Budapest University of Technology and Economics
Department of Artificial Intelligence and Systems Engineering

Artificial intelligence – VIMIAC16-EN, VIMIAC10

2024 Fall Semester

Dr. Gábor Hullám

Slides Adapted from Berkeley CS188, from Dan Klein, Pieter Abbeel and Sergey Levine
http://ai.berkeley.edu

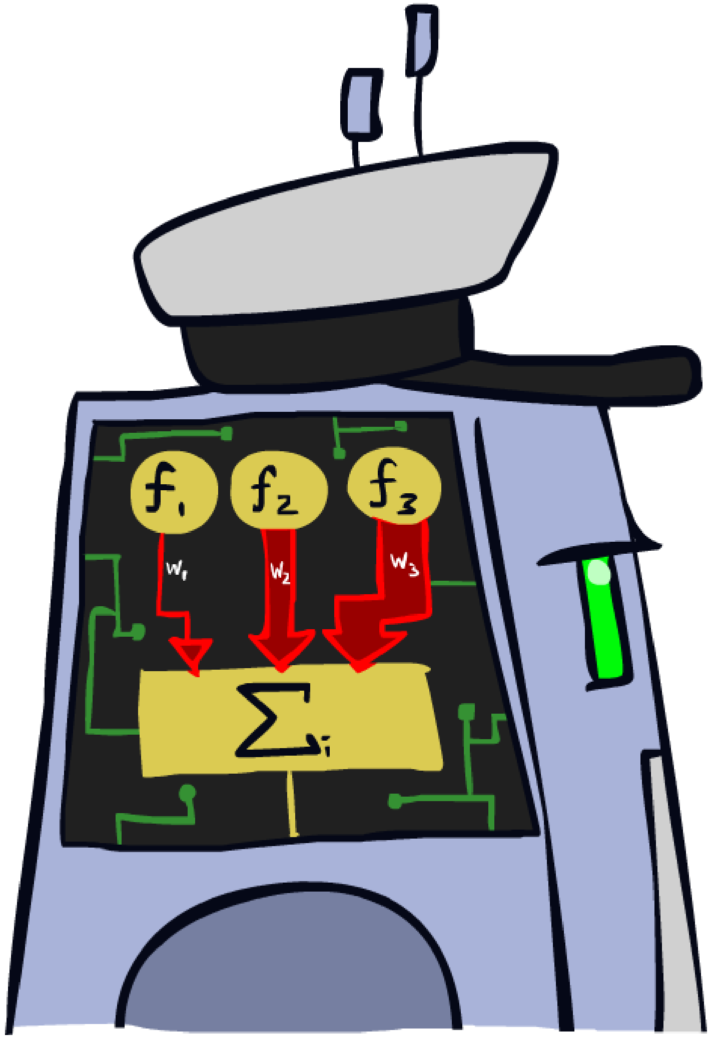
MŰEGYETEM 1782

Artificial intelligence lectures

Az előadás diái az AIMA könyvre épülve (http://aima.cs.berkeley.edu) készültek a University of California, Berkeley mesterséges intelligencia kurzusának anyagainak felhasználásával (http://ai.berkeley.edu).

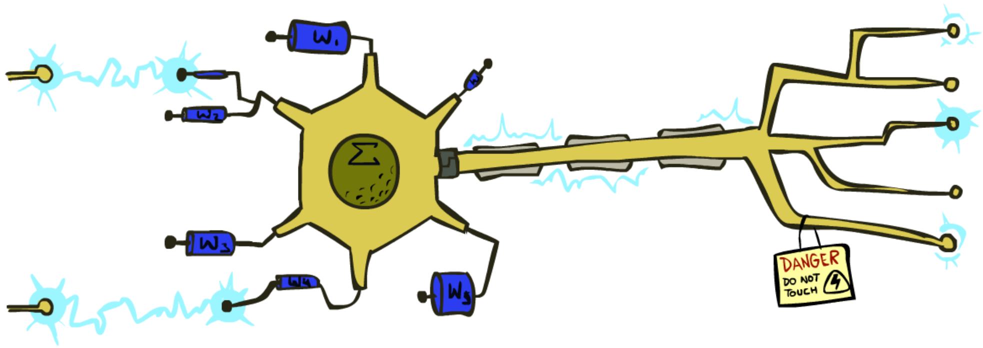
These slides are based on the AIMA book (http://aima.cs.berkeley.edu) and were adapted from the AI course material of University of California, Berkeley (http://ai.berkeley.edu).

Linear Models

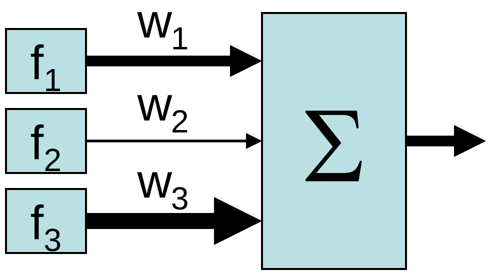


Linear Models

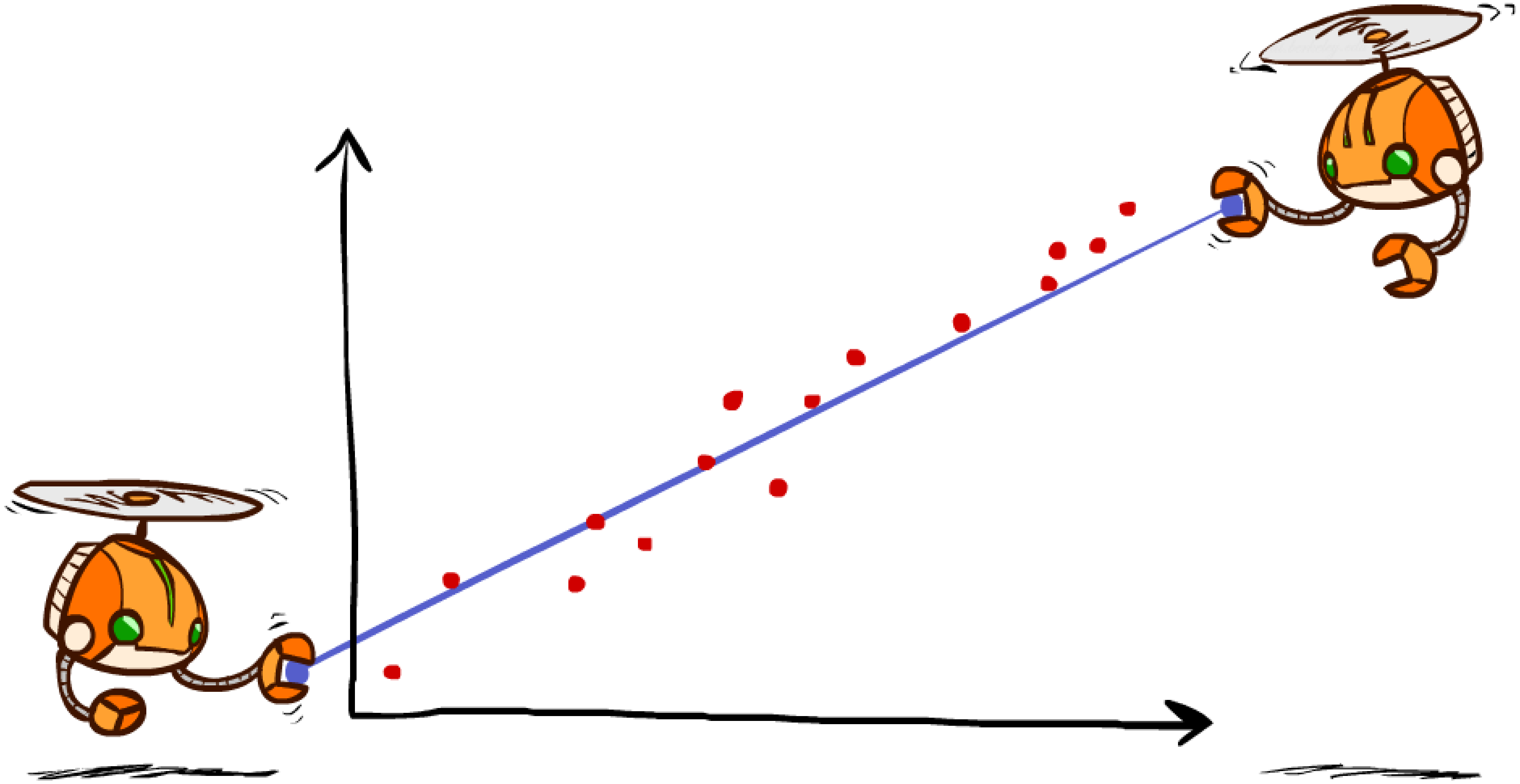
- Inputs are feature values
- Each feature has a weight
- Sum is the activation



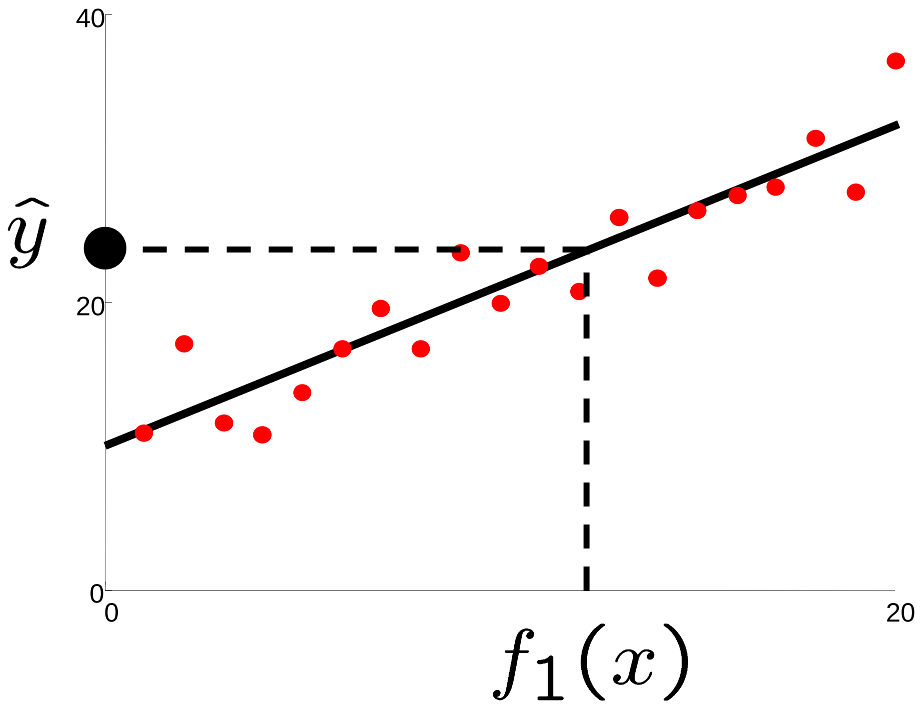
$$\text{activation}_w(x) = \sum_i w_i \cdot f_i(x) = w \cdot f(x)$$

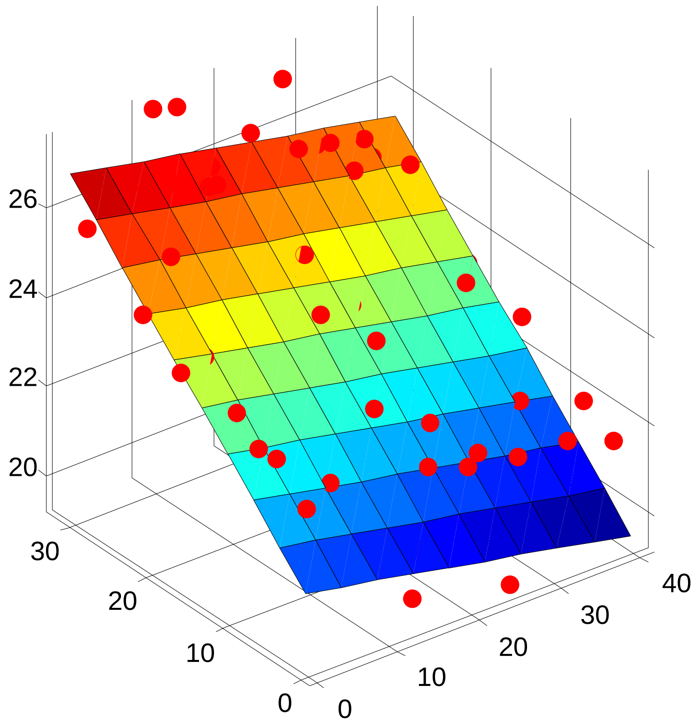


Linear regression



Linear Approximation: Regression





Prediction:

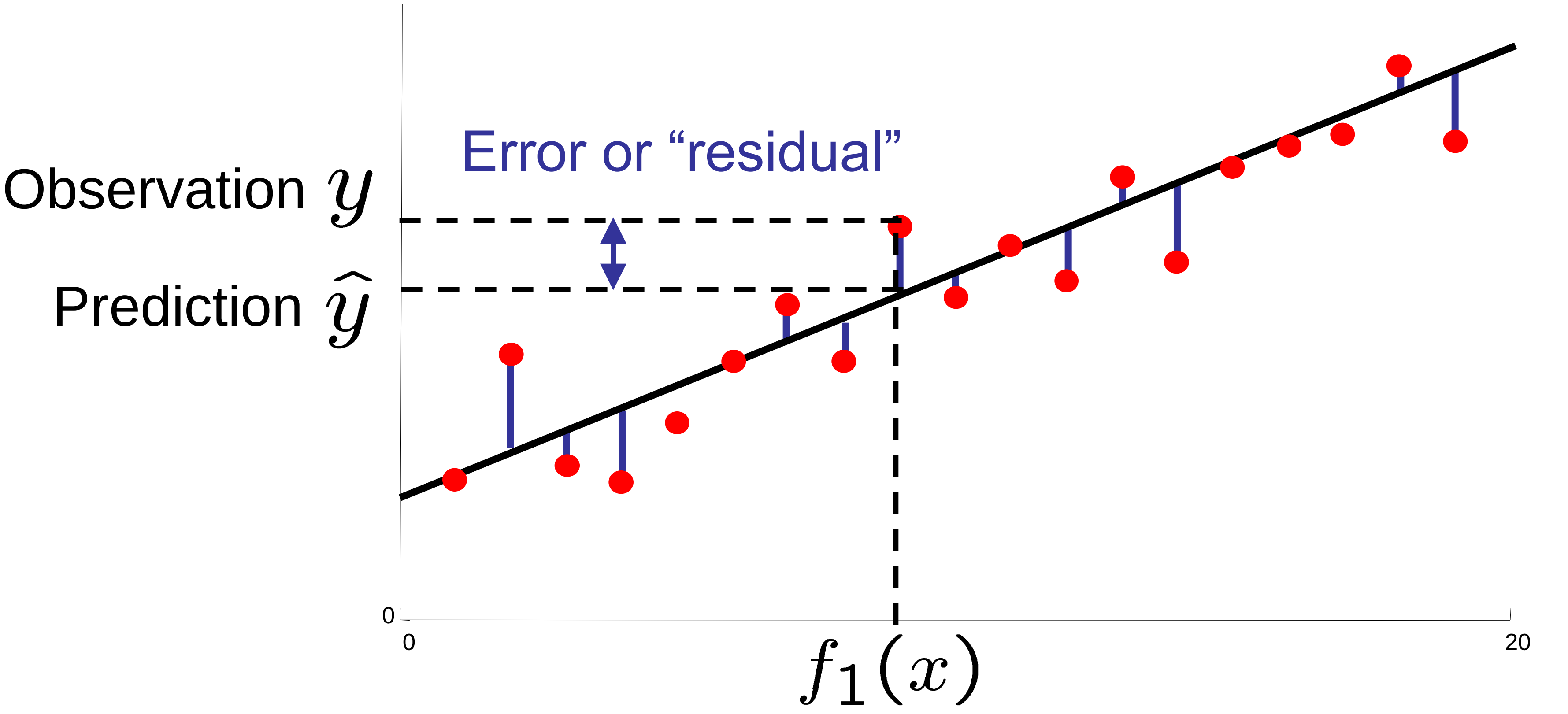
$$\hat{y} = w_0 + w_1 f_1(x)$$

Prediction:

$$\hat{y}_i = w_0 + w_1 f_1(x) + w_2 f_2(x)$$

Optimization: Least Squares

$$\text{total error} = \sum_i (y_i - \hat{y}_i)^2 = \sum_i \left(y_i - \sum_k w_k f_k(x_i) \right)^2$$



