Lecture 2: Introduction to Neural Networks

Rafael Martínez-Galarza, Pavlos Protopapas
Harvard-Smithsonian Center for Astrophysics

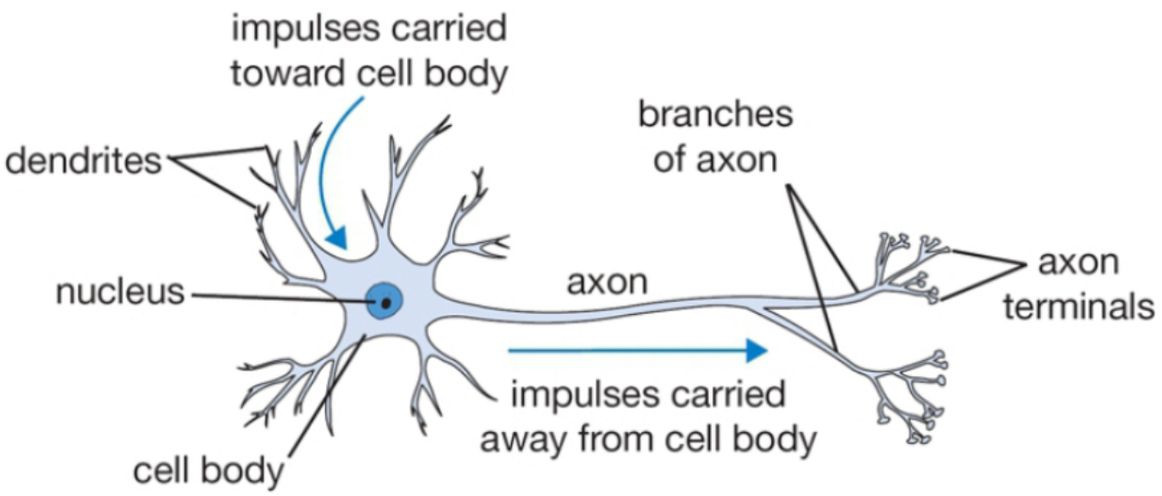
CENTER FOR ASTROPHYSICS
HARVARD & SMITHSONIAN

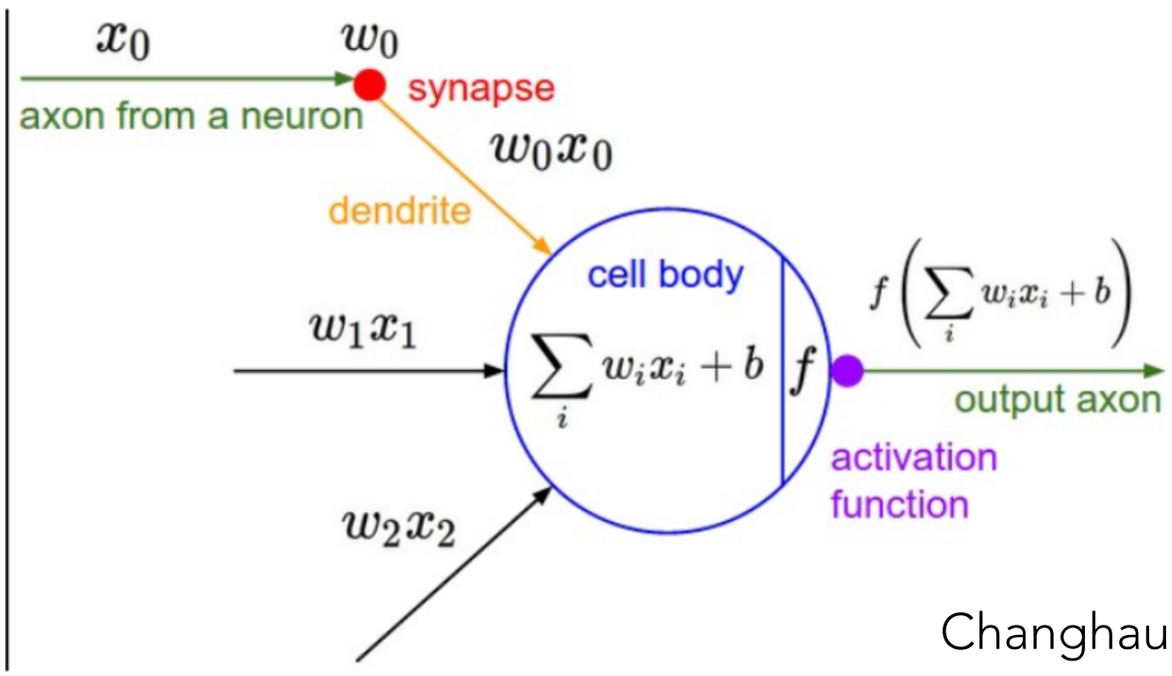
This lecture:

- Introduction to neural networks
- Feedforward networks
- Stochastic gradient descent
- Back propagation algorithm
- Validation and regularization

