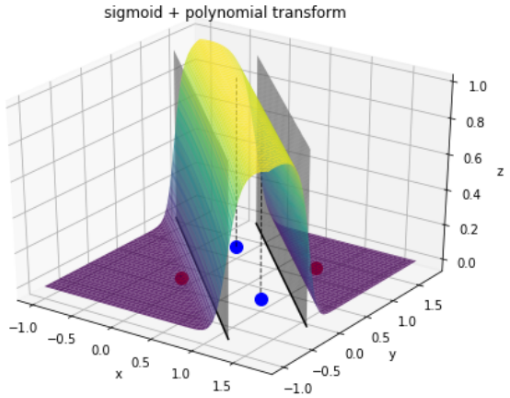
- Maybe there is a combination of functions that could create hyperplanes that separate the *XOR* classes:
  - By increasing the number of layers we increase the complexity of the function represented by the ANN:





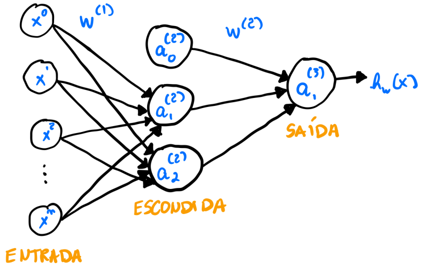
Perceptron - Solving the XOR problem

- The combination of the layers:



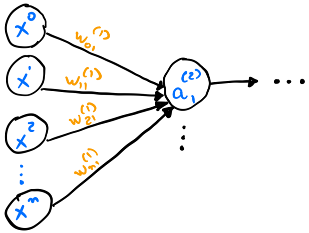
Multilayer Perceptron

So, what would be a Multilayer Perceptron (MLP) Neural Network? Basically, it's a group of neurons that, when combined, allow learning a greater number of decision functions.



In the illustration above, $a_i^{(j)}$ denotes neuron i from layer j, and $\boldsymbol{W}^{(l)}$ is the weight matrix connecting layers l and $l+1$. This architecture is generally represented as $n{:}3{:}1$.

Considering the previous neural network, let's analyze a more specific situation.

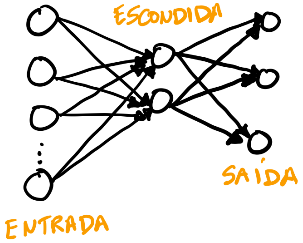


$$a_1^{(2)} = g(\underbrace{w_{01}^{(1)}x^0 + w_{11}^{(1)}x^1 + w_{02}^{(1)}x^2 + \ldots + w_{0n}^{(1)}x^n}_{\sum_{i=0}^{n} w_{i1}^{(1)}x^i = b_1^{(2)}})$$

The final decision function of the neural network is given by the following formulation:

$$h_{\boldsymbol{x}}(\boldsymbol{w}) = a_1^{(3)} = g\left(w_{01}^{(2)}a_0^{(2)} + w_{11}^{(2)}a_1^{(2)} + w_{21}^{(2)}a_2^{(2)}\right). \tag{6}$$

And in the case of problems with multiple classes? In this case, for a problem with c classes, our output layer needs c neurons.



We can use the *one-hot* encoding methodology to represent each output neuron, where $\boldsymbol{h}_{\boldsymbol{w}}(\boldsymbol{x}) \in \mathbb{R}^3$:

$$h^1_{\boldsymbol{w}}(\boldsymbol{x}) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad h^2_{\boldsymbol{w}}(\boldsymbol{x}) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad h^3_{\boldsymbol{w}}(\boldsymbol{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Class 1 Class 2 Class 3

The same happens with the label y of each sample, which now becomes a vector $\boldsymbol{y} \in \mathbb{R}^3$.