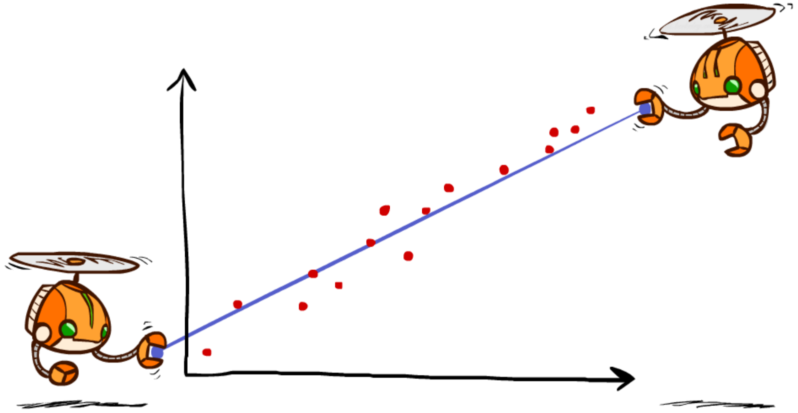
Minimizing Error

Imagine we had only one point x, with features f(x), target value y, and weights w:

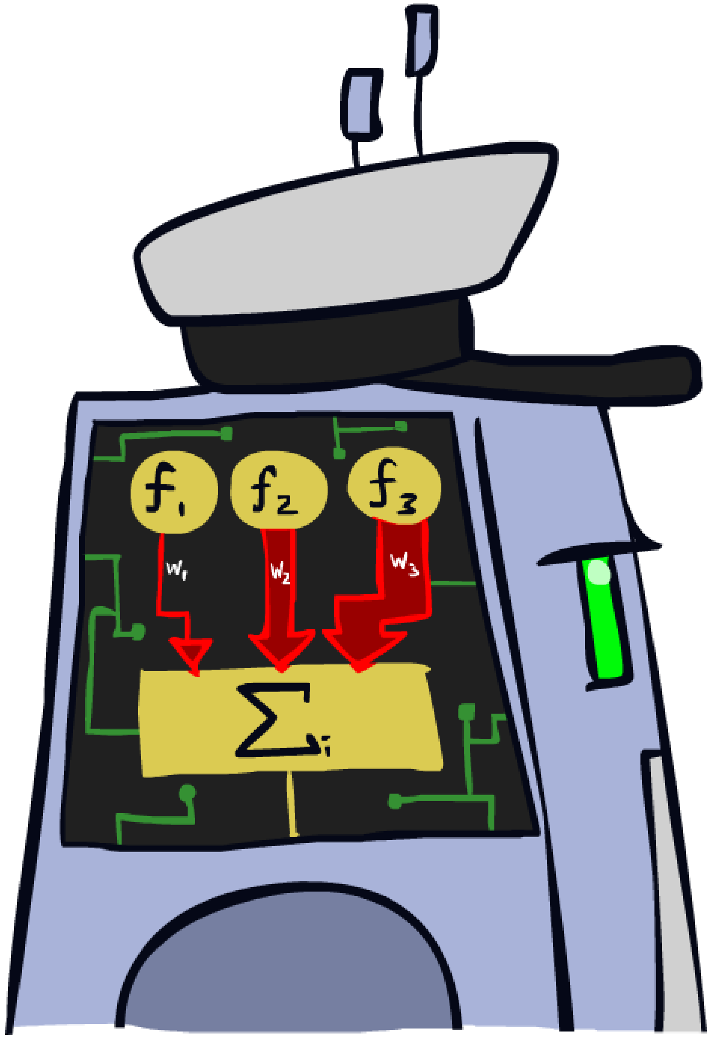
$$\text{error}(w) = \frac{1}{2}\left(y - \sum_k w_k f_k(x)\right)^2$$

$$\frac{\partial\ \text{error}(w)}{\partial w_m} = -\left(y - \sum_k w_k f_k(x)\right) f_m(x)$$

$$w_m \leftarrow w_m + \alpha \left(y - \sum_k w_k f_k(x)\right) f_m(x)$$



Linear Classifiers



Feature Vectors

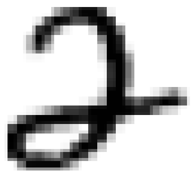
x $f(x)$ y

```
Hello,

Do you want free printr
cartriges?  Why pay more
when you can get them
ABSOLUTELY FREE!  Just
```

```
# free       : 2
YOUR_NAME    : 0
MISSPELLED   : 2
FROM_FRIEND  : 0
...
```

SPAM
or
+

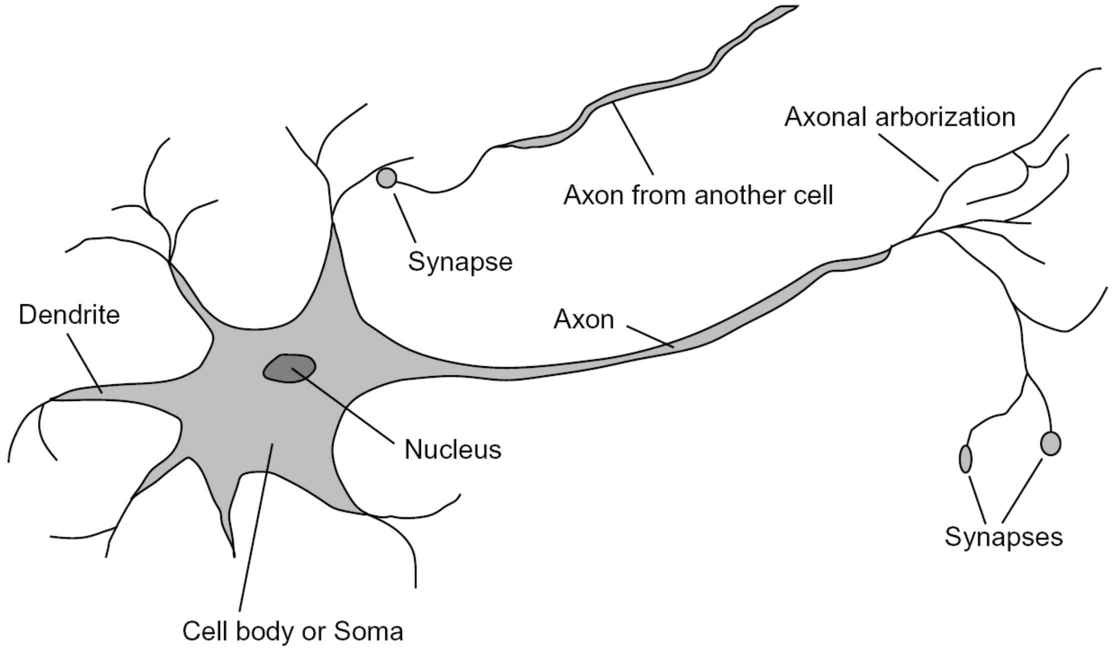


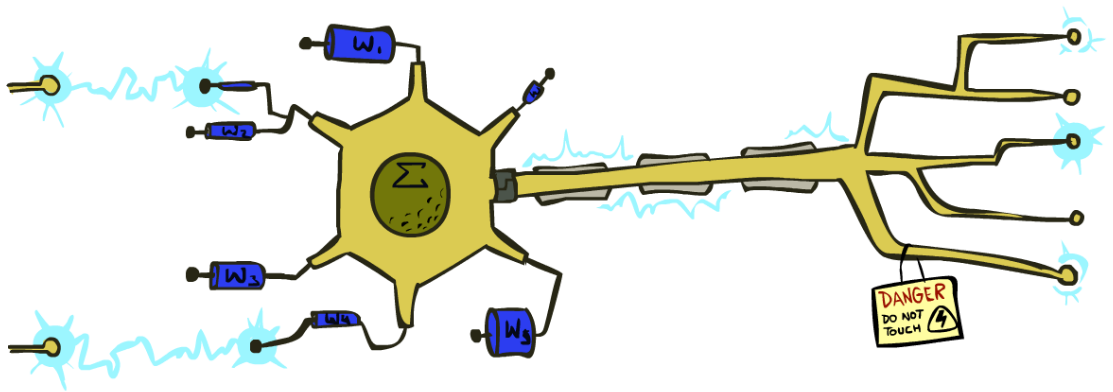
```
PIXEL-7,12   : 1
PIXEL-7,13   : 0
...
NUM_LOOPS    : 1
...
```

“2”

Some (Simplified) Biology

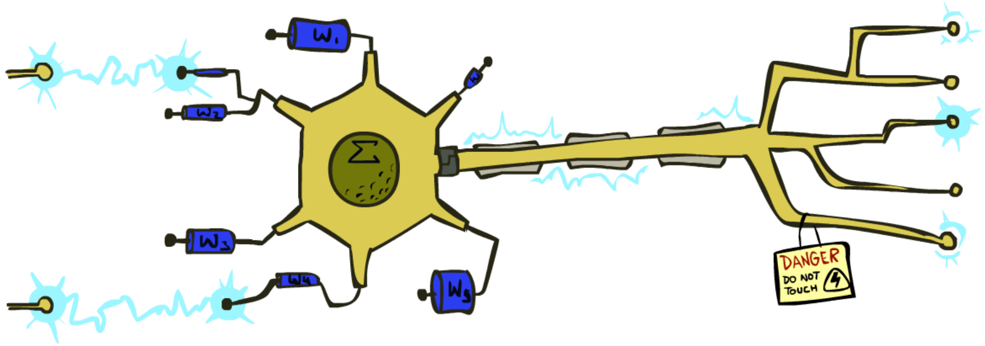
- Very loose inspiration: human neurons





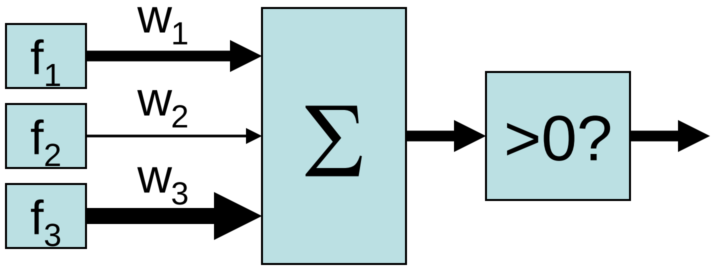
Linear Classifiers

- Inputs are feature values
- Each feature has a weight
- Sum is the activation



$$\text{activation}_w(x) = \sum_i w_i \cdot f_i(x) = w \cdot f(x)$$

- If the activation is:
  - Positive, output +1
  - Negative, output -1

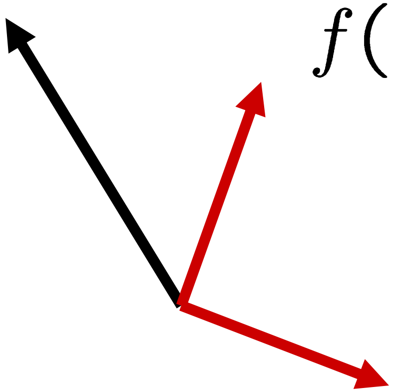


Weights

- Binary case: compare features to a weight vector
- Learning: figure out the weight vector from examples

```
# free      : 4
YOUR_NAME   :-1
MISSPELLED  : 1
FROM_FRIEND :-3
...
```

w



$f(x_1)$

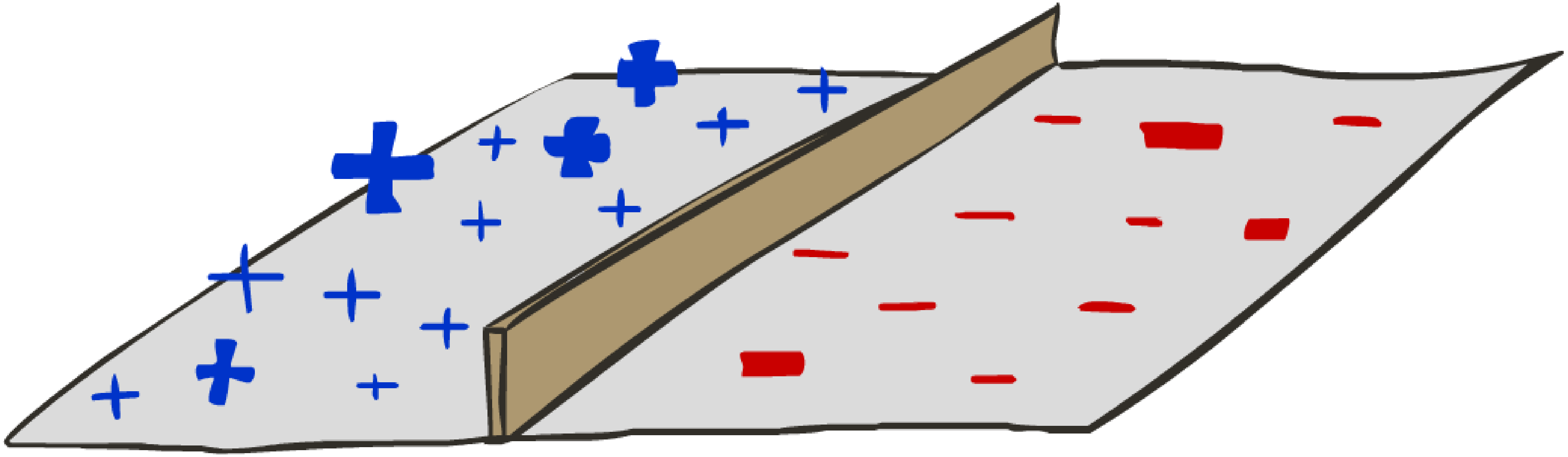
```
# free      : 2
YOUR_NAME   : 0
MISSPELLED  : 2
FROM_FRIEND : 0
...
```