Feedforward Networks

A SIMPLE PERCEPTRON





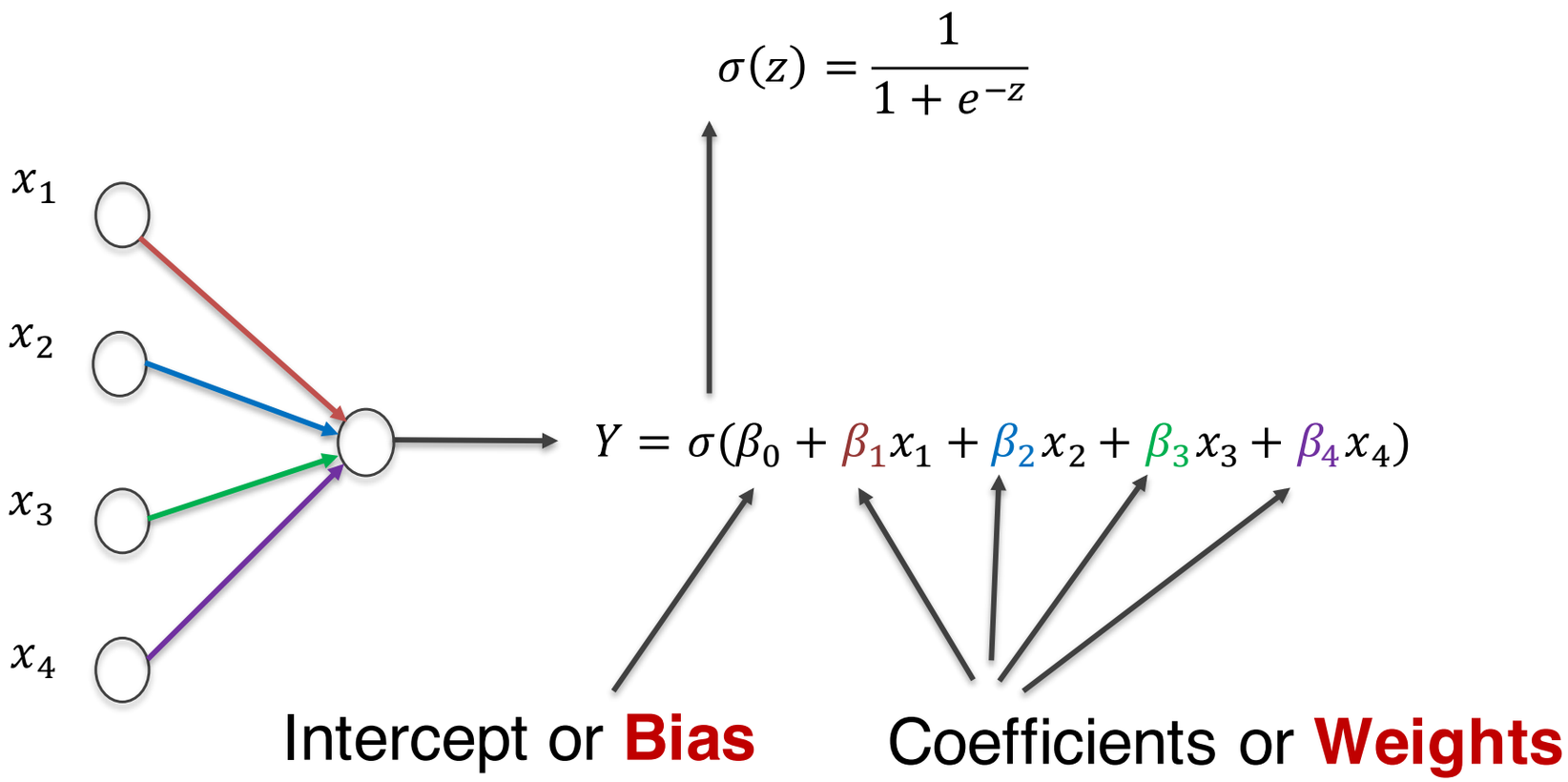
Changhau

- A non-linear activation function is applied to the linear combination of the features.
- The loss (error) function that we want to minimize is minus the logarithm of the joint probability of the labels observed in the training set:

$$J(\theta) = -\sum_i \left(y^{(i)} \log(h_\theta(x^{(i)})) + (1 - y^{(i)}) \log(1 - h_\theta(x^{(i)})) \right)$$

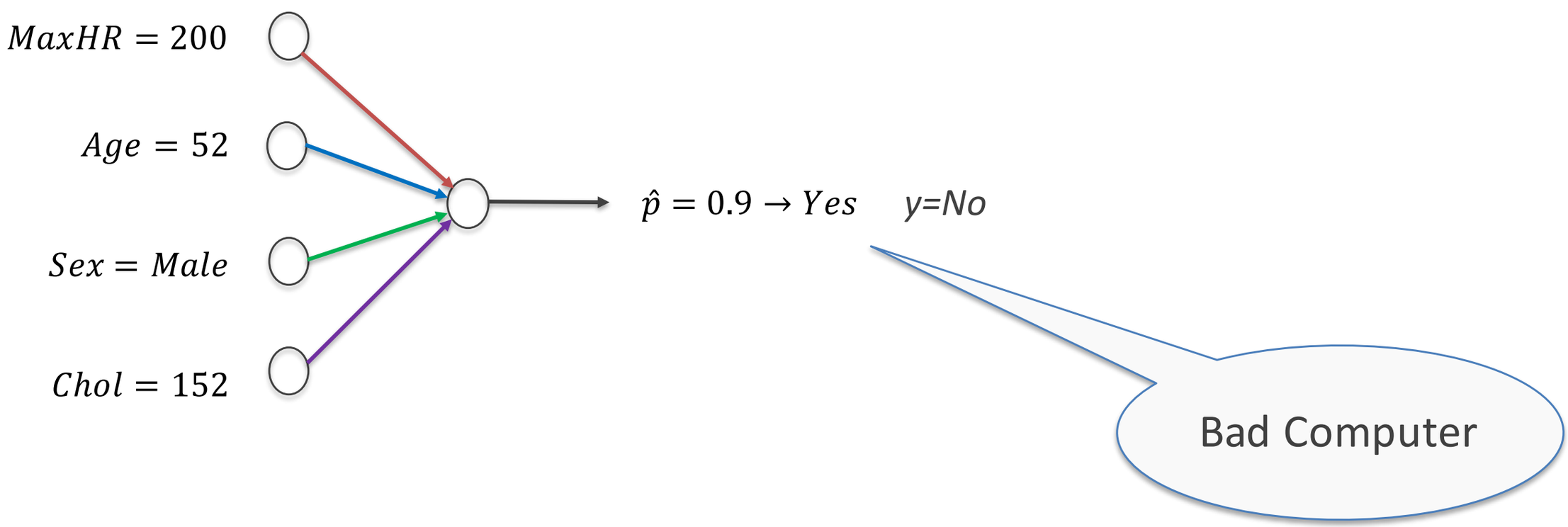
Schematic view

Start with Logistic Regression



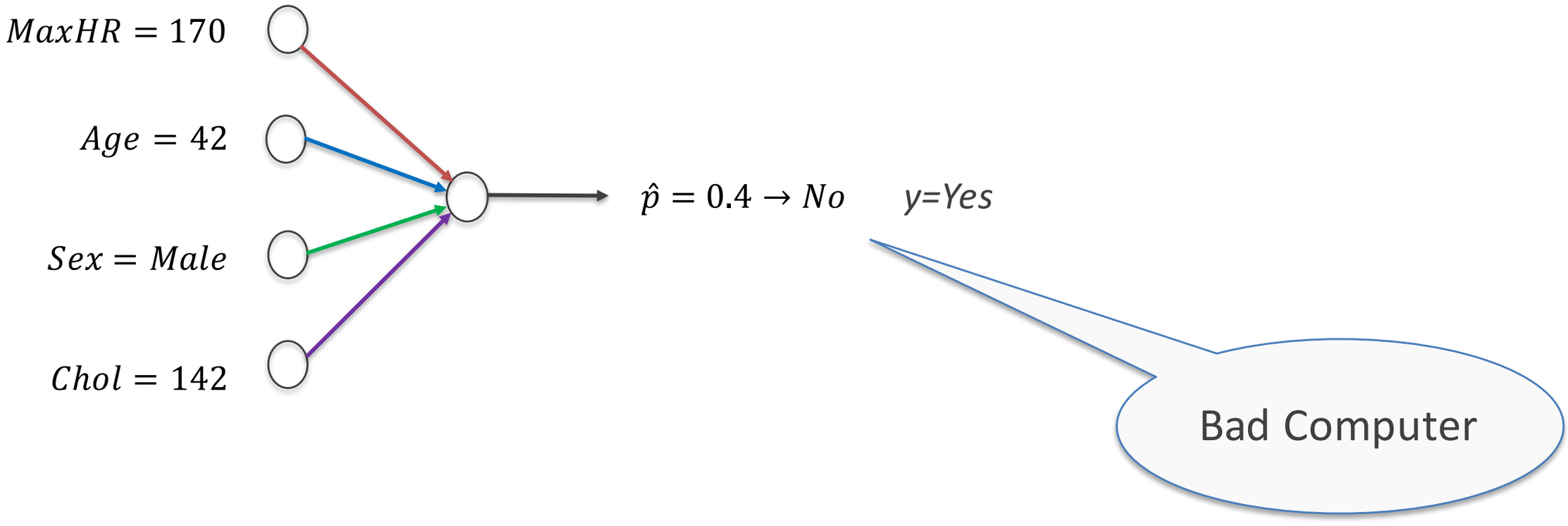
Schematic view

Start with all randomly selected weights. Most likely it will perform horribly.
For example, in our heart data, the model will be giving us the wrong answer.