There are several training algorithms for MLP Neural Networks, where the most well-known is called **backpropagation**. It has two steps: (i) forward propagation and (ii) backward propagation. Before studying its operation, let's take a look at the cost function:

$$J(\boldsymbol{w}) = \frac{1}{m}\left[\sum_{i=1}^{m}\sum_{k=1}^{c} -y_i^k \log\Big(h_{\boldsymbol{w}}^k(\boldsymbol{x}_i)\Big) - (1-y_i^k)\log\Big(1-h_{\boldsymbol{w}}^k(\boldsymbol{x}_i)\Big)\right]. \tag{7}$$

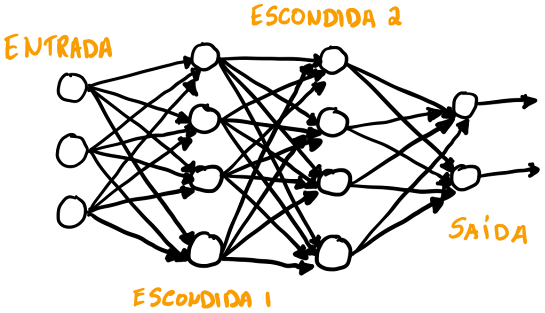
By analogy, the above formulation encompasses a neural network with c logistic regressors if we have a logistic activation function in the output layer.

We can use, once again, gradient descent to optimize the cost function $J(\boldsymbol{w})$. However, note that the problem becomes more complex because we need to calculate the partial derivatives with respect to all the weights of the network, i.e.,

$$\frac{\partial J(\boldsymbol{w})}{\partial w_{ij}^{(l)}}, \tag{8}$$

where $l = 1, 2, \ldots, L-1$ such that L denotes the number of layers in the neural network.

Let's consider a network of type $3{:}4{:}4{:}2$ illustrated below. Suppose we have only one sample in the training set. In this case, the *forward* step is given by the following stages:



1. $\boldsymbol{a}^{(1)} \leftarrow \boldsymbol{x}$
2. $\boldsymbol{b}^{(2)} \leftarrow \left(\boldsymbol{W}^{(1)}\right)^T \boldsymbol{a}^{(1)}$
3. $\boldsymbol{a}^{(2)} \leftarrow g\left(\boldsymbol{b}^{(2)}\right)$
4. $\boldsymbol{b}^{(3)} \leftarrow \left(\boldsymbol{W}^{(2)}\right)^T \boldsymbol{a}^{(2)}$
5. $\boldsymbol{a}^{(3)} \leftarrow g\left(\boldsymbol{b}^{(3)}\right)$
6. $\boldsymbol{b}^{(4)} \leftarrow \left(\boldsymbol{W}^{(3)}\right)^T \boldsymbol{a}^{(3)}$
7. $h_{\boldsymbol{w}}(\boldsymbol{x}) = \boldsymbol{a}^{(4)} \leftarrow g\left(\boldsymbol{b}^{(4)}\right)$

For the *backpropagation* step, we have the definition of a new variable $\delta_j^{(l)}$ which denotes a partial error accumulated in neuron j of layer l. This error must be calculated differently for the output neurons and hidden layers, as follows:

- Output layer ($l = 4$):

$$\begin{aligned} \delta_j^{(4)} &= a_j^{(4)} - y^j \\ &= h_{\boldsymbol{w}}^j(\boldsymbol{x}) - y^j \end{aligned} \tag{9}$$

  In vector notation, we have $\boldsymbol{\delta}^{(4)} = \boldsymbol{a}^{(4)} - \boldsymbol{y} = \boldsymbol{h}_{\boldsymbol{w}}(\boldsymbol{x}) - \boldsymbol{y}$.