$f(x_2)$

```
# free      : 0
YOUR_NAME   : 1
MISSPELLED  : 1
FROM_FRIEND : 1
...
```

Dot product $w \cdot f$ *positive means the positive class*

Decision Rules

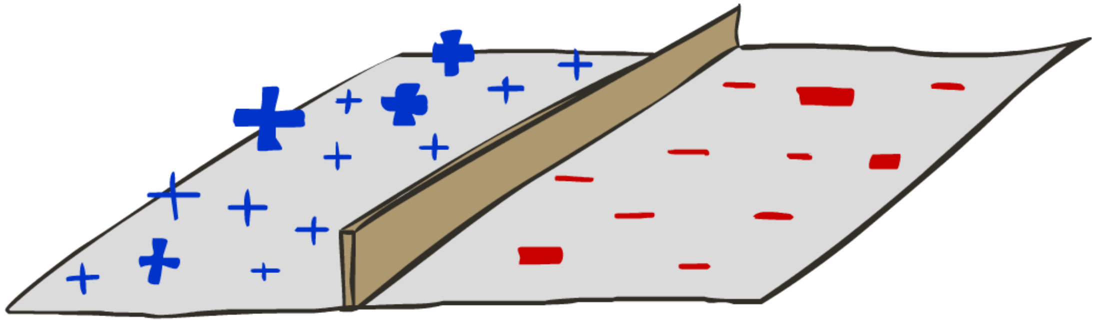


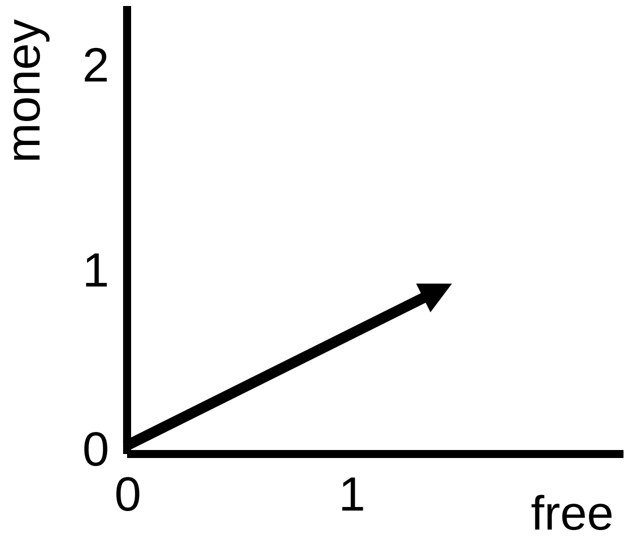
Binary Decision Rule

- In the space of feature vectors
  - Examples are points
  - Any weight vector is a hyperplane
  - One side corresponds to Y=+1
  - Other corresponds to Y=-1

w

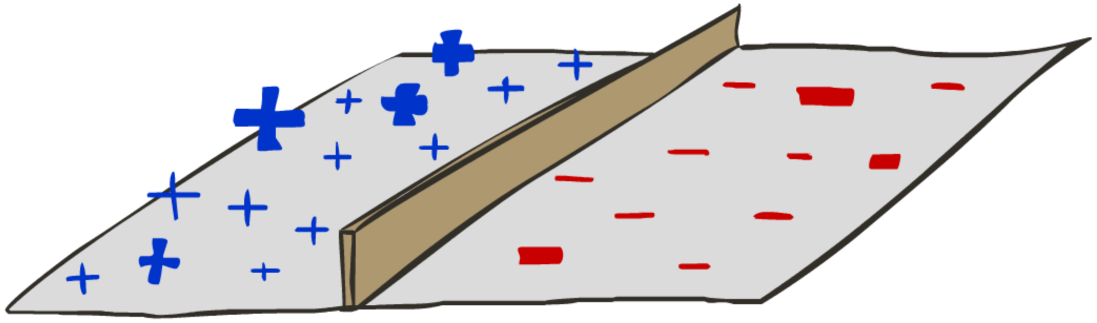
```
BIAS  : -3
free  :  4
money :  2
...
```





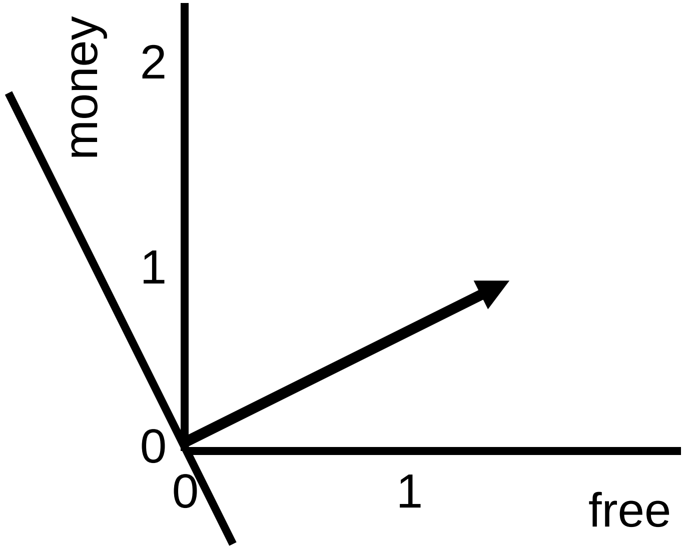
Binary Decision Rule

- In the space of feature vectors
  - Examples are points
  - Any weight vector is a hyperplane
  - One side corresponds to Y=+1
  - Other corresponds to Y=-1



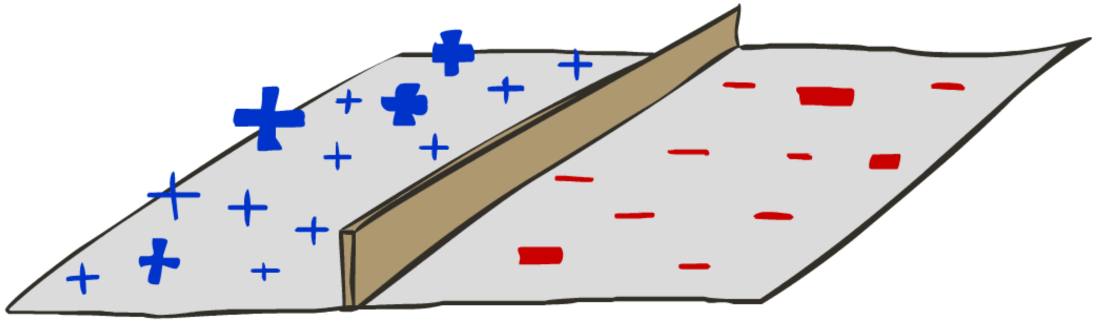
w

```
BIAS  : -3
free  :  4
money :  2
...
```



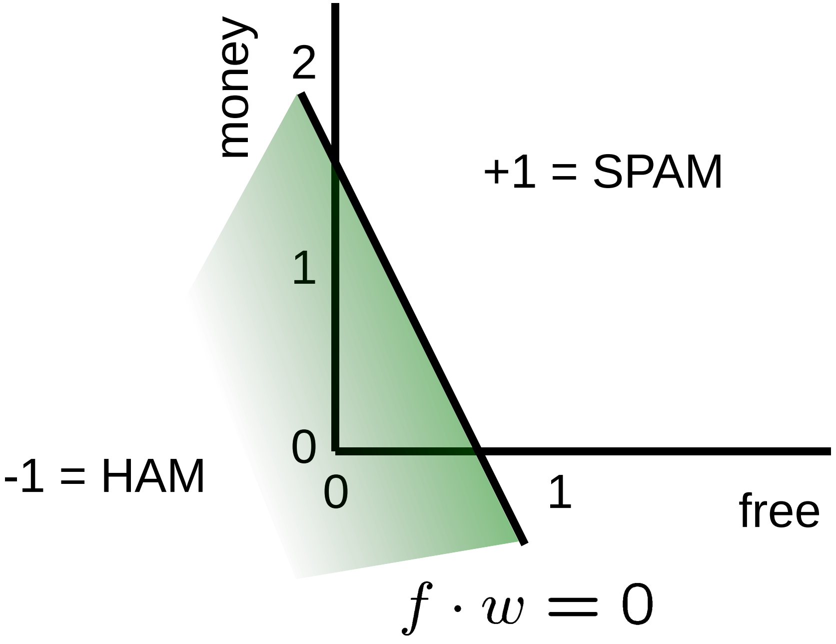
Binary Decision Rule

- In the space of feature vectors
  - Examples are points
  - Any weight vector is a hyperplane
  - One side corresponds to Y=+1
  - Other corresponds to Y=-1



w

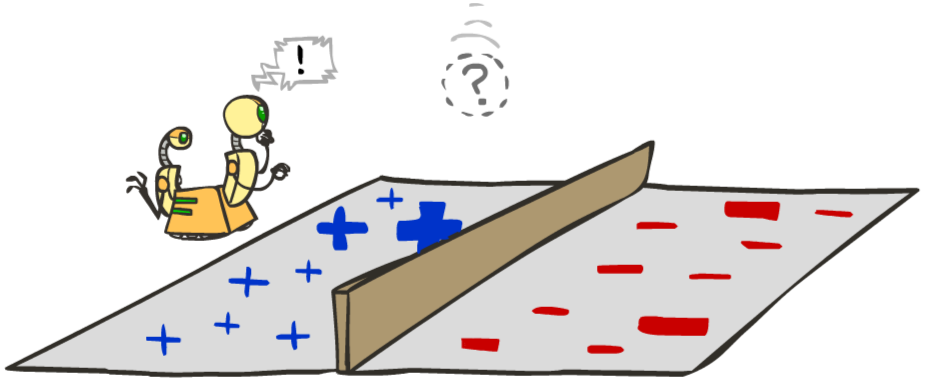
BIAS	:	-3
free	:	4
money	:	2
...		



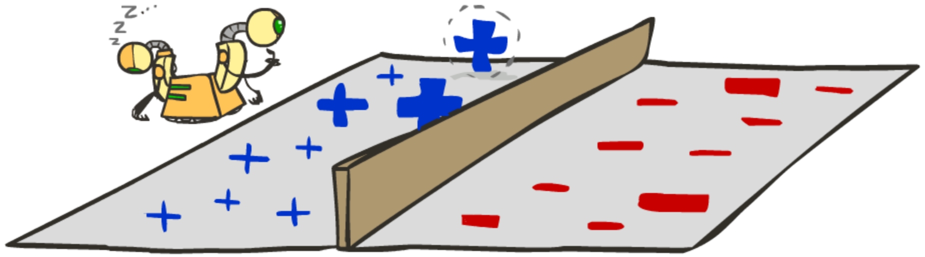
Weight Updates

Learning: Binary Perceptron

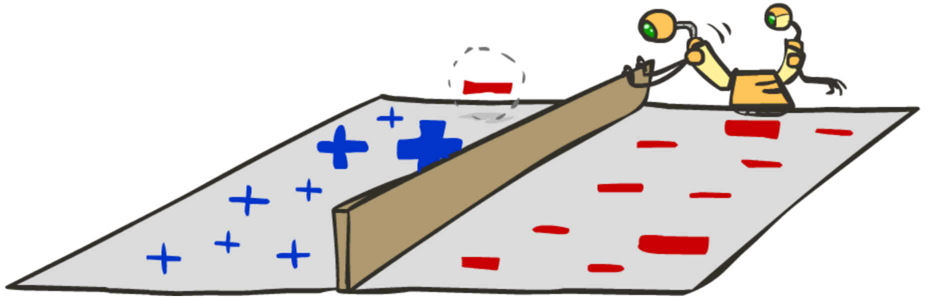
- Start with weights = 0
- For each training instance:
  - Classify with current weights



  - If correct (i.e., y=y*), no change!



  - If wrong: adjust the weight vector



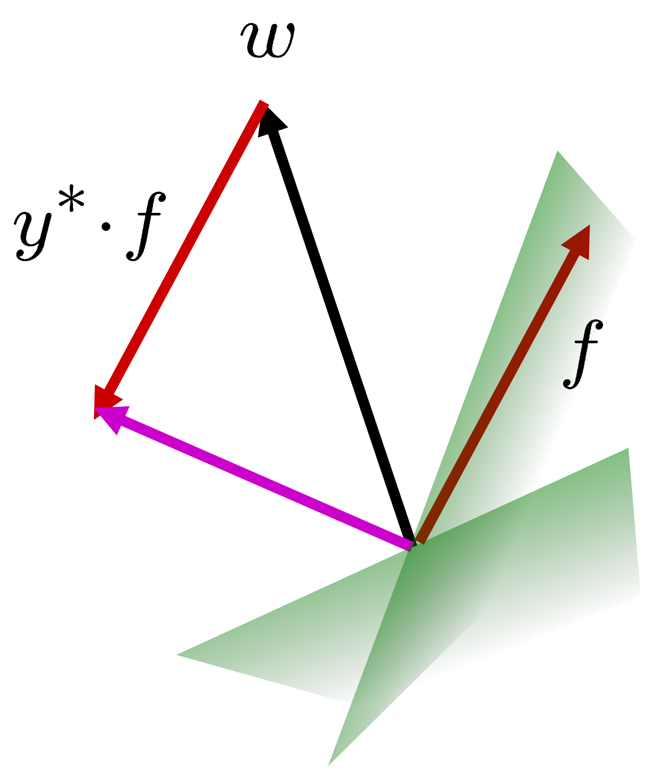
Learning: Binary Perceptron

- Start with weights = 0
- For each training instance:
  - Classify with current weights

$$y = \begin{cases} +1 & \text{if } \; w \cdot f(x) \geq 0 \\ -1 & \text{if } \; w \cdot f(x) < 0 \end{cases}$$