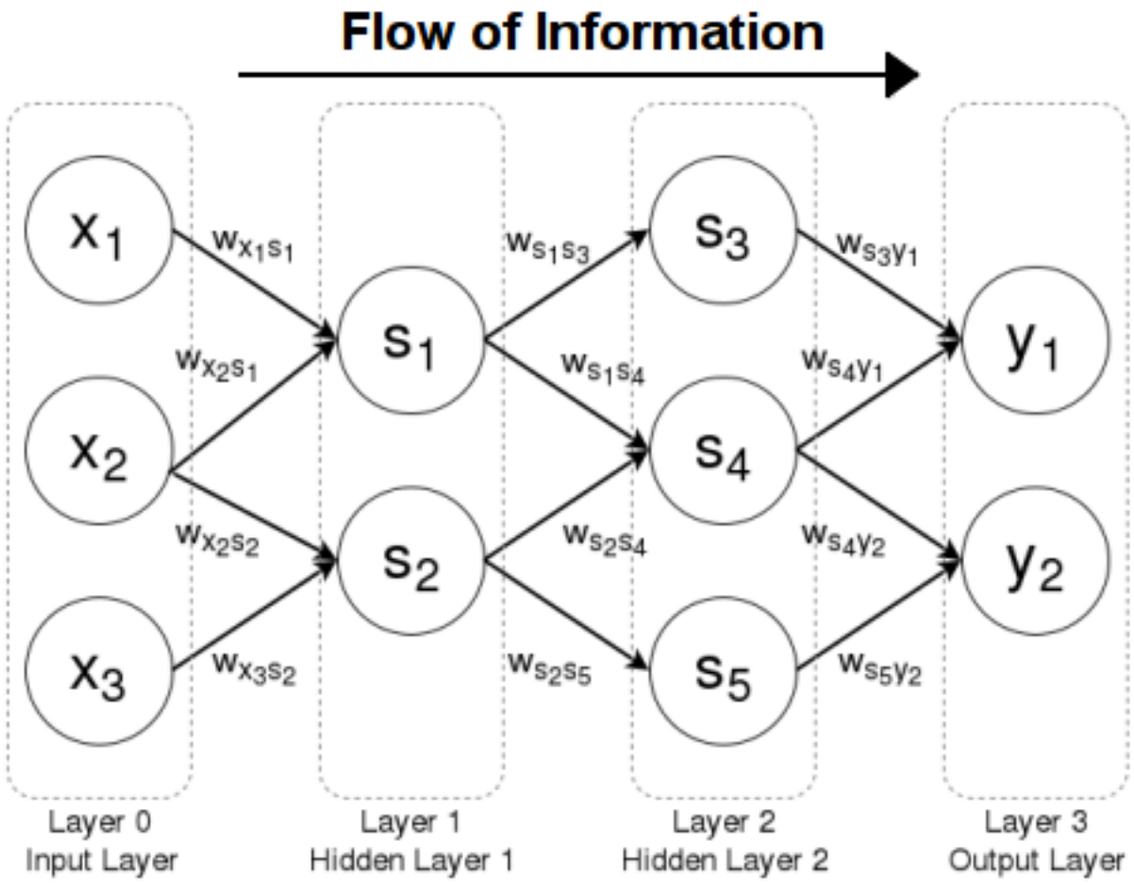


Schematic view

Start with all randomly selected weights. Most likely it will perform horribly.
For example, in our heart data, the model will be giving us the wrong answer.



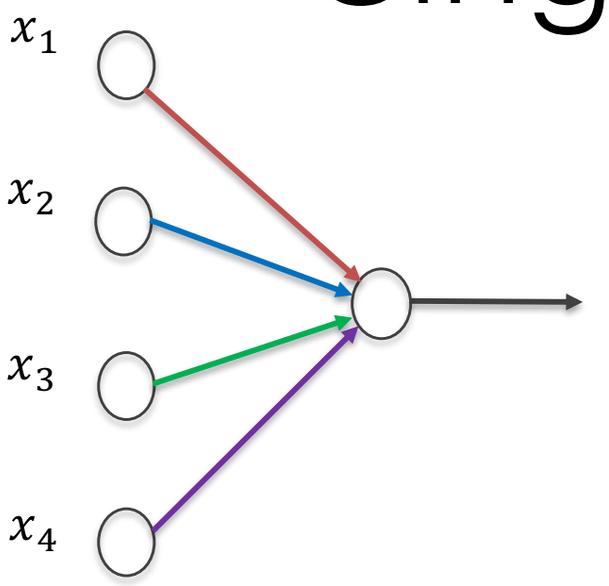
More in general



Feedforward: connections do not form a cycle

Computes a function *f* on fixed size input *x* such that *f(x) ~ y* for training pairs *(x,y)*

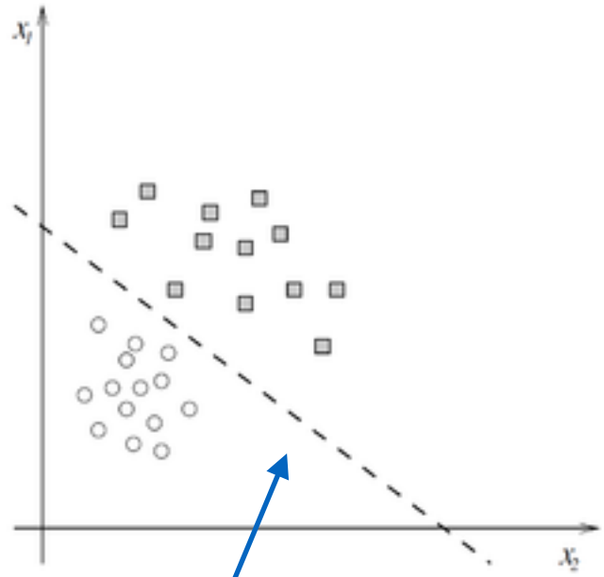
Single-layer perceptron 1



$$Y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4$$

Intercept or **Bias**

Coefficients or **Weights**



This is a straight line!
The perceptron is
a linear classifier!

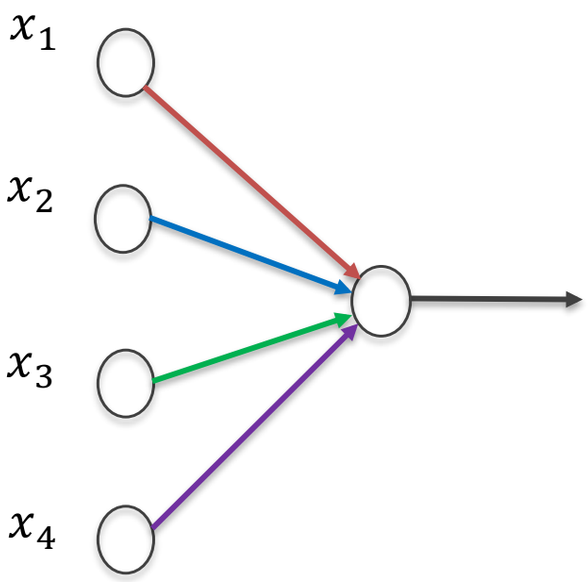
DEFINITION

$$o = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} + b \geq 0 \\ 0 & \text{if } \vec{w} \cdot \vec{x} + b < 0. \end{cases}$$

- This is the simplest form of feedforward network.
- No hidden units. Only input and output layers.
- Output is just dot product between weights and feature vectors
- Output is binary

Single-layer perceptron 2

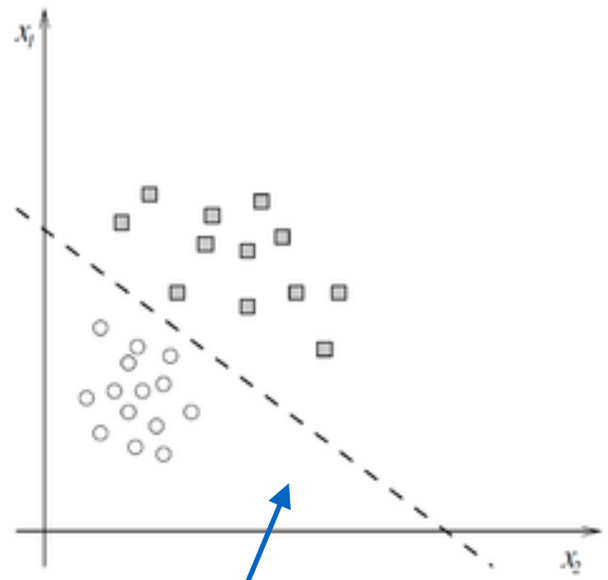
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



$$Y = \sigma(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4)$$

Intercept or **Bias**

Coefficients or **Weights**

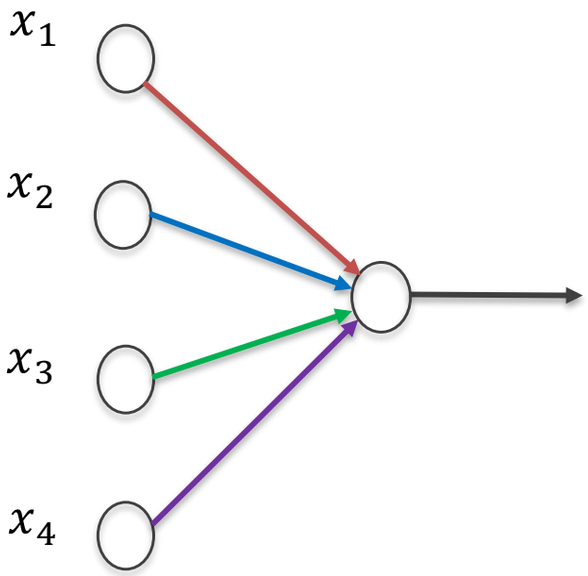


This is STILL a straight line!
The perceptron is
a linear classifier!