- Hidden layers $l = \{2, 3\}$:
  - $\boldsymbol{\delta}^{(3)} = \left(\boldsymbol{W}^{(3)}\right)^T \boldsymbol{\delta}^{(4)} .* g'\left(\boldsymbol{b}^{(3)}\right)$
  - $\boldsymbol{\delta}^{(2)} = \left(\boldsymbol{W}^{(2)}\right)^T \boldsymbol{\delta}^{(3)} .* g'\left(\boldsymbol{b}^{(2)}\right)$

  In practice, we have the following formulation for error propagation in the intermediate layers:

$$\boldsymbol{\delta}^{(l)} = \left(\boldsymbol{W}^{(l)}\right)^T \boldsymbol{\delta}^{(l+1)} .* g'\left(\boldsymbol{b}^{(l)}\right). \tag{10}$$

The name **backpropagation** comes from the fact that the algorithm ”propagates backward” the estimated error in each layer. The partial derivatives can be calculated as follows:

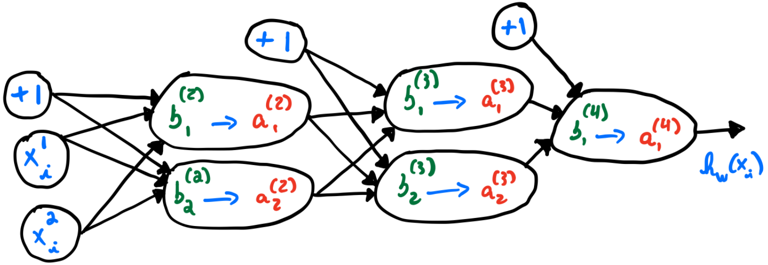
$$\frac{\partial J(\boldsymbol{w})}{\partial w_{ij}^{(l)}} = a_i^{(l)} \delta_j^{(l+1)}. \tag{11}$$

Note, then, that the partial derivatives are calculated with respect to all the weights of the neural network.

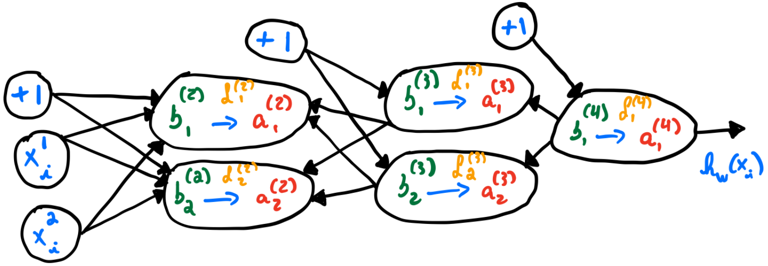
Here is the backpropagation algorithm for training an MLP neural network.

1. Assign random weights to $w^{(l)}ij$ for $\forall l, i, j$
2. Execute the steps below until the stopping criterion has been established: Epoch loop
    1. $\Delta^{(l)}ij = 0 \ \ \forall l, i, j$ Variable used to store $\frac{\partial J(\boldsymbol{w})}{\partial w^{(l)}ij}$
    2. For each sample $\boldsymbol{x}i \in \mathcal{X}^1$, do:
        1. Execute the *forward propagation* step to calculate $\boldsymbol{a}^l, \ \ l = 2, 3, \dots, L$
        2. Execute the *backward propagation* step to calculate:
        3. $\boldsymbol{\delta}^l, \ \ l = L, L-1, \dots, 2$ Error in each neuron
        4. $\Delta^{(l)}ij = \Delta^{(l)}ij + a^{(l)}i\delta^{(l+1)}j$ Partial derivatives are accumulated
    3. $D^{(l)}ij = \frac{1}{m}\Delta^{(l)}ij \ \ \forall l, i, j$ Calculate the average gradient
    4. $w^{(l)}ij = w^{(l)}ij - \alpha D^{(l)}_{ij} \ \ \forall l, i, j$ Update weights with gradient descent
    5. Evaluate the cost function $J(\boldsymbol{w})$

Summarizing: *forward* step.