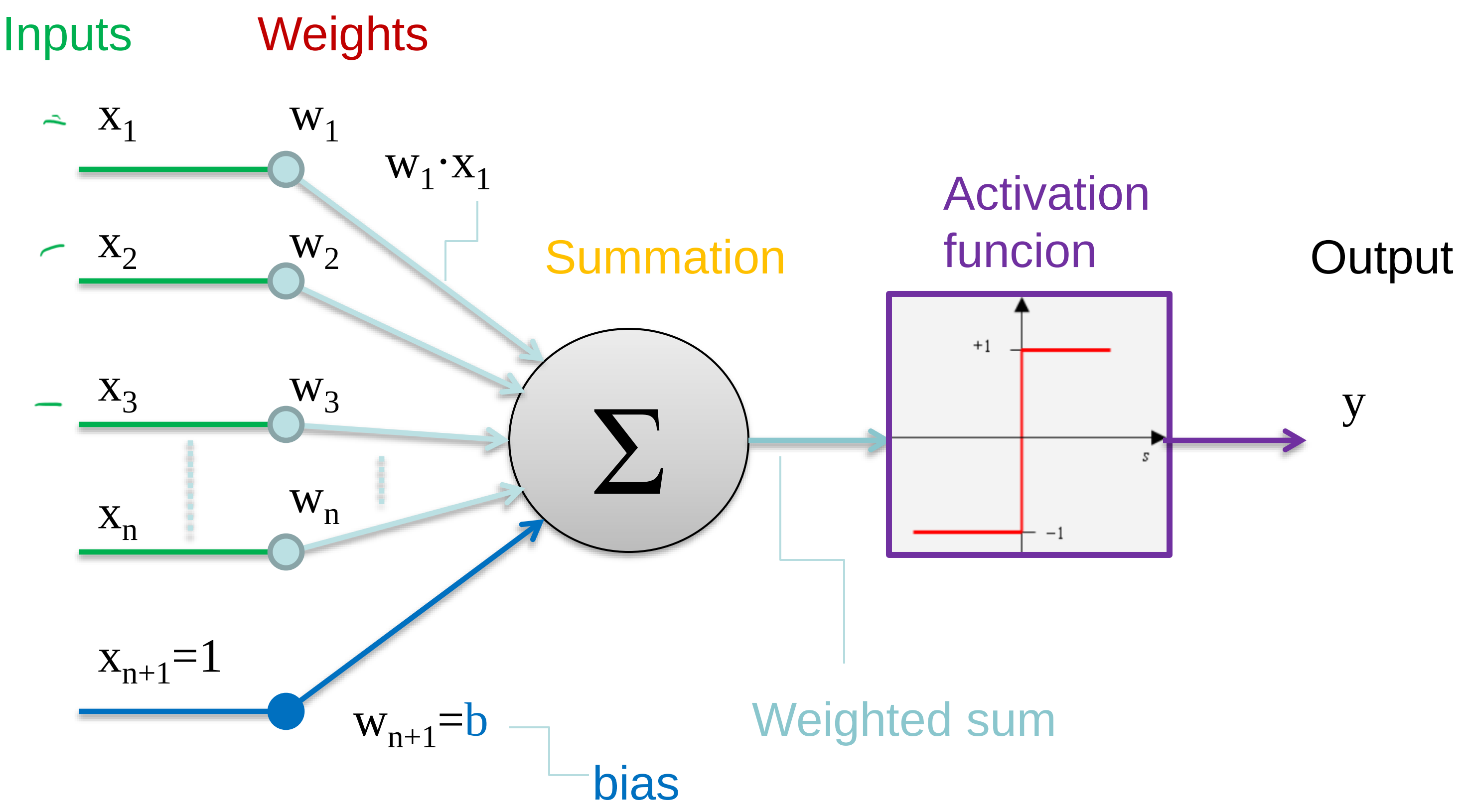
  - If correct (i.e., y=y*), no change!
  - If wrong: adjust the weight vector by adding or subtracting the feature vector. Subtract if y* is -1.

$$w = w + y^* \cdot f$$



Before: $w\,f$
After: $wf + y^* f\,f$
$f\,f >= 0$

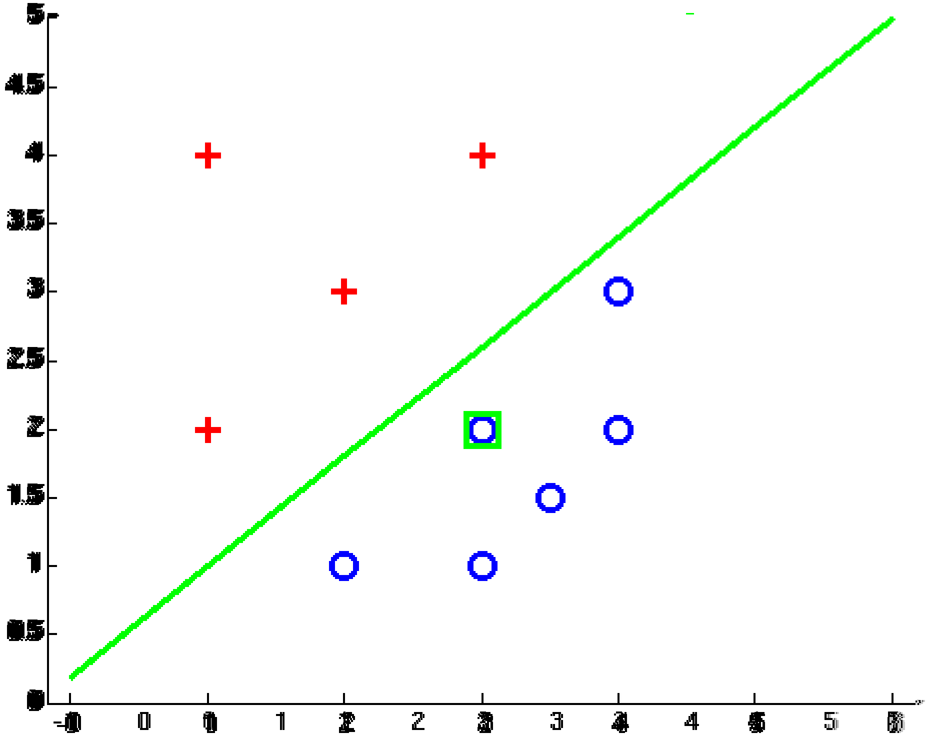
Perceptron



$$y = \mathrm{sgn}\left(\sum_{i=1}^{n+1} w_i\ x_i\right) = \mathrm{sgn}\left(\sum_{i=1}^{n} w_i\ x_i + w_{n+1}\right) = \mathrm{sgn}(\mathbf{x}^T\mathbf{w} + b)$$

Examples: Perceptron

- Separable Case



Multiclass Decision Rule

- If we have multiple classes:
  - A weight vector for each class:

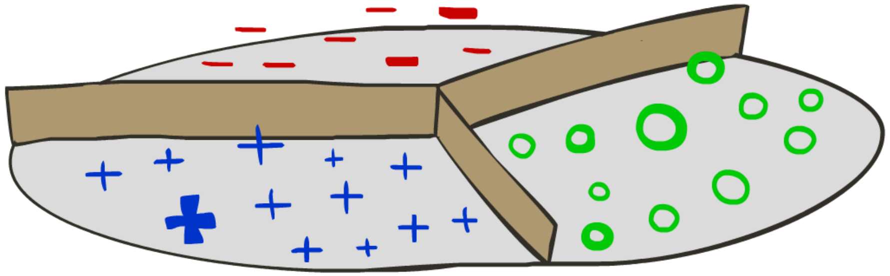
$$w_y$$

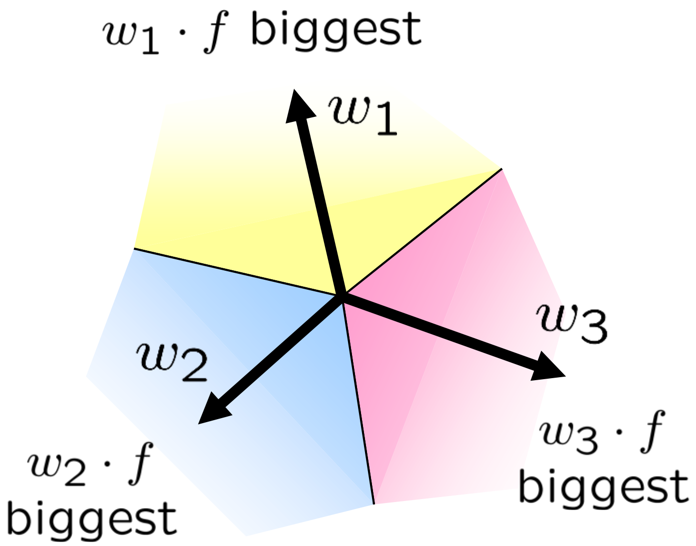
  - Score (activation) of a class y:

$$w_y \cdot f(x)$$

  - Prediction highest score wins

$$y = \arg\max_y \; w_y \cdot f(x)$$





Binary = multiclass where the negative class has weight zero

Learning: Multiclass Perceptron

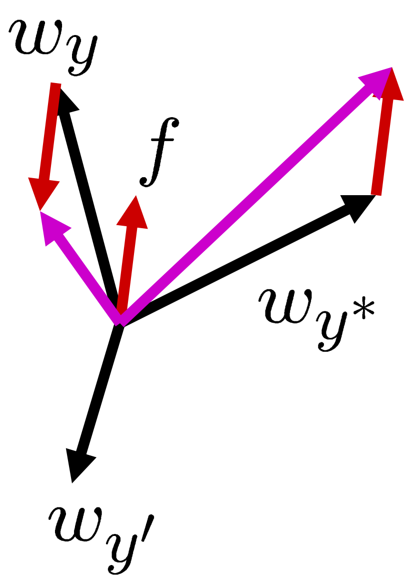
- Start with all weights = 0
- Pick up training examples one by one
- Predict with current weights

$$y = \arg\max_y \ w_y \cdot f(x)$$

- If correct, no change!
- If wrong: lower score of wrong answer, raise score of right answer

$$w_y = w_y - f(x)$$

$$w_{y^*} = w_{y^*} + f(x)$$

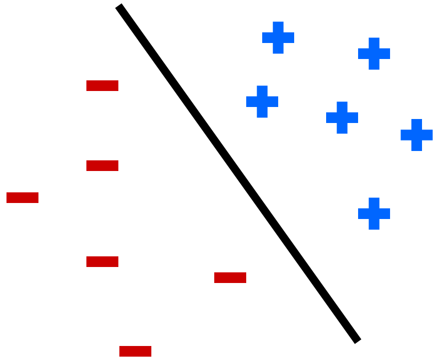


Properties of Perceptrons

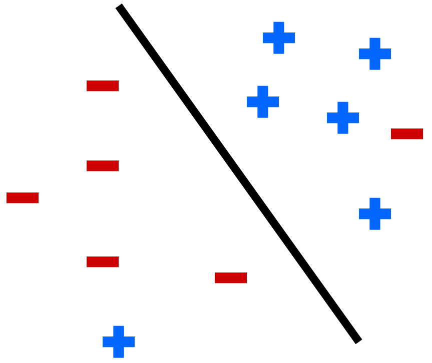
- **Separability**: true if some parameters get the training set perfectly correct

- **Convergence**: if the training is separable, perceptron will eventually converge (binary case)

Separable

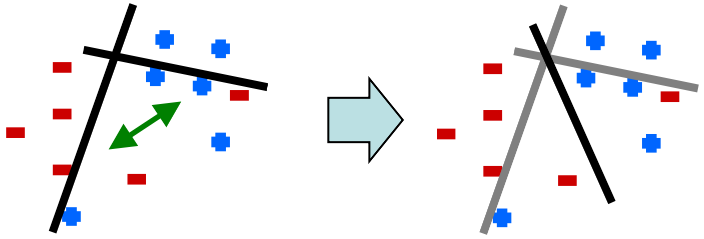


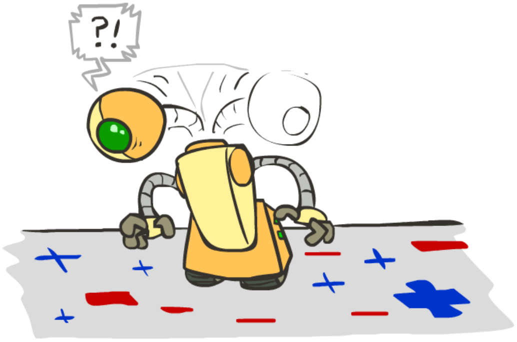
Non-Separable



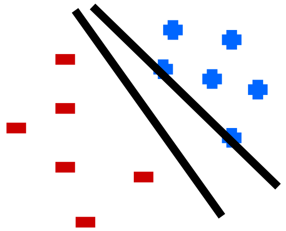
Problems with the Perceptron

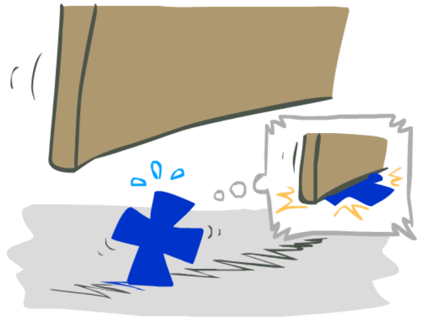
- Noise: if the data isn't separable, weights might thrash
  - Averaging weight vectors over time can help (averaged perceptron)



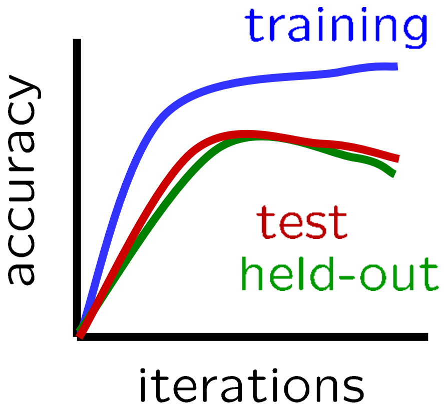


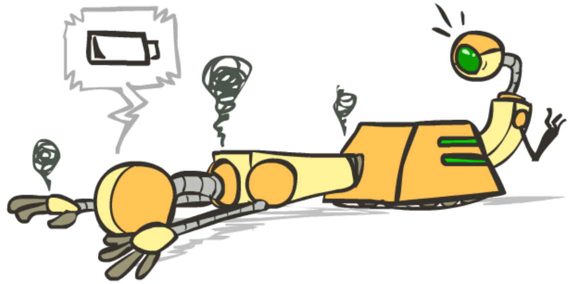
- Mediocre generalization: finds a "barely" separating solution





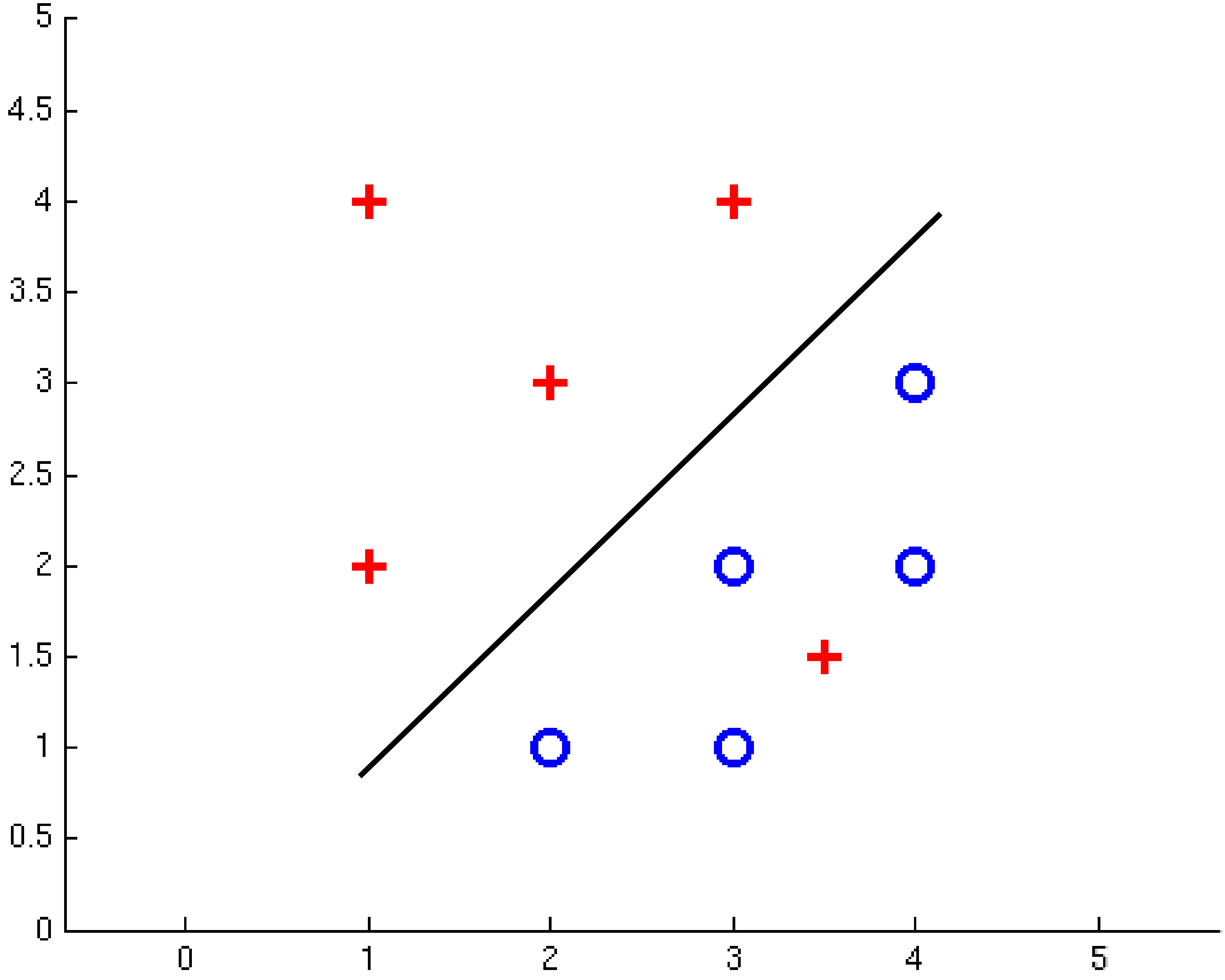
- Overtraining: test / held-out accuracy usually rises, then falls
  - Overtraining is a kind of overfitting