- We add a non-linear function, but the decision boundary is still a line.
- In many dimensions, the decision boundary is a hyperplane.
- The non-linear logistic function is called the activation function.
- Adding this non linear term allows us to interpret the output as a probability

What is learning?

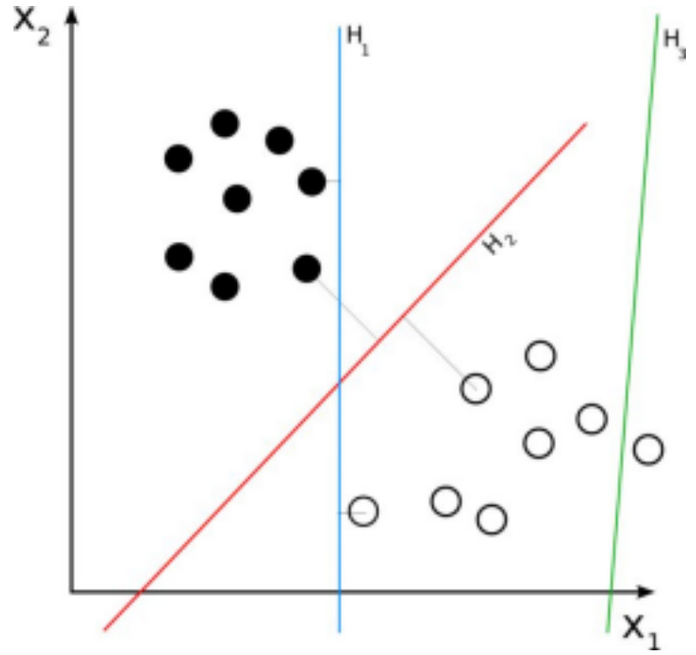
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



$$Y = \sigma(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4)$$

Intercept or **Bias**

Coefficients or **Weights**



$$o = g(\vec{w} \cdot \vec{x} + b).$$

- Learning is adjusting the weights and bias to find the best possible decision boundary.
- Learning is performed on the TRAINING DATA.
- Another way to see it: adjust the weights and bias so that g(w*x +b) -the probability- is high if dot is black and low otherwise.

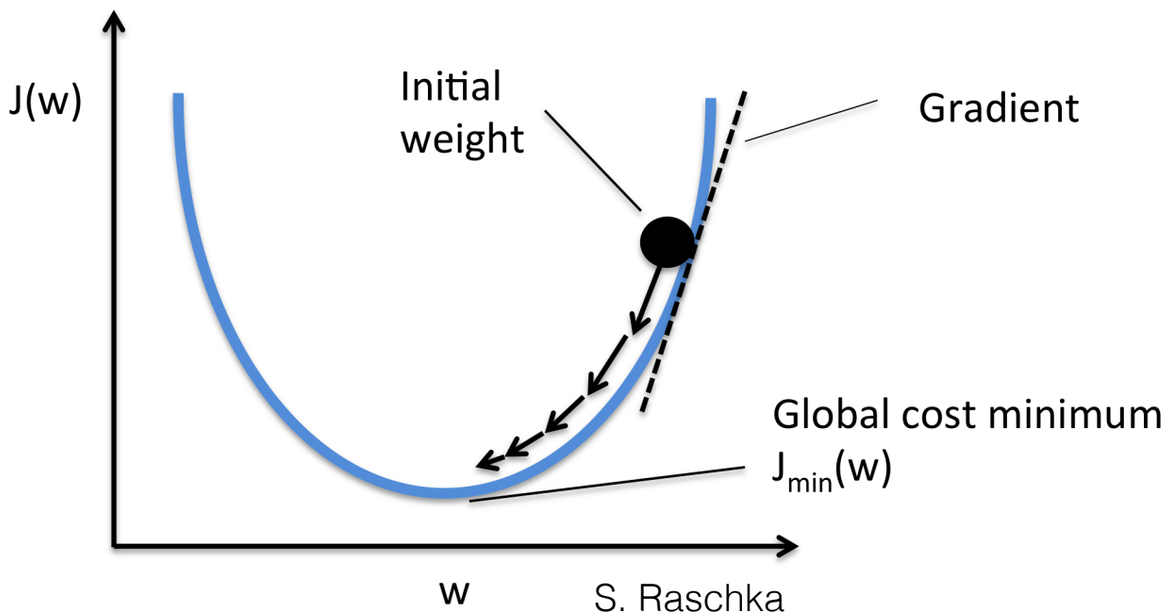
Error Function

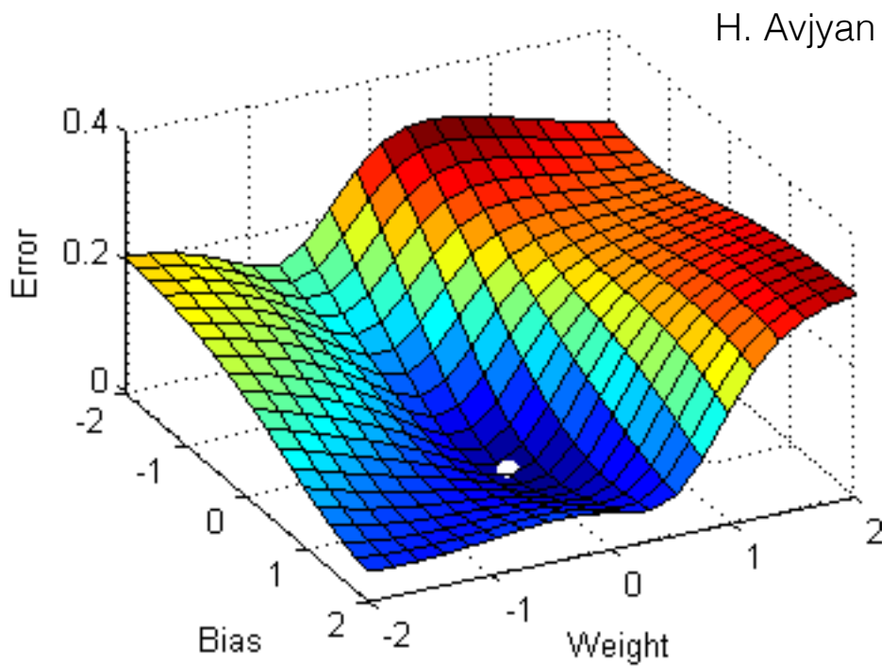
- How do we adjust the weights?
- We need to quantify the difference between the perceptron output and the true value y for an input **x**, over a set of input-output pairs.
- The mean square error (MSE) is a natural choice:

$$E(X) = \frac{1}{2N}\sum_{i=1}^{N}(o_i - y_i)^2 = \frac{1}{2N}\sum_{i=1}^{N}\left(g(\vec{w}\cdot\overrightarrow{x_i} + b) - y_i\right)^2$$

- Note that E(X) = 0 when $o_i = y_i$ for all input-output pairs.
- Therefore, we want to MINIMIZE E(X)
- How do we do that?