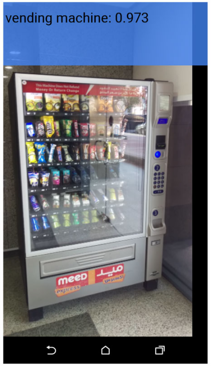

+1
x_i^1
x_i^2
+1
$b_1^{(2)} \rightarrow a_1^{(2)}$
$b_2^{(2)} \rightarrow a_2^{(2)}$
+1
$b_1^{(3)} \rightarrow a_1^{(3)}$
$b_2^{(3)} \rightarrow a_2^{(3)}$
$b_1^{(4)} \rightarrow a_1^{(4)}$
$h_w(x_i)$

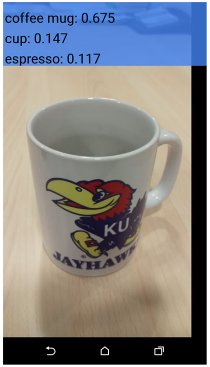
Summarizing: *backward* step.


+1
x_i^1
x_i^2
+1
+1
$b_1^{(2)}$ $d_1^{(2)}$ $\rightarrow a_1^{(2)}$
$b_2^{(2)}$ $d_2^{(2)}$ $\rightarrow a_2^{(2)}$
$b_1^{(3)}$ $d_1^{(3)}$ $\rightarrow a_1^{(3)}$
$b_2^{(3)}$ $d_2^{(3)}$ $\rightarrow a_2^{(3)}$
$b_1^{(4)}$ $d_1^{(4)}$ $\rightarrow a_1^{(4)}$
$h_w(x_i)$

Predicting probabilities

- Imagine that we have *more than 2 classes* to output;
- One of the *most popular usages* for ANN;





Predicting probabilities

- The Softmax function;
- Takes an array and outputs a probability distribution, i.e., *the probability of the input example belonging to each of the classes* in my problem;
- One of the activation functions available at `Pytorch`:

```
    return F.log_softmax(output, dim=1)
```

Note

Softmax - function that takes as input a vector of K real numbers, and normalizes it into a probability distribution

Loss functions

- For regression problems
- Mean squared error is *not always the best one to go*;
- What if we have a three classes problem?
- Alternatives: `mean_absolute_error`, `mean_squared_logarithmic_error`

Note

logarithm means changing scale as the error can grow really fast;

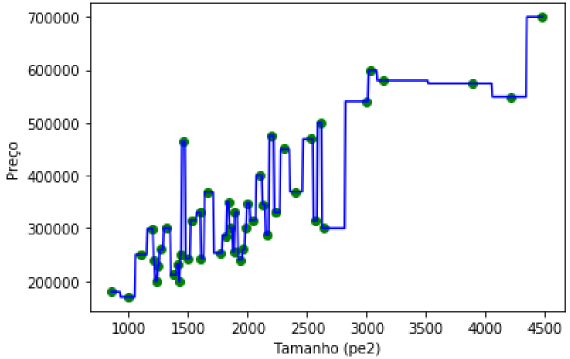
Loss functions

- Cross Entropy loss:
- Default loss function to use for binary classification problems.
- Measures the *performance of a model* whose output is a probability value between 0 and 1;
- *Loss increases* as the *predicted probability diverges* from the actual label;
- A *perfect model* would have a log loss of 0;

Note

As the correct predicted probability decreases, however, the log loss increases rapidly: In case the model has to answer 1, but it does with a very low probability;

Even with relatively simple models, we can have a good fit on the training set



Would that be true?

- Models tend to become too specialized in the training set.
- Models adhering perfectly to the shown examples risk overfitting.
- **Remember:** The training set is not a perfect representation of the real world.
- The main goal is to model a phenomenon through examples; merely classifying the training set is worthless.

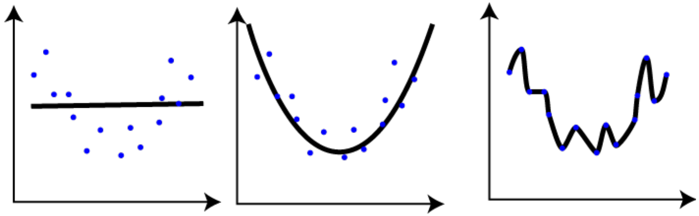


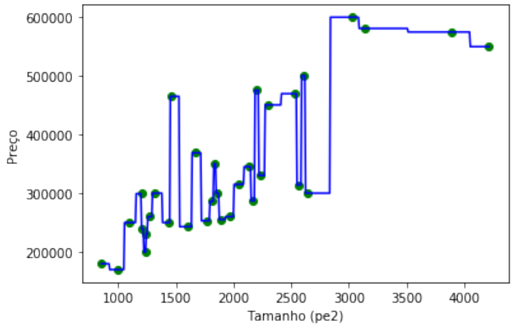
Figure: (Left): Underfit, (Center): Fit, (Right): Overfit