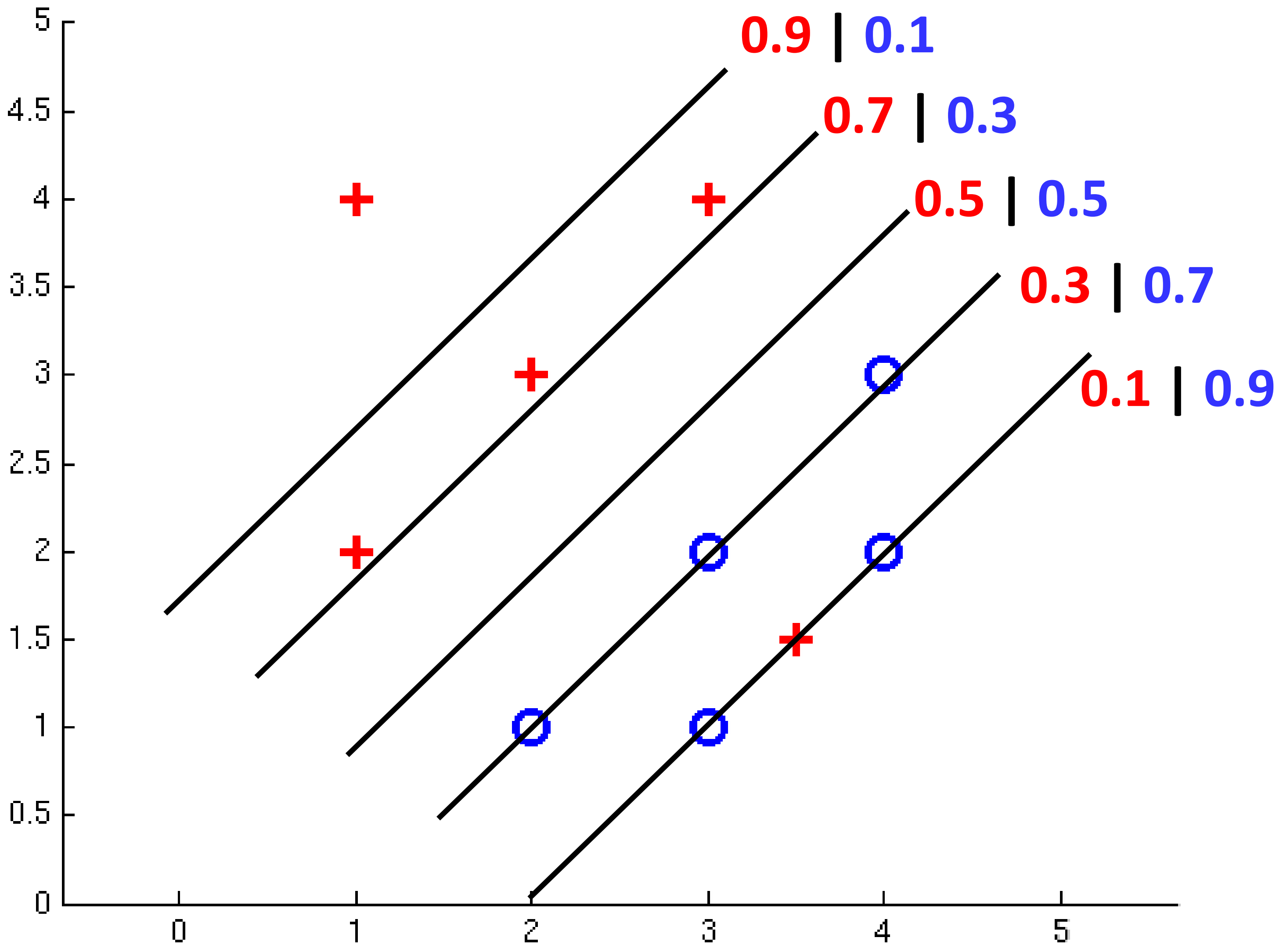


Non-Separable Case: Deterministic Decision

Even the best linear boundary makes at least one mistake



Non-Separable Case: Probabilistic Decision

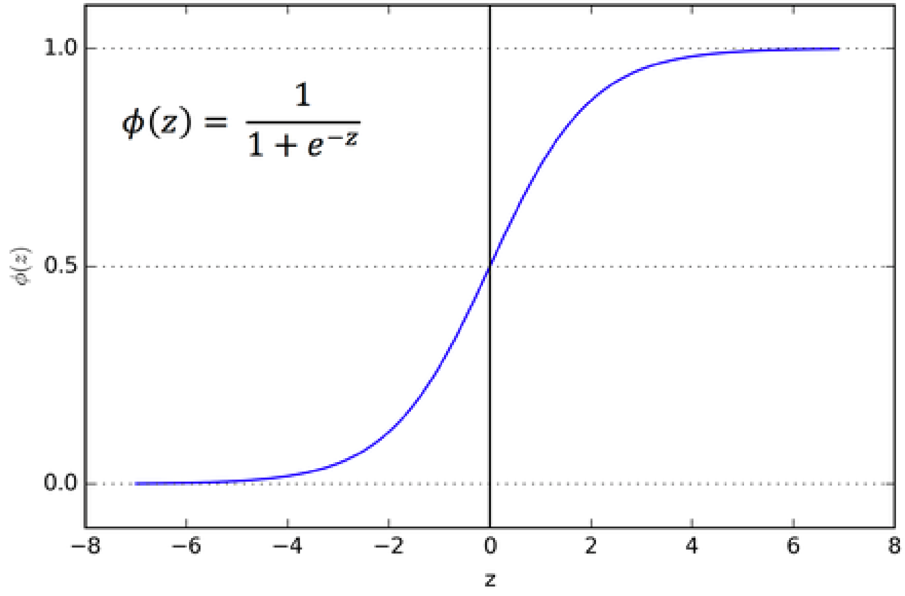


How to get probabilistic decisions?

- Activation: $z = w \cdot f(x)$
- If $z = w \cdot f(x)$ very positive → want probability going to 1
- If $z = w \cdot f(x)$ very negative → want probability going to 0

- Sigmoid function

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



Best w?

- Maximum likelihood estimation:

$$\max_w \quad ll(w) = \max_w \quad \sum_i \log P(y^{(i)}|x^{(i)}; w)$$

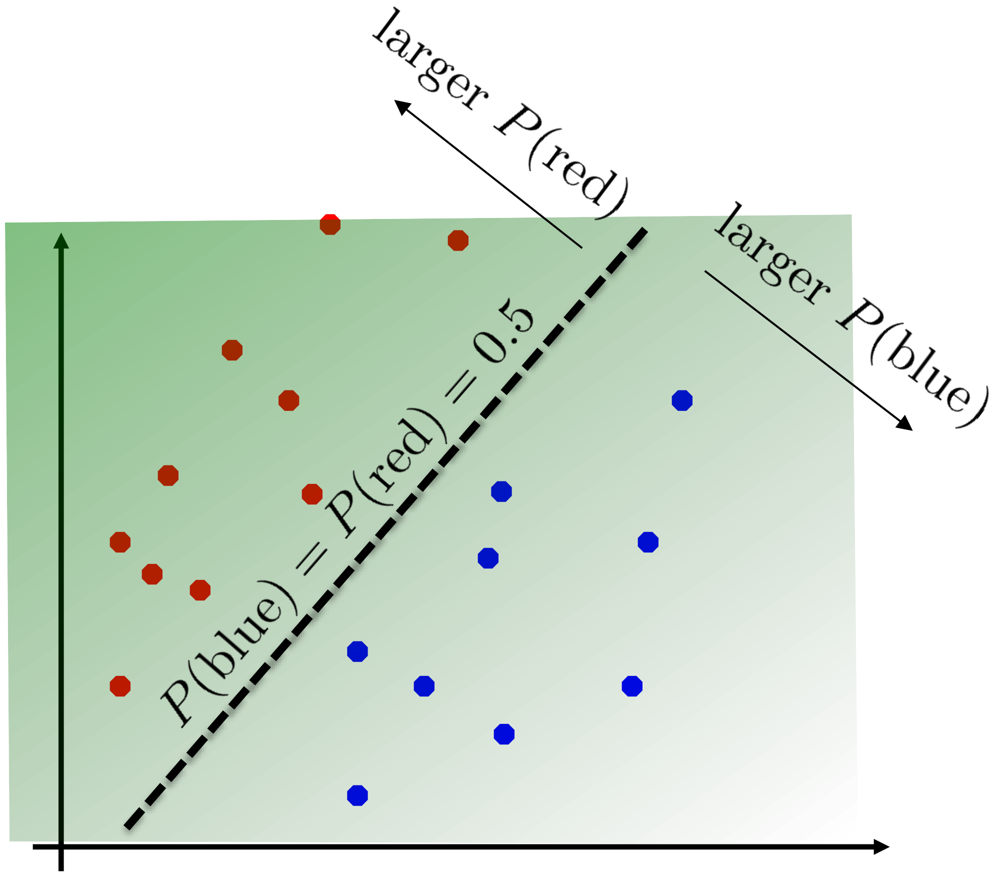
with:

$$P(y^{(i)} = +1|x^{(i)}; w) = \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}$$

$$P(y^{(i)} = -1|x^{(i)}; w) = 1 - \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}$$

= Logistic Regression

A *Probabilistic* Perceptron



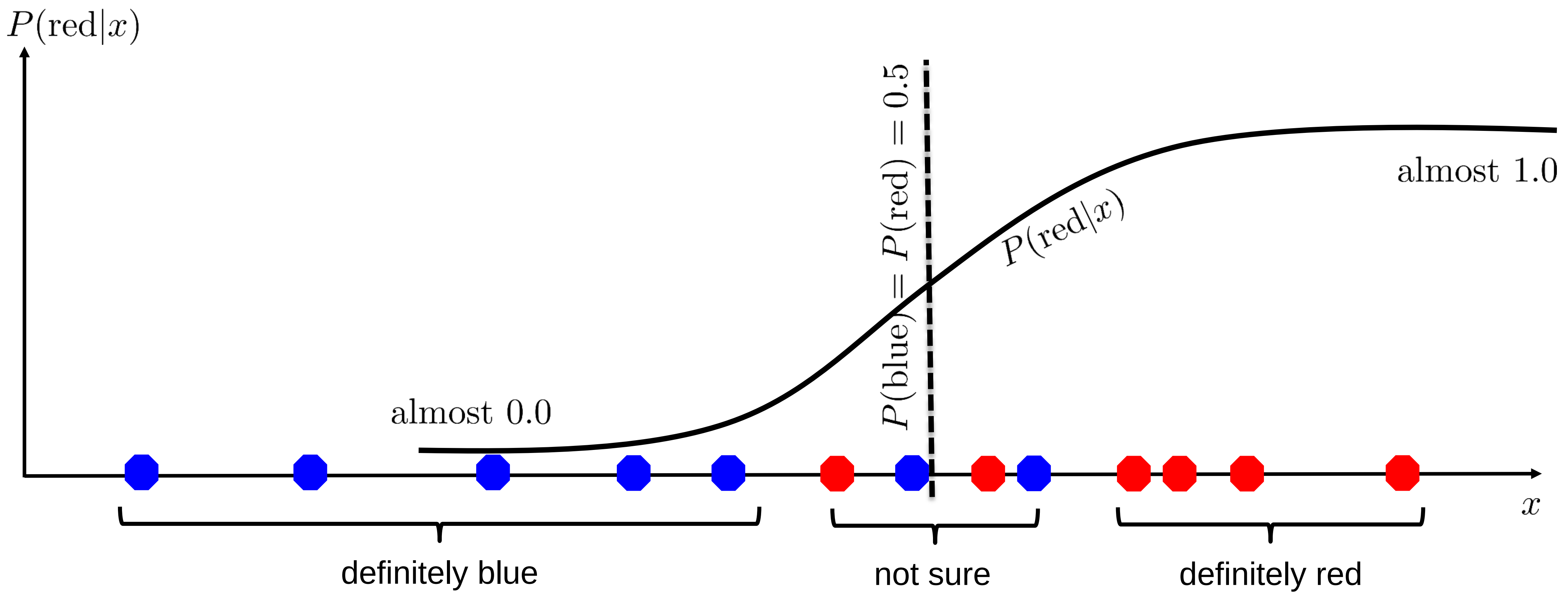
As $w_y \cdot x$ gets bigger, $P(y|x)$ gets bigger

A 1D Example

$$P(\text{red}|x) = \frac{e^{w_{\text{red}} \cdot x}}{e^{w_{\text{red}} \cdot x} + e^{w_{\text{blue}} \cdot x}}$$