Gradient Descent





- Start with random bias and weights
- Use gradient of error function w.r.t. the bias and weights at each point to update those values.
- Iterate until certain convergence criterium is met.

$$\overrightarrow{w_{i+1}} = \overrightarrow{w_i} - \alpha \frac{\partial E(X)}{\partial \overrightarrow{w_i}}$$

$$b_{i+1} = b_i - \alpha \frac{\partial E(X)}{\partial b_i}$$

Learning rate!

Parenthesis: Stochastic GD

- Remember that the error function is a sum over all examples in the training set:

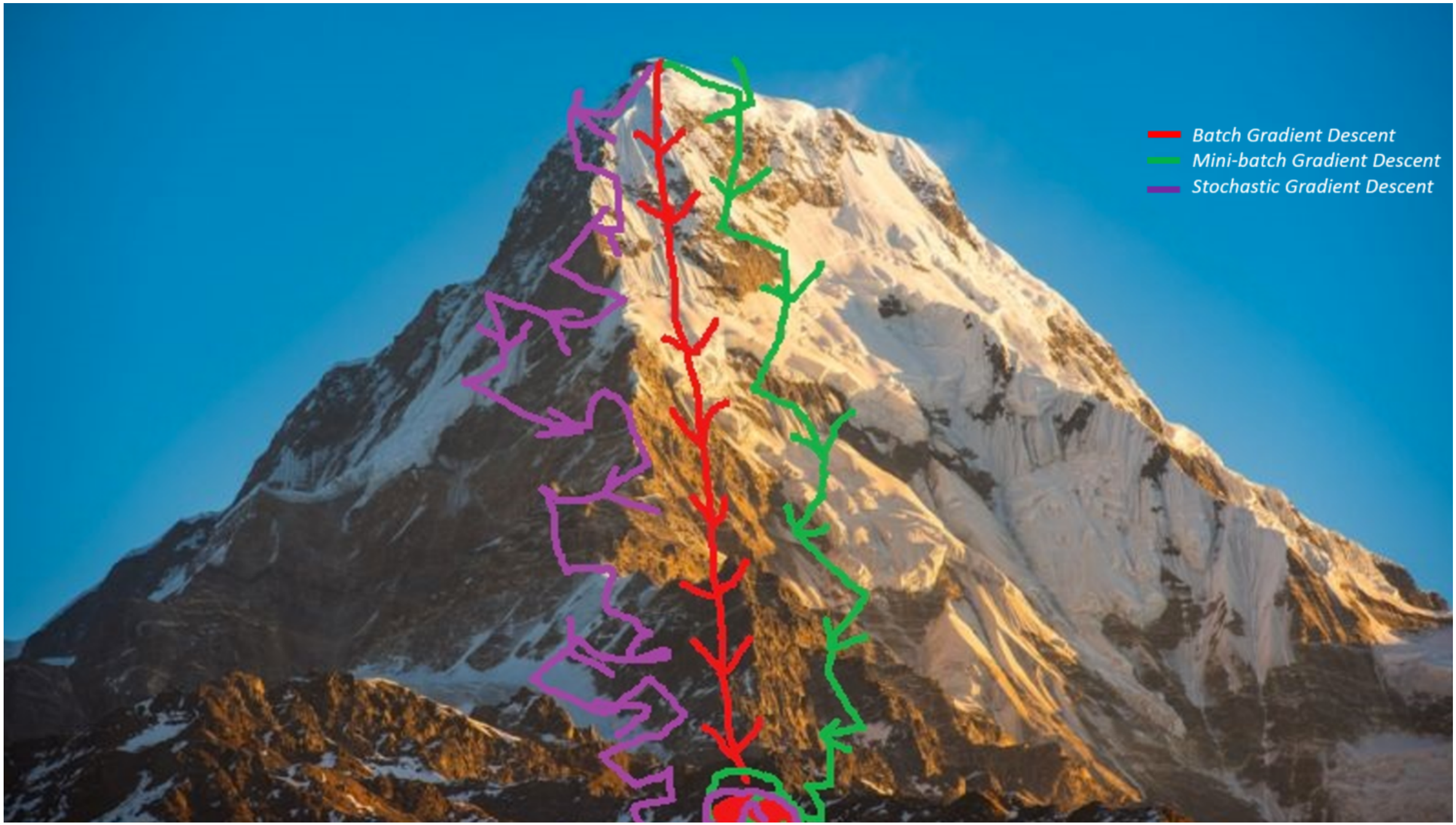
$$E(X) = \frac{1}{2N}\sum_{i=1}^{N}(o_i - y_i)^2 = \frac{1}{2N}\sum_{i=1}^{N}\left(g(\vec{w}\cdot\overrightarrow{x_i} + b) - y_i\right)^2$$

- For a single update of weights, I need to go over the entire data set. Not efficient.

- Solution 1: approximate the gradient using a single random example. Not too accurate.

- Solution 2: Perform an update using a mini-batch of examples.

Parenthesis: Stochastic GD



I. Dabbura

The Delta Rule

- Using the chain rule and power rule for derivation, it can be shown that for the simple perceptron:

$$\Delta\vec{w} = \frac{1}{N}\sum_{i=1}^{N}\alpha(y_i - o_i)g'(h_i)\overrightarrow{x_i}$$

$$\Delta b = \frac{1}{N}\sum_{i=1}^{N}\alpha(y_i - o_i)g'(h_i)$$

- where h_i = **w*x** + b. Note that error is propagated backwards (proof in whiteboard).
- This is a special case of the backpropagation algorithm, which is an efficient way to perform gradient descent.

Training the perceptron

- Given a set of *N* examples:

    1. Calculate forward values h_i and o_i, for al $\boldsymbol{x_i}$

    2. Calculate the backward values (i.e. update weights) using delta rule.

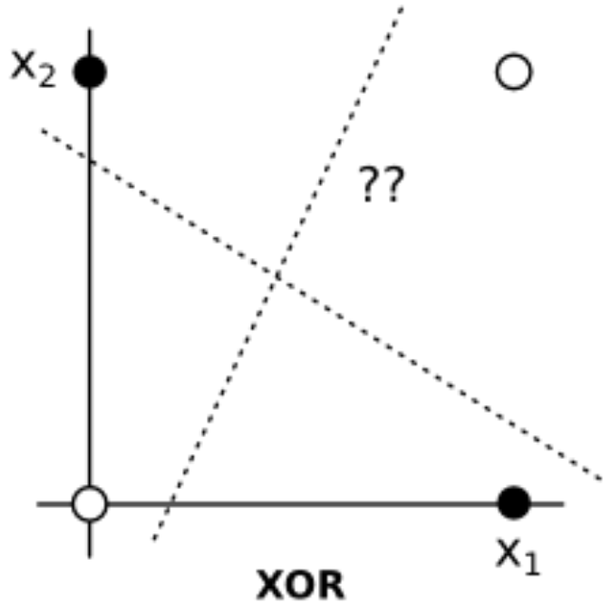
- Do this iteratively until error function converges.

- That's it. We can now fix the **w** and b and use the forward phase used to calculate the output of unseen examples.

Limitations

- The single-layer perceptron cannot learn non-linear functions.

- This means that the dataset needs to be separated by straight lines or hyperplanes.