How to know if the model is overfitting?

- Always evaluate models on samples the **model has never seen**.
- Divide data into training and test sets, possibly using cross-validation.

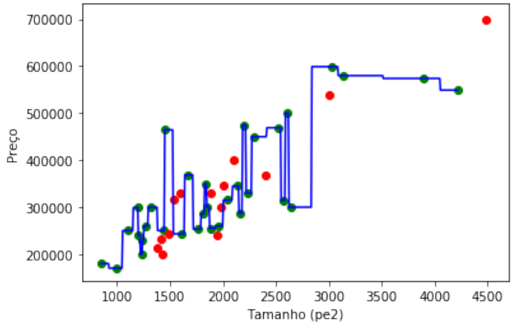
Portland Study: Tree

- Training set: $R^2 = 1.0$



Portland Study: Tree

- Test set: $R^2 = 0.43$



Interpreting the test set

- Perfect metric on the test set means the model is perfect?
- **Not necessarily:** In no non-trivial problem will you have access to a completely representative database of the problem.
- Evaluating with a test set *helps*, but doesn't solve the problem.
- There will never be enough examples to perfectly model the phenomenon.

Model error analysis

- Prediction error can be divided into three parts:
    - Irreducible Error: cannot be eliminated, regardless of the algorithm used.
    - Introduced from the chosen framing of the problem.
    - Caused by unknown factors.
    - Bias Error: assumptions made by a model to make the target function easier to learn.
    - Variance Error: the amount the estimate of the target function will change if different training data is used.

Bias Error

- Difference between the expected (or average) prediction of our model and the correct value we are trying to predict.
- Imagine repeating the entire model-building process more than once:
    - Each time you gather new data and run a new analysis, you create **a new model**.
    - Due to randomness in the underlying data sets, the resulting models will have **a variety of predictions**.
    - Measures **how far, on average, the predictions of these models are from the correct value**.
- Our model has bias if it **systematically predicts below or above the target variable**.

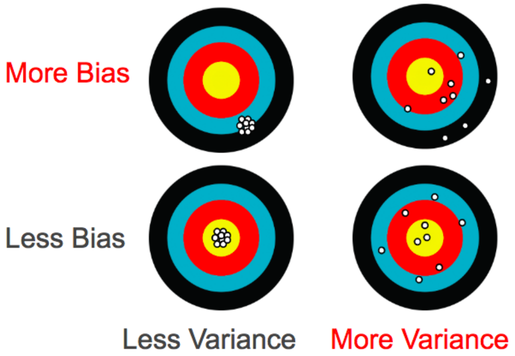
Variance Error

- In a sense, it captures the **model's generalization capability**.
- How much our prediction would change if we trained it with different data.
- Ideally, it shouldn't change much from one training data set to the next.
- Algorithms with high variance are **strongly influenced by the specifications of the training data**.
- Generally, nonlinear machine learning algorithms that have a lot of flexibility have **high variance**.
  - e.g., **Polynomial Regression with high-degree polynomials**!

Dilemma: Variance x Bias

- Low bias: suggests fewer assumptions about the shape of the target function.
  - Regression Trees, KNN Regression.
- High bias: suggests more assumptions about the shape of the target function.
  - Linear Regression, Logistic Regression.
- Low variance: suggests small changes in the estimated target function with changes in the training data.
  - Linear Regression, Logistic Regression.
- High variance: suggests large changes in the estimated target function with changes in the training data.
  - Regression Trees, KNN Regression.