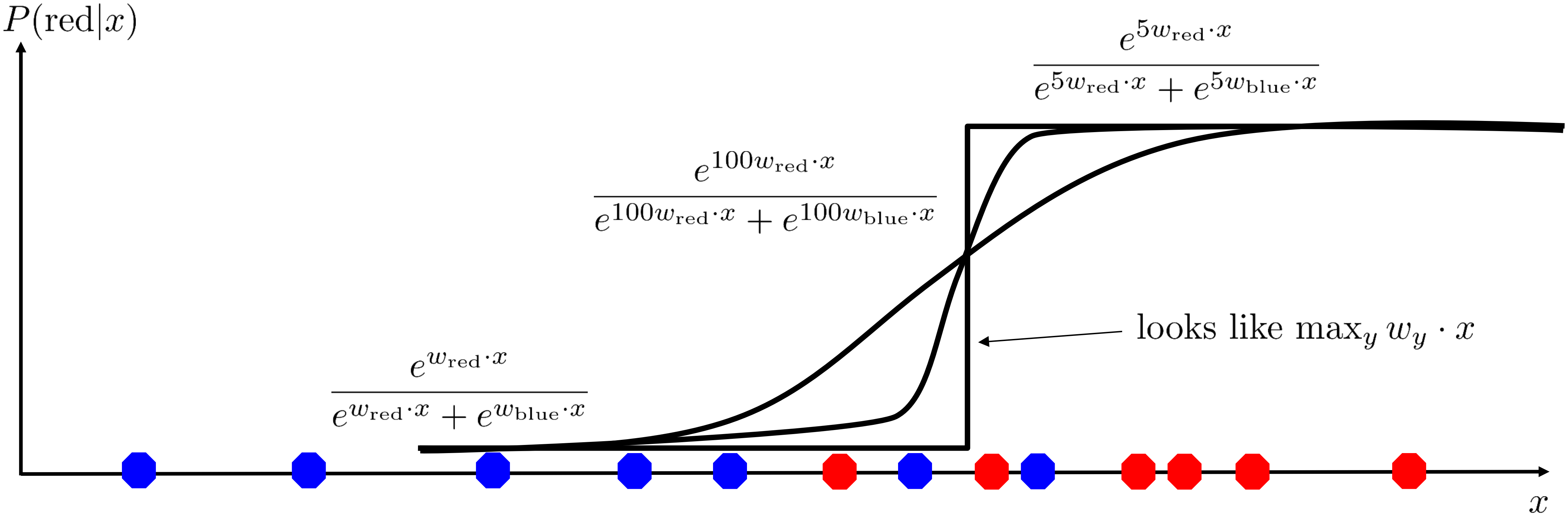
probability increases exponentially as we move away from boundary

normalizer



The *Soft* Max

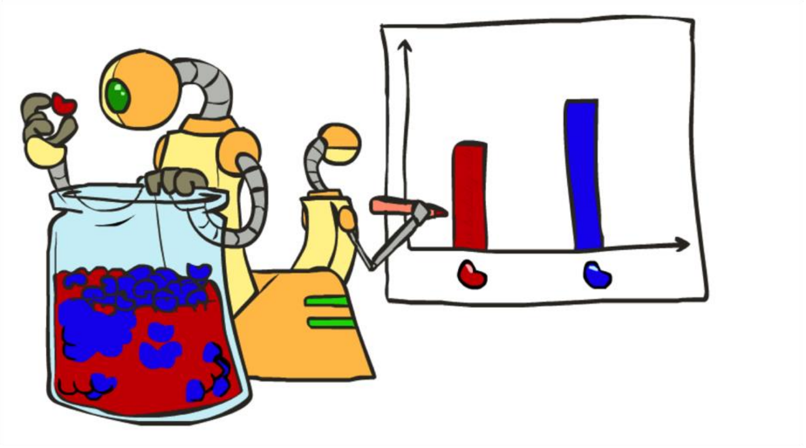
$$P(\text{red}|x) = \frac{e^{w_{\text{red}} \cdot x}}{e^{w_{\text{red}} \cdot x} + e^{w_{\text{blue}} \cdot x}}$$



How to Learn?

- Maximum likelihood estimation

$$\theta_{ML} = \arg\max_{\theta} P(\mathbf{X}|\theta)$$
$$= \arg\max_{\theta} \prod_i P_\theta(X_i)$$



- Maximum *conditional* likelihood estimation

$$\theta^\star = \arg\max_{\theta} P(\mathbf{Y}|\mathbf{X}, \theta)$$
$$= \arg\max_{\theta} \prod_i \underbrace{P_\theta(y_i|x_i)}$$
$$\ell(w) = \prod_i \frac{e^{w_{y_i} \cdot x_i}}{\sum_y e^{w_y \cdot x_i}}$$

$$\ell\ell(w) = \sum_i \log P_w(y_i|x_i)$$
$$= \sum_i w_{y_i} \cdot x_i - \log \sum_y e^{w_y \cdot x_i}$$

Best w?

- Maximum likelihood estimation:

$$\max_{w} \quad ll(w) = \max_{w} \quad \sum_{i} \log P(y^{(i)}|x^{(i)}; w)$$

with:
$$P(y^{(i)}|x^{(i)}; w) = \frac{e^{w_{y^{(i)}} \cdot f(x^{(i)})}}{\sum_{y} e^{w_y \cdot f(x^{(i)})}}$$

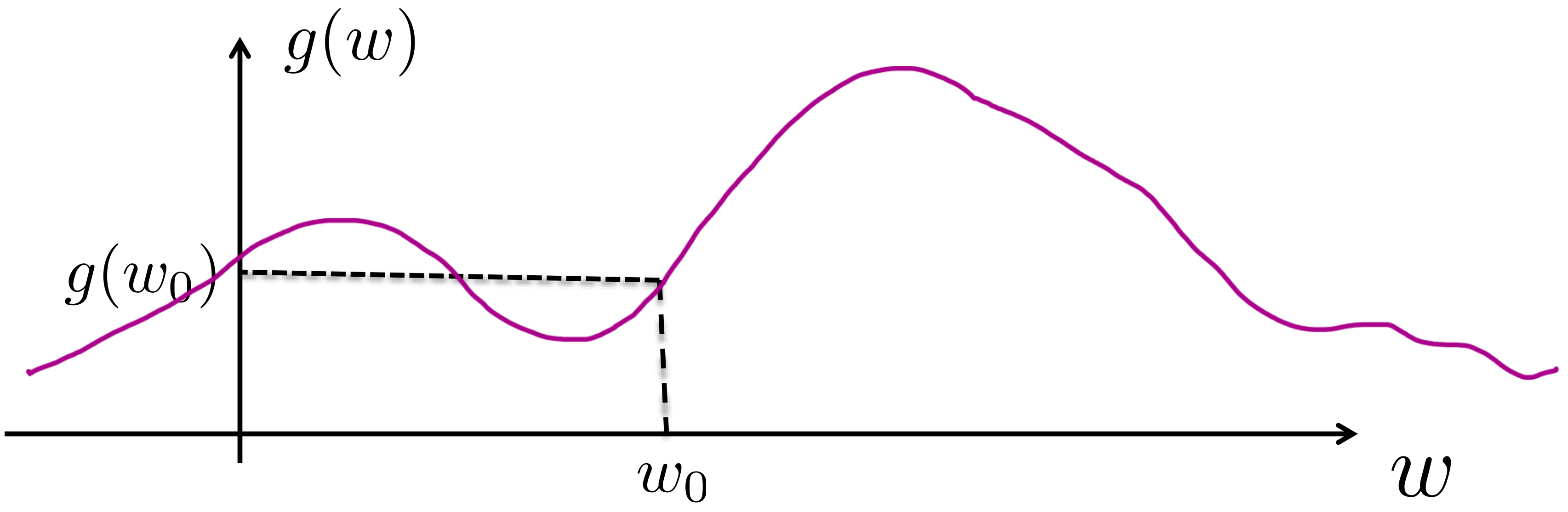
= Multi-Class Logistic Regression

Hill Climbing

- Recall from CSPs lecture: simple, general idea
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

- What's particularly tricky when hill-climbing for multiclass logistic regression?
  - Optimization over a continuous space
    - Infinitely many neighbors!
    - How to do this efficiently?

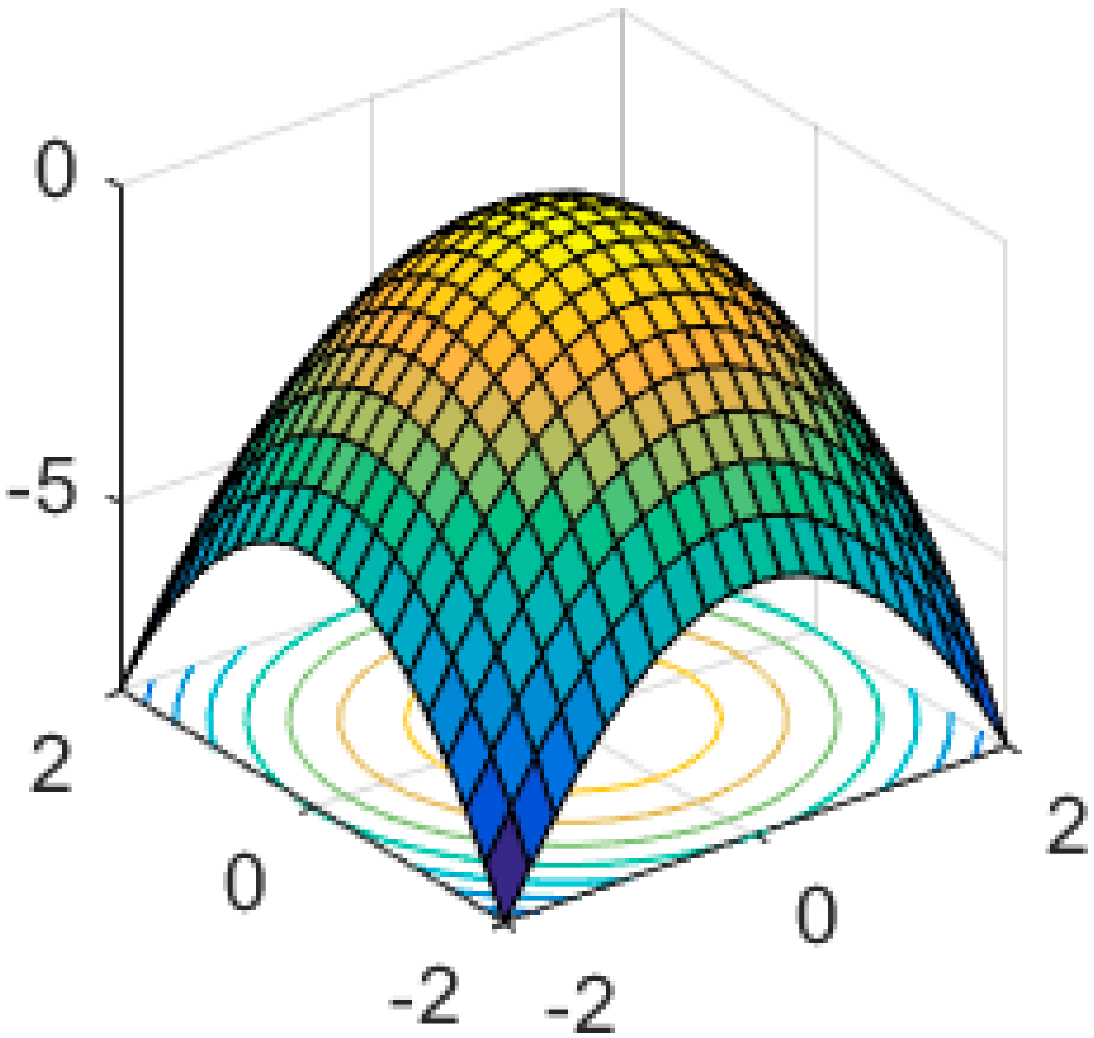
1-D Optimization



- Could evaluate $g(w_0 + h)$ and $g(w_0 - h)$
  - Then step in best direction

- Or, evaluate derivative: $$\frac{\partial g(w_0)}{\partial w} = \lim_{h \to 0} \frac{g(w_0 + h) - g(w_0 - h)}{2h}$$
  - Tells which direction to step into

2-D Optimization



Source: offconvex.org

Gradient Ascent

- Perform update in uphill direction for each coordinate
- The steeper the slope (i.e. the higher the derivative) the bigger the step for that coordinate
- E.g., consider: $g(w_1, w_2)$

  - Updates:

$$w_1 \leftarrow w_1 + \alpha * \frac{\partial g}{\partial w_1}(w_1, w_2)$$

$$w_2 \leftarrow w_2 + \alpha * \frac{\partial g}{\partial w_2}(w_1, w_2)$$

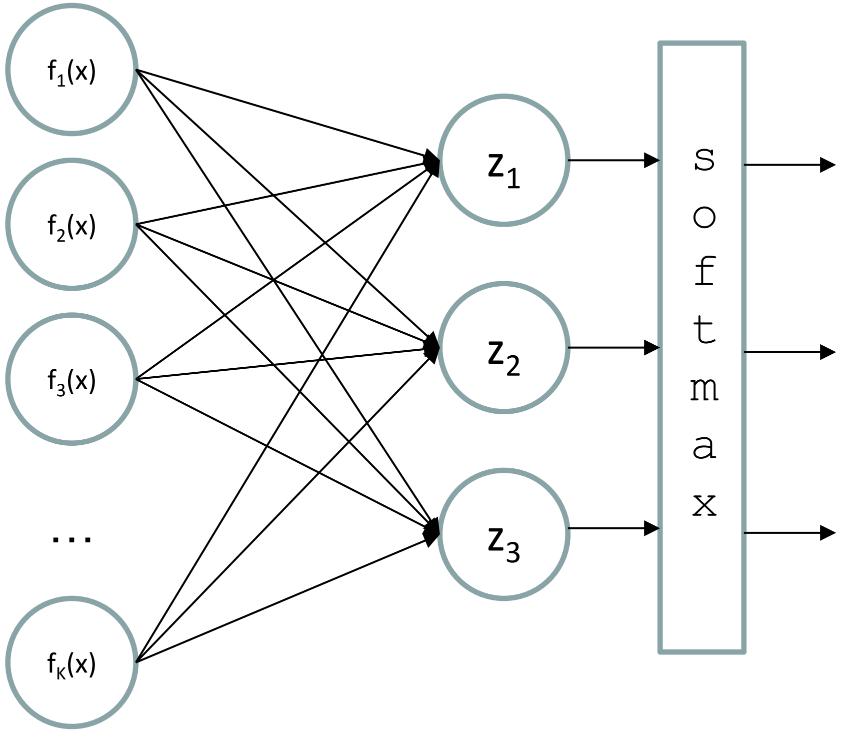
- Updates in vector notation:

$$w \leftarrow w + \alpha * \nabla_w g(w)$$

with: $\nabla_w g(w) = \begin{bmatrix} \frac{\partial g}{\partial w_1}(w) \\ \frac{\partial g}{\partial w_2}(w) \end{bmatrix}$ **= gradient**

Multi-class Logistic Regression

- = special case of neural network



$$P(y_1|x;w) = \frac{e^{z_1}}{e^{z_1}+e^{z_2}+e^{z_3}}$$

$$P(y_2|x;w) = \frac{e^{z_2}}{e^{z_1}+e^{z_2}+e^{z_3}}$$

$$P(y_3|x;w) = \frac{e^{z_3}}{e^{z_1}+e^{z_2}+e^{z_3}}$$

Neural Networks Properties

- Theorem (Universal Function Approximators). A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.

- Practical considerations
  - Can be seen as learning the features
  - Large number of neurons
    - Danger for overfitting
    - (hence early stopping!)

Universal Function Approximation Theorem*

Hornik theorem 1: Whenever the activation function is *bounded and nonconstant*, then, for any finite measure μ, standard multilayer feedforward networks can approximate any function in $L^p(\mu)$ (the space of all functions on R^k such that $\int_{R^k} |f(x)|^p d\mu(x) < \infty$) arbitrarily well, provided that sufficiently many hidden units are available.