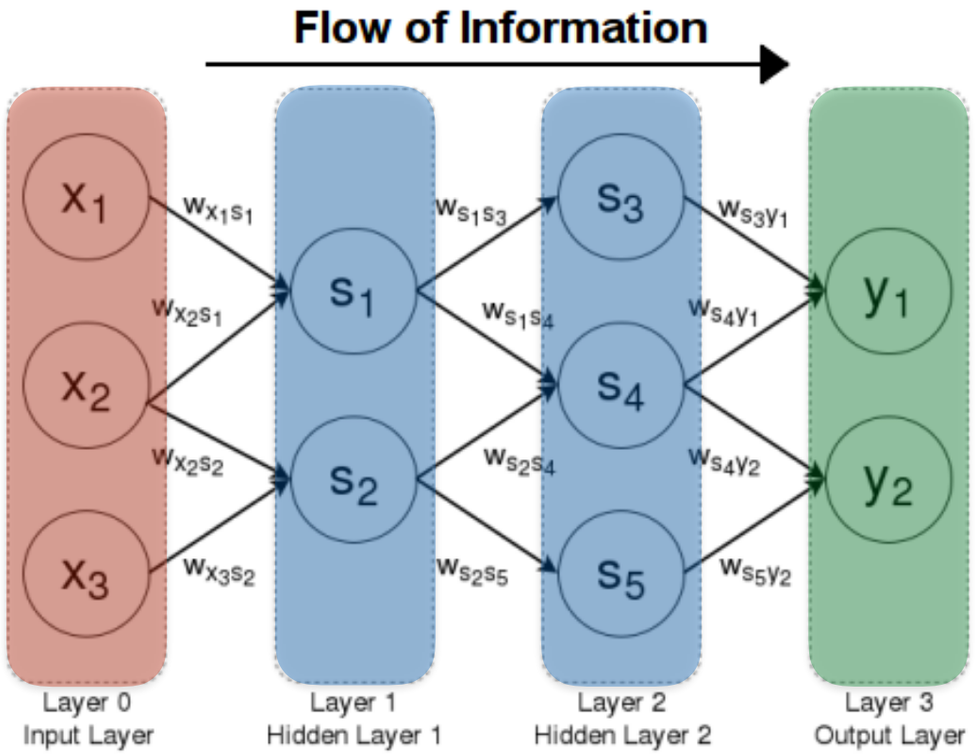


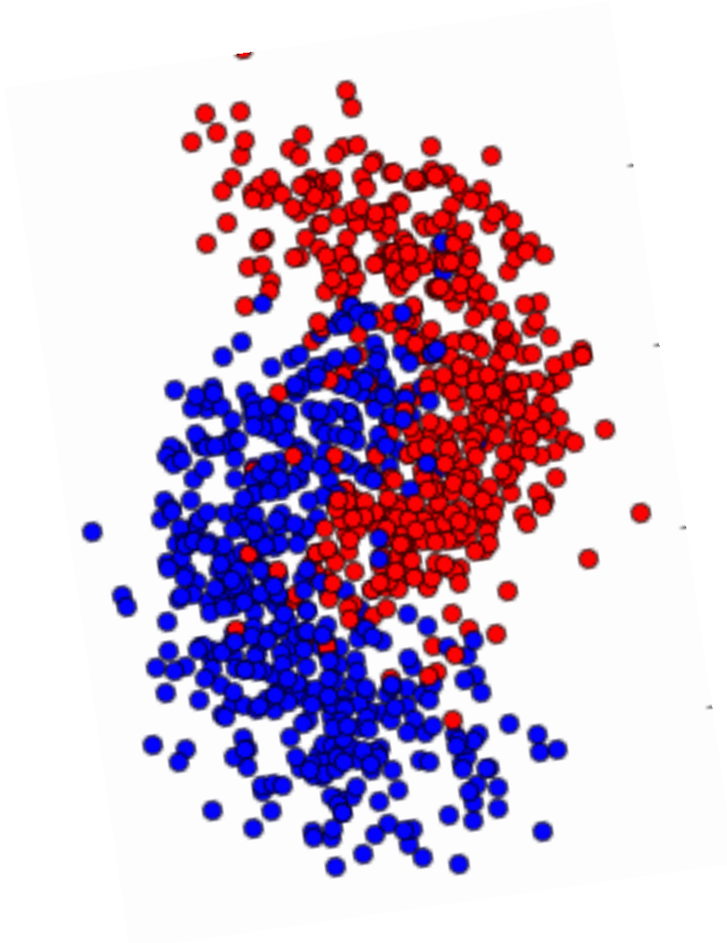
- Perceptron can’t even handle the most simple non-linear functions, such as XOR

- What do we use to learn non-linear boundaries?

X_1	X_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

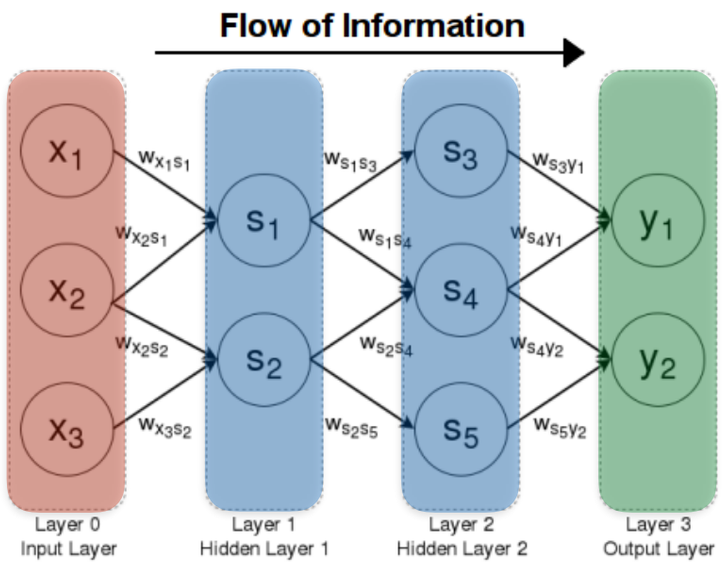
Multi-layer perceptron

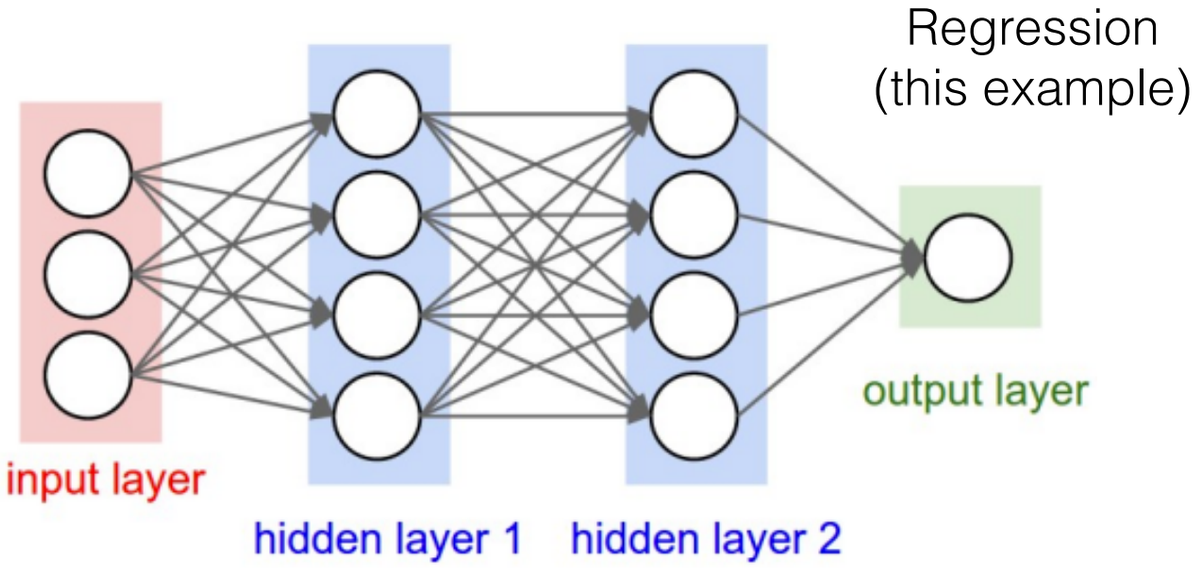




- ANN composed on many perceptrons.
- Capable of learning non-linearly separable functions.
- Good for regression and classification in supervised learning.
- The MLP is organized in layers: an input layer, some hidden layers, and an output layer.
- Regression: single neuron in the output layer
- Classification: more than one neuron in output layer
- The MLP above is said to be FULLY CONNECTED (all neurons in a given layer are connected to all neurons in the next)