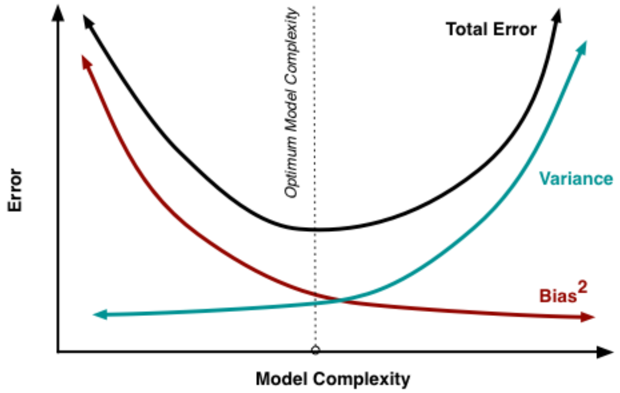
Dilemma: Variance x Bias



- Increasing bias will decrease variance.
- Increasing variance will decrease bias.

Dilemma: Variance x Bias

- **The tradeoff**



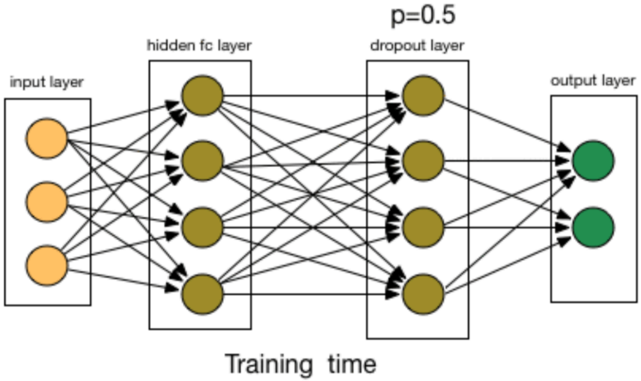
- A very simple model with few parameters has high bias and low variance.
- A complex model with a large number of parameters will have high variance and low bias.
- Seek for balance - good without overfitting or underfitting the data.

- Models should try to **generalize** beyond what is observed in the training set.
- **Regularization** plays a role in controlling classifiers' overfitting.

Artificial Neural Networks

Dealing with overfitting

- *Dropout* layers:
  - Randomly *disable* some of the neurons during the training passes.



Artificial Neural Networks

Dealing with overfitting

- *Dropout* layers:

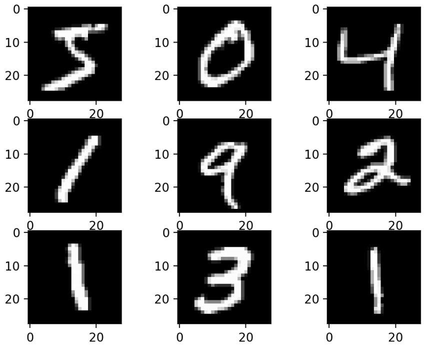
```
# Drop half of the neurons outputs from the previous layer
self.fc1_drop = nn.Dropout(0.5)
```

- **Note:**
    - "drops out" a random set of activations in that layer by setting them to zero.
    - forces the network to be redundant.
    - the net should be able to provide the right classification for a specific example even if some of the activations are dropped out.

Artificial Neural Networks

Larger Example

- The MNIST dataset: database of handwritten digits.
- Dataset included in Pytorch.



Artificial Neural Networks

The MNIST MLP

- Try to improve the classification results using this notebook.
- Things to try:
  - Increase the number of neurons at the first layer.
  - Change the optimizer and the loss function.
  - Try with other optimizers.
  - Try adding some extra layers.

Artificial Neural Networks

The MNIST MLP

- Try to improve the classification results using this notebook.
- Things to try:
  - Try adding `Dropout` layers.
  - Increase the number of `epochs`.
  - Try to *normalize the data*.
- What is the best accuracy?
- My solution: Solution notebook.