Hornik theorem 2: Whenever the activation function is *continuous, bounded and nonconstant*, then, for arbitrary compact subsets $X \subseteq R^k$, standard multilayer feedforward networks can approximate any continuous function on X arbitrarily well with respect to uniform distance, provided that sufficiently many hidden units are available.

- In words: Given any continuous function f(x), if a 2-layer neural network has enough hidden units, then there is a choice of weights that allow it to closely approximate f(x).

Cybenko (1989) "Approximations by superpositions of sigmoidal functions"
Hornik (1991) "Approximation Capabilities of Multilayer Feedforward Networks"
Leshno and Schocken (1991) "Multilayer Feedforward Networks with Non-Polynomial Activation Functions Can Approximate Any Function"

Universal Function Approximation Theorem*

Math. Control Signals Systems (1989) 2: 303–314

Mathematics of Control, Signals, and Systems

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

1. Introduction

A number of diverse application areas are concerned with the representation of general functions of an n-dimensional real variable, $x \in \mathbb{R}^n$, by finite linear combinations of the form

$$\sum_{j=1}^{N} \alpha_j \sigma(y_j^{\mathrm{T}} x + \theta_j), \tag{1}$$

where $y_j \in \mathbb{R}^n$ and $\alpha_j, \theta \in \mathbb{R}$ are fixed. (y^{T} is the transpose of y so that $y^{\mathrm{T}}x$ is the inner product of y and x.) Here the univariate function σ depends heavily on the context of the application. Our major concern is with so-called sigmoidal σ's:

$$\sigma(t) \to \begin{cases} 1 & \text{as} \quad t \to +\infty, \\ 0 & \text{as} \quad t \to -\infty. \end{cases}$$

Such functions arise naturally in neural network theory as the activation function of a neural node (or *unit* as is becoming the preferred term) [L1], [RHM]. The main result of this paper is a demonstration of the fact that sums of the form (1) are dense in the space of continuous functions on the unit cube if σ is any continuous sigmoidal

* Date received: October 21, 1988. Date revised: February 17, 1989. This research was supported in part by NSF Grant DCR-8619103, ONR Contract N000-86-G-0202 and DOE Grant DE-FG02-85ER25001.

† Center for Supercomputing Research and Development and Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois 61801, U.S.A.

Neural Networks, Vol. 4, pp. 251–257, 1991
Printed in the USA.

0893-6080/91 $3.00 + .00

ORIGINAL CONTRIBUTION

Approximation Capabilities of Multilayer Feedforward Networks

KURT HORNIK

Technische Universität Wien, Vienna, Austria

(*Received 30 January 1990; revised and accepted 25 October 1990*)

Abstract—*We show that standard multilayer feedforward networks with as few as a single hidden layer and arbitrary bounded and nonconstant activation function are universal approximators with respect to $L^p(\mu)$ performance criteria, for arbitrary finite input environment measures μ, provided only that sufficiently many hidden units are available. If the activation function is continuous, bounded and nonconstant, then continuous mappings can be learned uniformly over compact input sets. We also give very general conditions ensuring that networks with sufficiently smooth activation functions are capable of arbitrarily accurate approximation to a function and its derivatives.*

Keywords—Multilayer feedforward networks, Activation function, Universal approximation capabilities, Input environment measure, $L^p(\mu)$ approximation, Uniform approximation, Sobolev spaces, Smooth approximation.

1. INTRODUCTION

The approximation capabilities of neural network architectures have recently been investigated by many authors, including Carroll and Dickinson (1989), Cybenko (1989), Funahashi (1989), Gallant and White (1988), Hecht-Nielsen (1989), Hornik, Stinchcombe, and White (1989, 1990), Irie and Miyake (1988), Lapedes and Farber (1988), Stinchcombe and White (1989, 1990). (This list is by no means complete.)

If we think of the network architecture as a rule for computing values at l output units given values at k input units, hence implementing a class of mappings from R^k to R^l, we can ask how well arbitrary mappings from R^k to R^l can be approximated by the network, in particular, if as many hidden units as required for internal representation and computation may be employed.

How to measure the accuracy of approximation depends on how we measure closeness between functions, which in turn varies significantly with the specific problem to be dealt with. In many applications, it is necessary to have the network perform *simultaneously* well on all input samples taken from some compact input set X in R^k. In this case, closeness is measured by the uniform distance between functions on X, that is,

$$\rho_{\mu,X}(f, g) = \sup_{x \in X} |f(x) - g(x)|.$$

In other applications, we think of the inputs as random variables and are interested in the *average performance* where the average is taken with respect to the input environment measure μ, where $\mu(R^k) < \infty$. In this case, closeness is measured by the $L^p(\mu)$ distances

$$\rho_{p,\mu}(f, g) = \left[\int_{R^k} |f(x) - g(x)|^p \, d\mu(x) \right]^{1/p},$$

$1 \le p < \infty$, the most popular choice being $p = 2$, corresponding to mean square error.

Of course, there are many more ways of measuring closeness of functions. In particular, in many applications, it is also necessary that the *derivatives* of the approximating function implemented by the network closely resemble those of the function to be approximated, up to some order. This issue was first taken up in Hornik et al. (1990), who discuss the sources of need of smooth functional approximation in more detail. Typical examples arise in robotics (learning of smooth movements) and signal processing (analysis of chaotic time series); for a recent application to problems of nonparametric inference in statistics and econometrics, see Gallant and White (1989).

All papers establishing certain approximation ca-

Requests for reprints should be sent to Kurt Hornik, Institut für Statistik und Wahrscheinlichkeitstheorie, Technische Universität Wien, Wiedner Hauptstraße 8-10/107, A-1040 Wien, Austria.

MULTILAYER FEEDFORWARD NETWORKS WITH NON-POLYNOMIAL ACTIVATION FUNCTIONS CAN APPROXIMATE ANY FUNCTION

by

Moshe Leshno
Faculty of Management
Tel Aviv University
Tel Aviv, Israel 69978

and

Shimon Schocken
Leonard N. Stern School of Business
New York University
New York, NY 10003

September 1991

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-91-26

Appeared previously as *Working Paper No. 21/91* at The Israel Institute Of Business Research

Cybenko (1989) “Approximations by superpositions of sigmoidal functions”
Hornik (1991) “Approximation Capabilities of Multilayer Feedforward Networks”
Leshno and Schocken (1991) ”Multilayer Feedforward Networks with Non-Polynomial Activation Functions Can Approximate Any Function”

Fun Neural Net Demo Site

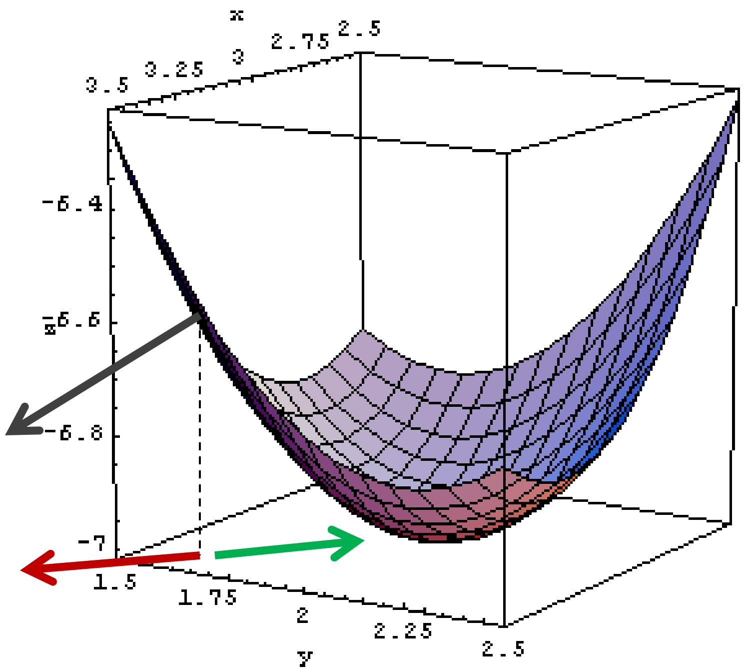
- Demo-site:
  - http://playground.tensorflow.org/

Different approach: Let’s Minimize the Error

- Example inputs are shown to neural network,
- if an error occurs (output and expected value differ),
  the weights are adjusted to reduce the error.
- **The trick is to determine the error and to compute the proportion of the error corresponding to each weight that caused the error.**

$$W \leftarrow W - \alpha \frac{\partial E}{\partial W}$$

Gradient Descent



Loss / Error functions

- Mean Squared Error – regression

$$E(\underline{d}, \underline{y}) = \frac{1}{2} \Sigma_i (d_i - y_i)^2$$

- Crossentropy – binary classification

$$E(\underline{d}, \underline{y}) = -\Sigma_i d_i \log y_i$$

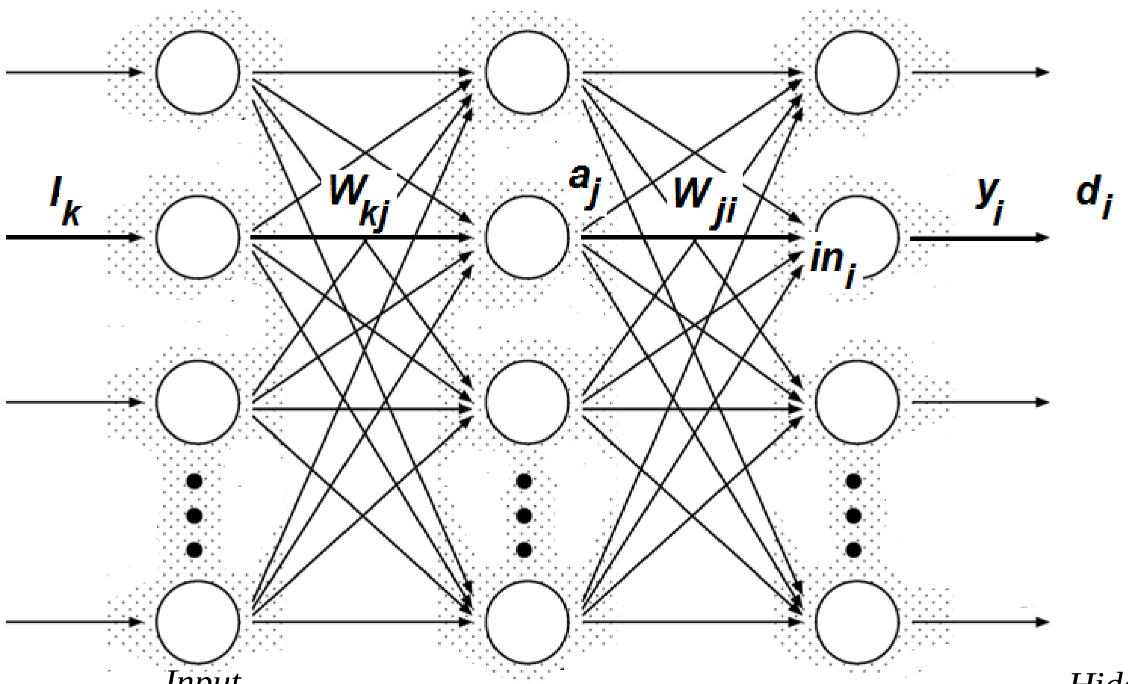
- The aim is to find weights at which the error is minimal

$$w^* = \arg\min_{w} \sum_{i=1}^{N} E(d_i, y_i)$$

, where $\underline{y} = f(\underline{x}, \underline{w})$

Learning steps

- Forward pass
- Backpropagation

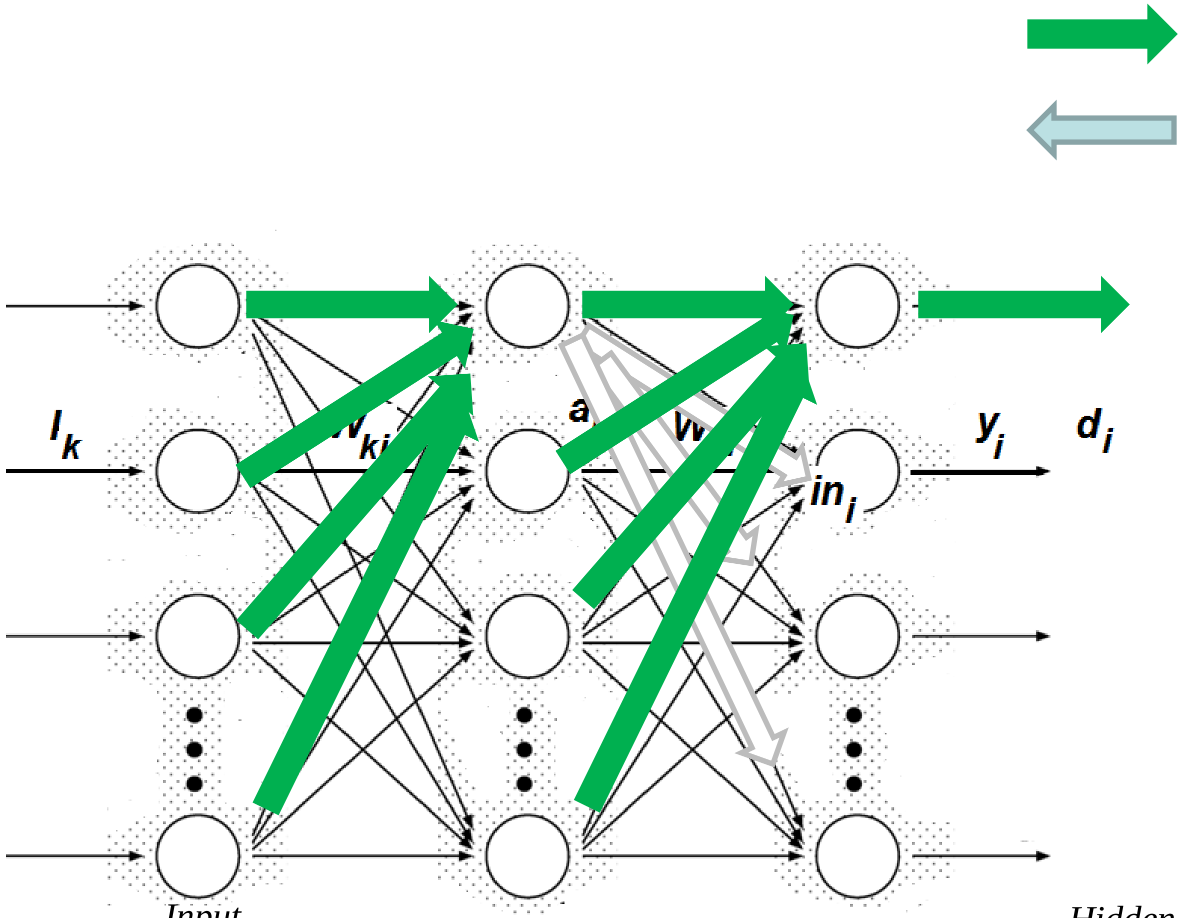


$$a_j(\underline{x}) = g(w_{k0} + \sum_{k=1}^{Input} x_k \cdot w_{kj})$$

$$y_i(\underline{x}) = g^*(w_{j0} + \sum_{j=1}^{Hidden} a_j(\underline{x}) \cdot w_{ji})$$