Formal definition





DEFINITION

w_{ij}^k : weight for perceptron j in layer l_k for incoming node i (a perceptron if $k \geq 1$) in layer l_{k-1}

b_i^k : bias for perceptron i in layer l_k

h_i^k : product sum plus bias for perceptron i in layer l_k

o_i^k : output for node i in layer l_k

r_k : number of nodes in layer l_k

$\overrightarrow{w_i^k}$: weight vector for perceptron i in layer l_k, i.e. $\vec{w}_i^k = \left\{w_{1i}^k, \ldots, w_{r_k i}^k\right\}$

$\overrightarrow{o^k}$: output vector for layer l_k, i.e. $\overrightarrow{o^k} = \left\{o_1^k, \ldots, o_{r_k}^k\right\}$

Computing the output

EXAMPLE

1. **Initialize the input layer l_0**

   Set the values of the outputs o_i^0 for nodes in the input layer l_0 to their associated inputs in the vector $\vec{x}$ = $o_i^0 = x_i$.

2. **Calculate the product sums and outputs of each hidden layer in order from l_1 to l_{m-1}**

   For k from 1 to $m-1$,

   a. compute $h_i^k = \overrightarrow{w_i^k} \cdot \overrightarrow{o^{k-1}} + b_i^k = b_i^k + \sum_{j=1}^{r_{k-1}} w_{ji}^k o_j^{k-1}$ for $i = 1, \ldots, r_k$;

   b. compute $o_i^k = g\left(h_i^k\right)$ for $i = 1, \ldots, r_k$.

3. **Compute the output y for output layer l_m**

   a. Compute $h_1^m = \overrightarrow{w_1^m} \cdot \overrightarrow{o^{m-1}} + b_1^m = b_1^m + \sum_{j=1}^{r_{m-1}} w_{j1}^k o_j^{k-1}$.

   b. Compute $o = o_1^m = g_o(h_1^m)$.

Training the MLP: Backpropagation

- Just as before, iteratively update the values of w and b until E(X) is minimized.

$$E(X,\theta) = \frac{1}{2N}\sum_{i=1}^{N}\left(\hat{y}_i - y_i\right)^2$$

- So we want to know how the change in E(X) depends on changes of each weight. **Problem is that intermediate layer neurons have no target output.**
- But E(X) is a sum over examples, and so it's derivative with respect to each weight is a sum of derivatives:

$$\frac{\partial E(X,\theta)}{\partial w_{ij}^k} = \frac{1}{N}\sum_{d=1}^{N}\frac{\partial}{\partial w_{ij}^k}\left(\frac{1}{2}\left(\hat{y}_d - y_d\right)^2\right) = \frac{1}{N}\sum_{d=1}^{N}\frac{\partial E_d}{\partial w_{ij}^k}$$

- So all we care about is the error for a single example:

$$E = \frac{1}{2}\left(\hat{y} - y\right)^2$$

Training the MLP: Backpropagation

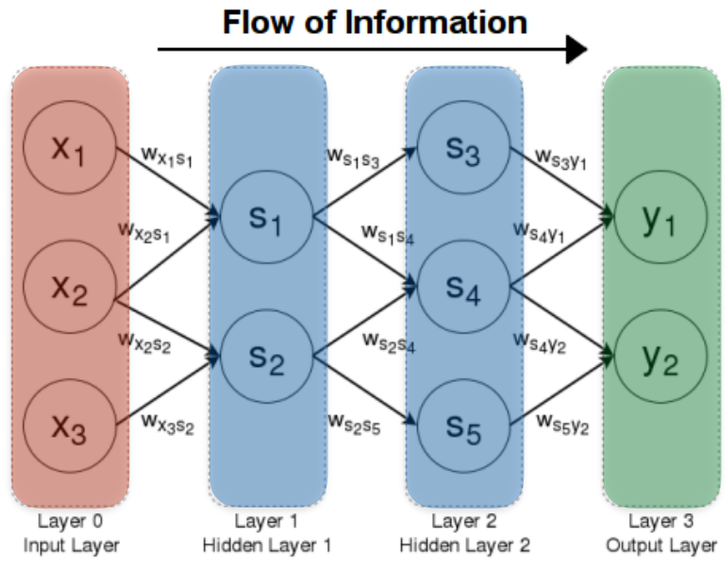
- Now, we can make use of the chain rule:

$$\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k}$$

$$\delta_j^k \equiv \frac{\partial E}{\partial a_j^k}$$

- But:

$$\frac{\partial a_j^k}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k} \left(\sum_{l=0}^{r_{k-1}} w_{lj}^k o_l^{k-1} \right) = o_i^{k-1}$$



- And therefore:

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}$$

- **The partial derivative of the error with respect to the weight for neuron *j* in layer *k*, for incoming neuron *i*, is the product of its derivative with respect to the linear activation in that neuron and the output of the incoming neuron in layer *k-1*.**

Training the MLP: Backpropagation

- Hey, but we have not calculated $\delta_j^k \equiv \frac{\partial E}{\partial a_j^k}$
- Spoiler: it depends on the values or error terms in the next layer - that's why we need to go backwards.
- **Part 1**: Output layer. Let's find δ_1^m first. The error function for this layer is:

$$E = \frac{1}{2}(\hat{y} - y)^2 = \frac{1}{2}\left(g_o(a_1^m) - y\right)^2$$

- Lets apply the chain rule:

$$\delta_1^m = \left(g_0(a_1^m) - y\right) g_o'(a_1^m) = (\hat{y} - y) g_o'(a_1^m)$$

- Therefore:

$$\frac{\partial E}{\partial w_{i1}^m} = \delta_1^m o_i^{m-1} = (\hat{y} - y) g_o'(a_1^m)\, o_i^{m-1}$$