Forward pass

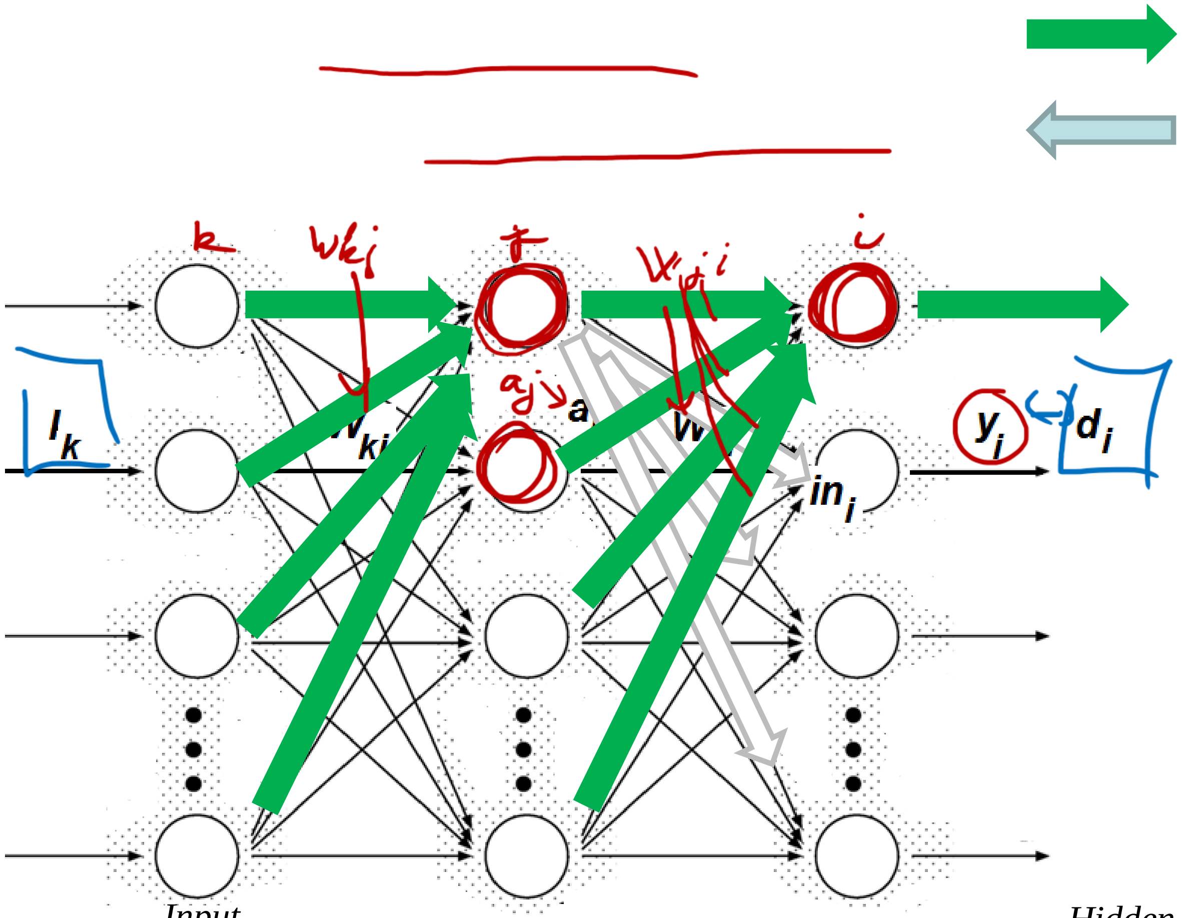
- Forward pass



$$a_j(\underline{x}) = g(w_{k0} + \sum_{k=1}^{Input} x_k \cdot w_{kj}) \qquad y_i(\underline{x}) = g^*(w_{j0} + \sum_{j=1}^{Hidden} a_j(\underline{x}) \cdot w_{ji})$$

Forward pass

- Forward pass

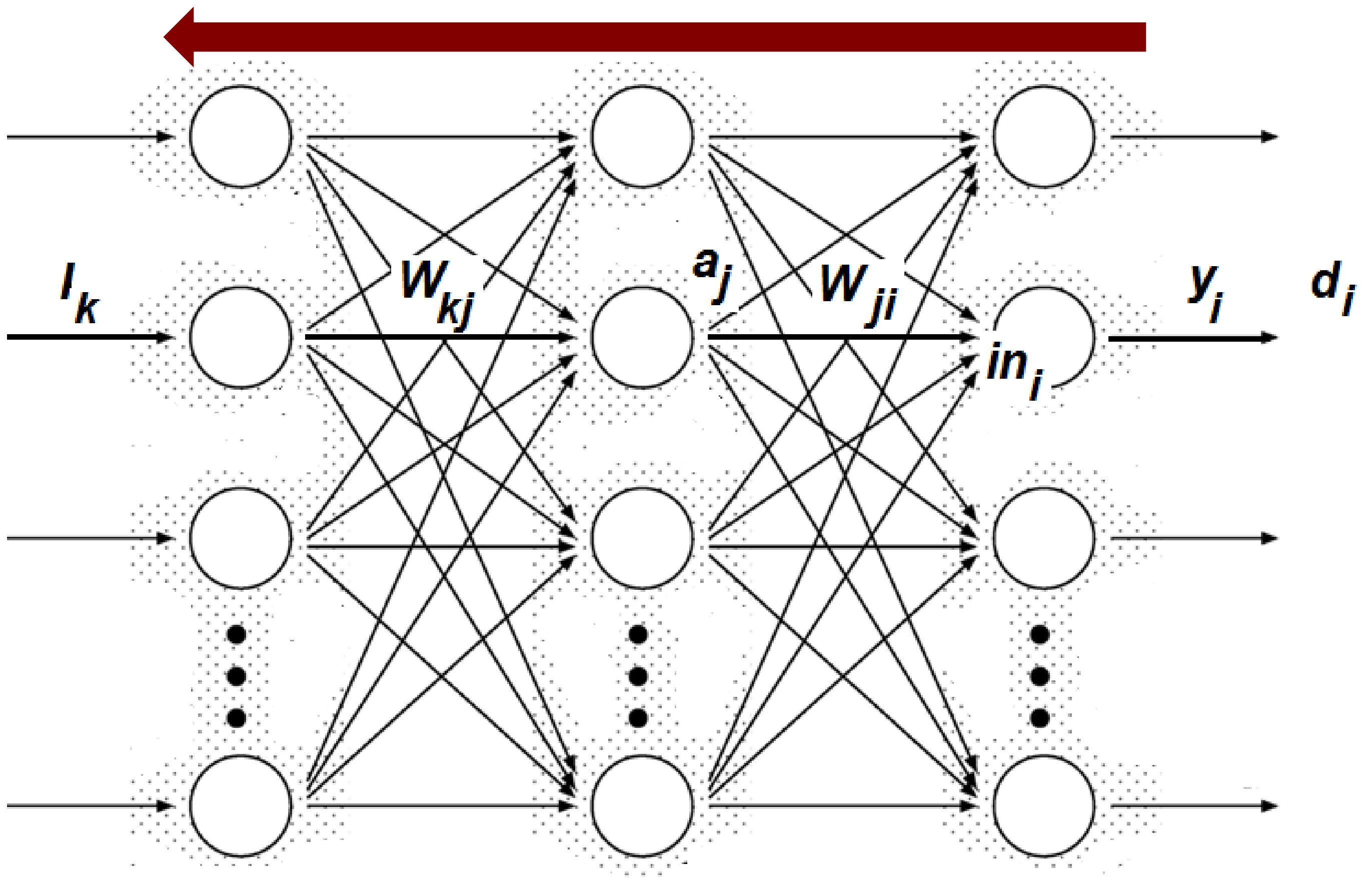


$$a_j(\underline{x}) = g\left(w_{k0} + \sum_{k=1}^{Input} x_k \cdot w_{kj}\right) \qquad y_i(\underline{x}) = g^*\left(w_{j0} + \sum_{j=1}^{Hidden} a_j(\underline{x}) \cdot w_{ji}\right)$$

aktivációs fgv

bias

Backpropagation



$$W_{k,j} \leftarrow W_{k,j} - \alpha \frac{\partial E}{\partial W_{k,j}} \qquad W_{j,i} \leftarrow W_{j,i} - \alpha \frac{\partial E}{\partial W_{j,i}} \qquad E = \frac{1}{2}\sum_i (d_i - y_i)^2$$

Backpropagation

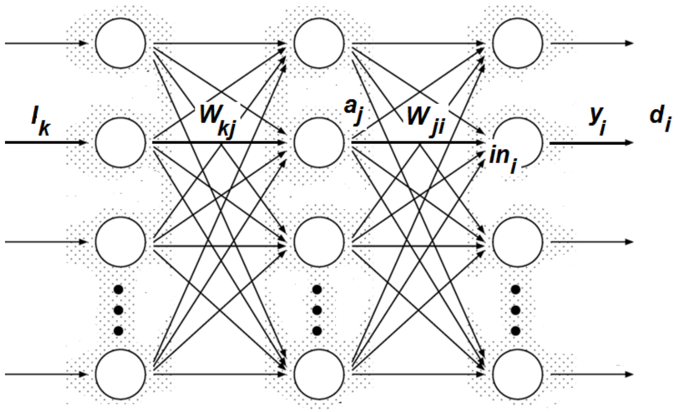
$$E = \frac{1}{2}\sum_i (d_i - y_i)^2 = \frac{1}{2}\sum_i Err_i \qquad W_{j,i} \leftarrow W_{j,i} - \alpha \frac{\partial E}{\partial W_{j,i}}$$

$$E(\mathbf{W}) = \frac{1}{2}\sum_i \left(d_i - g\left(\sum_j W_{j,i}\, a_j\right)\right)^2 = \frac{1}{2}\sum_i \left(d_i - g\left(\sum_j W_{j,i}\, g\left(\sum_k W_{k,j}\, I_k\right)\right)\right)^2$$

$$\frac{\partial E}{\partial W_{j,i}} = -a_j (d_i - y_i) g'\left(\sum_j W_{j,i}\, a_j\right) = -a_j (d_i - y_i) g'(in_i) = -a_j\, \Delta_i$$

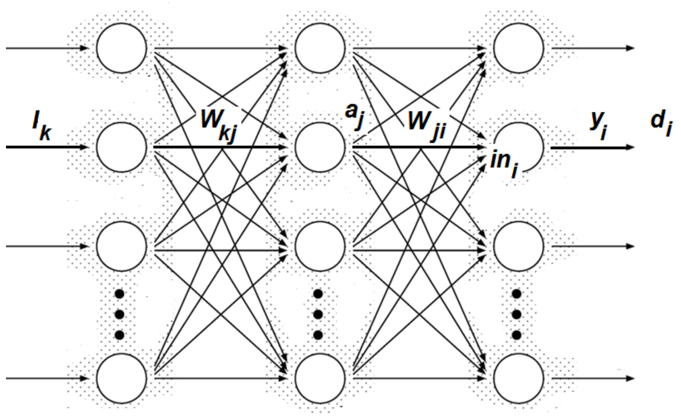
$$W_{j,i} \leftarrow W_{j,i} - \alpha \frac{\partial E}{\partial W_{j,i}} = W_{j,i} + \alpha\, a_j\, Err_i\, g'(in_i)$$

$$\Delta_i = Err_i\, g'(in_i)$$



$$W_{j,i} \leftarrow W_{j,i} + \alpha\, a_j\, \Delta_i$$

Backpropagation



$$\frac{\partial E}{\partial W_{j,i}} = -a_j (d_i - y_i) g'(\sum_j W_{j,i}\, a_j) = -a_j (d_i - y_i) g'(in_i) = -a_j\, \Delta_i$$

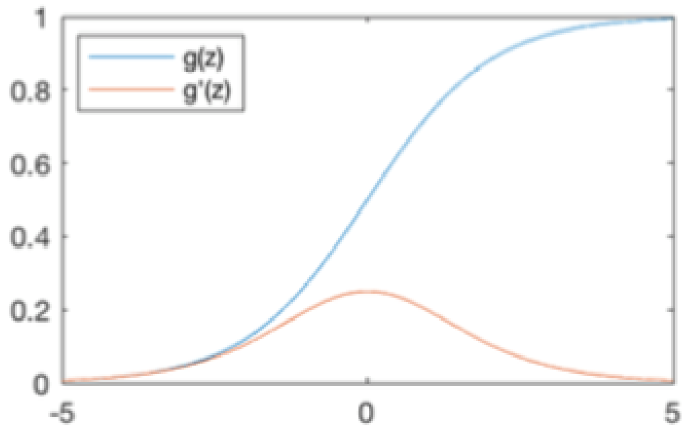
$$\frac{\partial E}{\partial W_{k,j}} = -I_k\, \Delta_j$$

$$W_{k,j} \leftarrow W_{k,j} - \alpha \frac{\partial E}{\partial W_{k,j}}$$

$$W_{k,j} \leftarrow W_{k,j} + \alpha\, I_k\, \Delta_j \qquad \Delta_j = g'(in_j) \sum_i W_{j,i}\, \Delta_i$$

Common Activation Functions

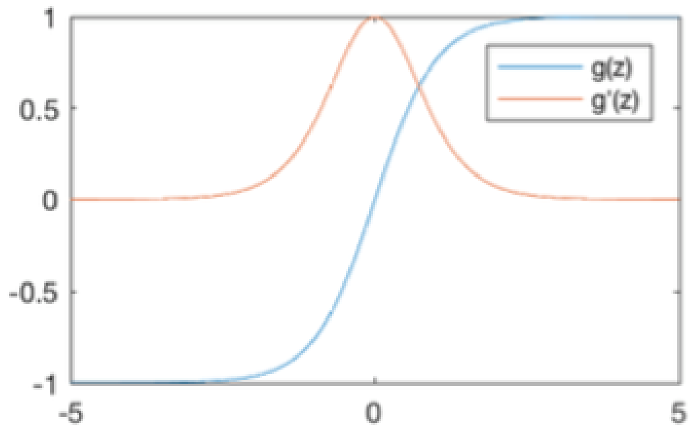
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

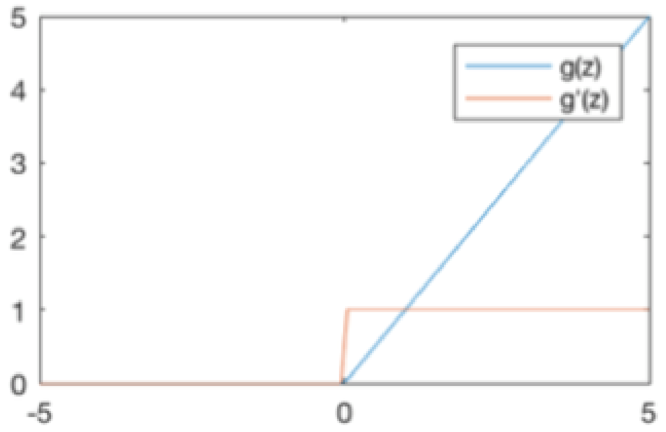
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

[source: MIT 6.S191 introtodeeplearning.com]