- Easy¡

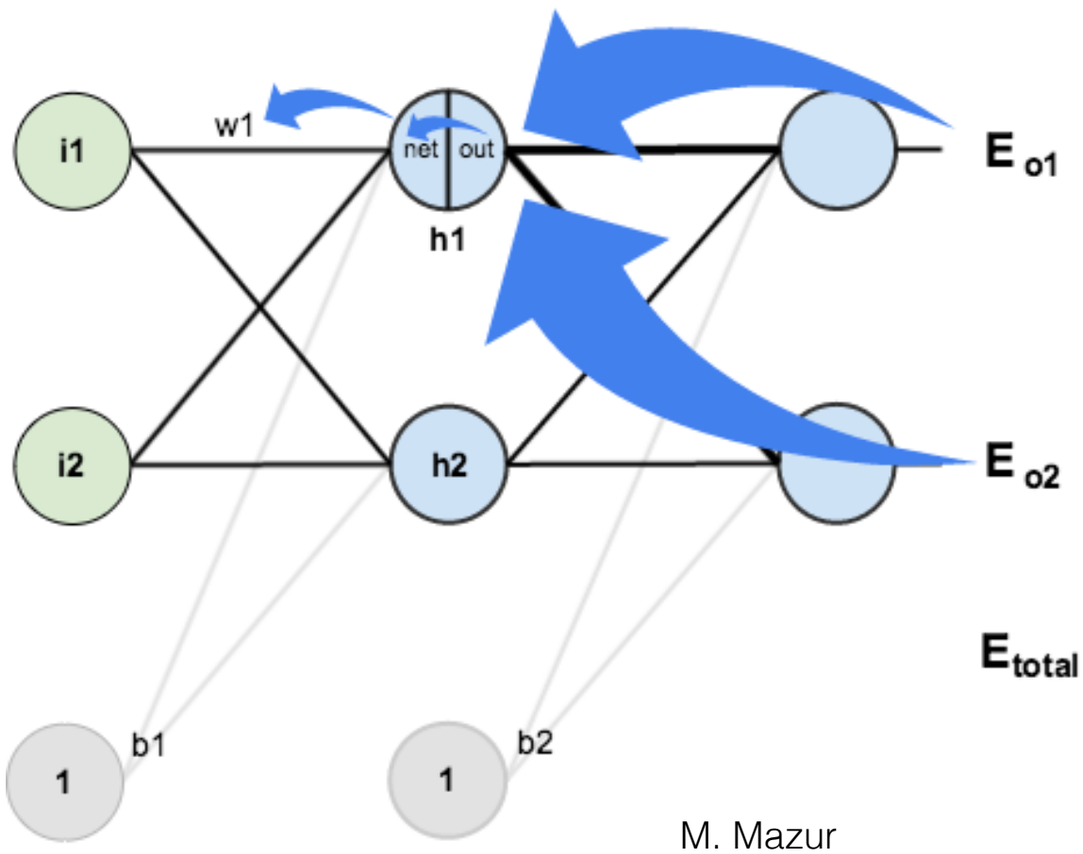
Training the MLP: Backpropagation

- **Part 2**: Hidden layers. For the output layer we have a target. Not so for the hidden layers. What to do?? Worry not, as the chain rule comes to the rescue:

$$\delta_j^k = \frac{\partial E}{\partial a_j^k} = \sum_{l=1}^{r^{k+1}} \frac{\partial E}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{\partial a_j^k}$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



$$E_{total} = E_{o1} + E_{o2}$$

- Note that it depends on the values or error terms in the next layer - that's why we need to go backwards. Graphically:

M. Mazur

Training the MLP: Backpropagation

- Let's plug in some things we know:

$$\delta_j^k = \sum_{l=1}^{r^{k+1}} \delta_l^{k+1} \frac{\partial a_l^{k+1}}{\partial a_j^k}$$

$$a_l^{k+1} = \sum_{j=1}^{r^k} w_{jl}^{k+1} g\left(a_j^k\right) \longrightarrow \frac{\partial a_l^{k+1}}{\partial a_j^k} = w_{jl}^{k+1} g'\left(a_j^k\right)$$

- Plugging this in the definition of the error term gives the **backpropagation formula:**

$$\delta_j^k = \sum_{l=1}^{r^{k+1}} \delta_l^{k+1} w_{jl}^{k+1} g'\left(a_j^k\right) = g'\left(a_j^k\right) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}$$

- And therefore the derivative of the total error w.r.t. the weight in a hidden layer is:

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} = g'\left(a_j^k\right) o_i^{k-1} \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}$$

The general algorithm

1. For each example in the training set, calculate the forward phase starting with the input layer. Get output.

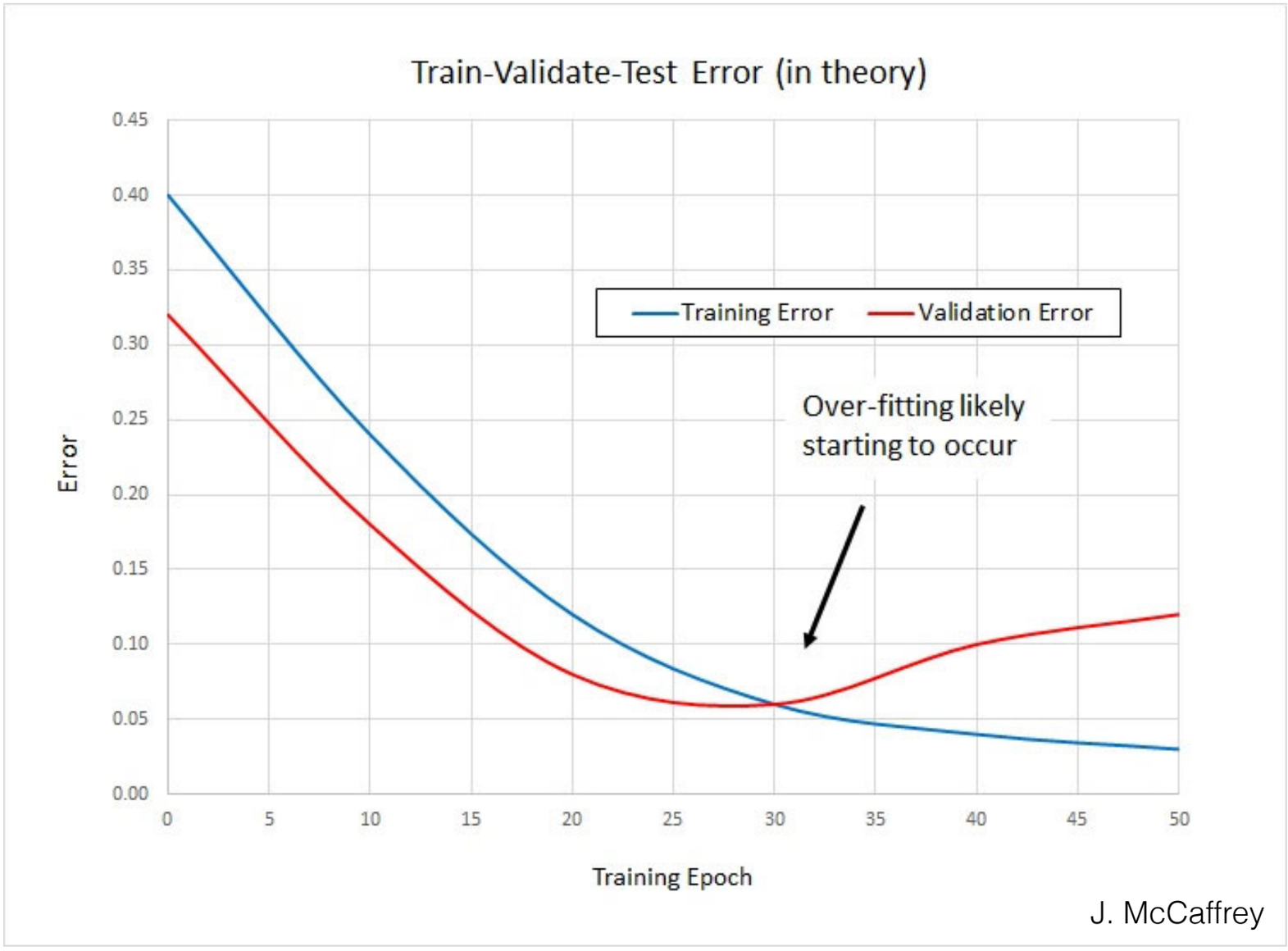
2. For each example, calculate the backward phase, starting from the error in the output layer, and propagating the error backwards:

    - Evaluate error term in the output layer.
    - Back propagate the error in the hidden layers using the formulas derived.
    - Evaluate the partial derivative of the total error wrt each weight

3. Combine the individual gradients by averaging them.

4. Update the weights with a certain learning rate.

When to stop training? Validation



- Your training set error can become arbitrarily small.

- But at some point, you will start overfitting and your validation error will start to increase.

Regularization

- Use a Bayesian approach to avoid over-fitting.
- Set priors to the coefficients of the model. Make them smaller, so that the model is less complex.
- Discourage overfitting by simplifying the model:

$$\mathrm{RSS} = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 .$$

No regularization

$$\sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j x_{ij} \right)^2 + \lambda \sum_{j=1}^{p} \beta_j^2 = \mathrm{RSS} + \lambda \sum_{j=1}^{p} \beta_j^2$$

Regularize