

Supervised Machine Learning

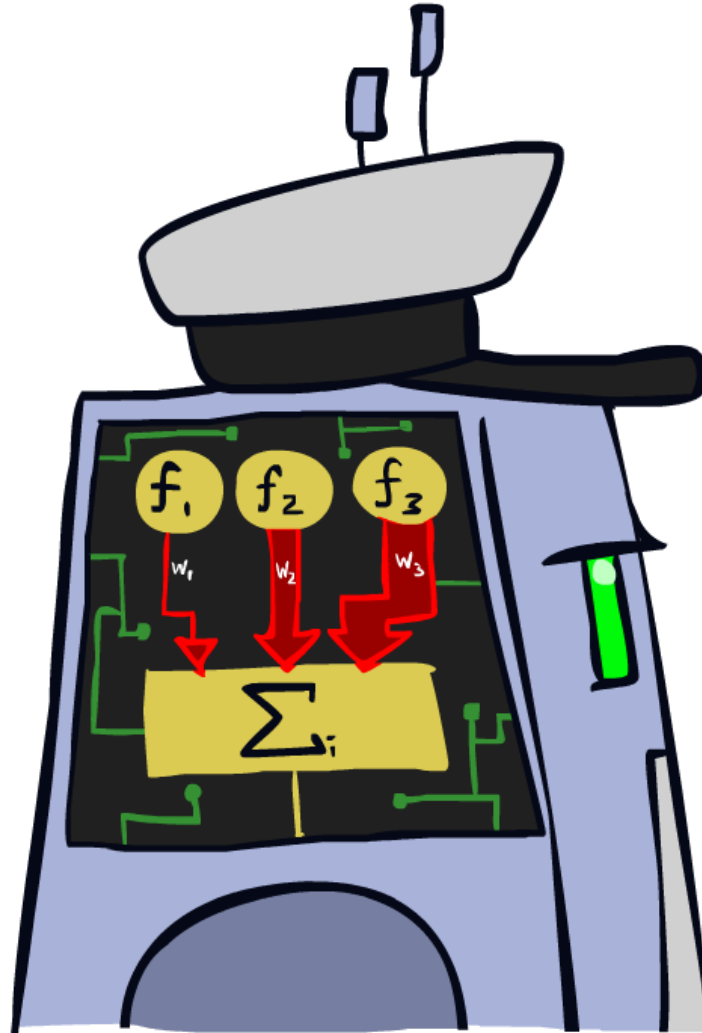


AIMA Chapter 18, 20

More classification methods

- Naive Bayes
- Perceptron / Neural networks
- Decision trees / Random forest
- Support Vector Machines
- Nearest neighbors
- Model ensembles: bagging, boosting, etc.
-

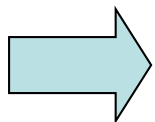
Linear Classifiers



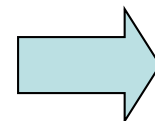
Feature Vectors

 x $f(x)$ y

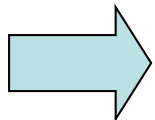
```
Hello,  
  
Do you want free printr  
cartridges? Why pay more  
when you can get them  
ABSOLUTELY FREE! Just
```



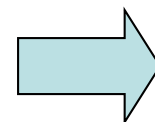
```
# free      : 2  
YOUR_NAME   : 0  
MISPELLED   : 2  
FROM_FRIEND : 0  
...
```



SPAM
or
+



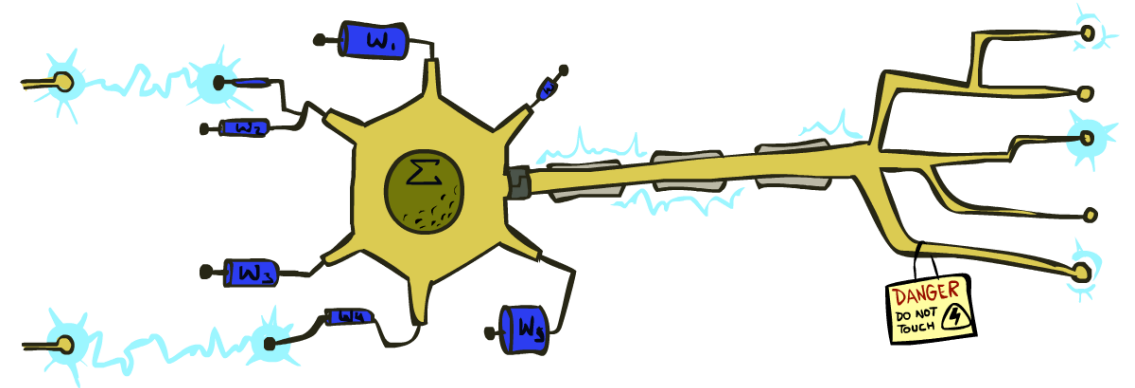
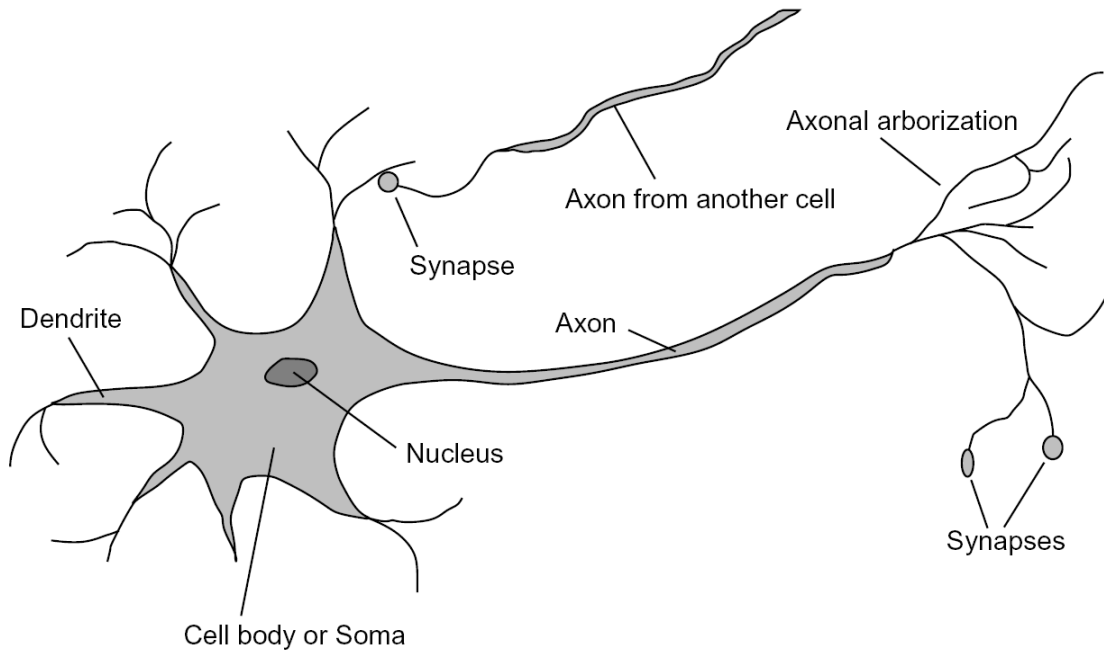
```
PIXEL-7,12 : 1  
PIXEL-7,13 : 0  
...  
NUM_LOOPS  : 1  
...
```



"2"

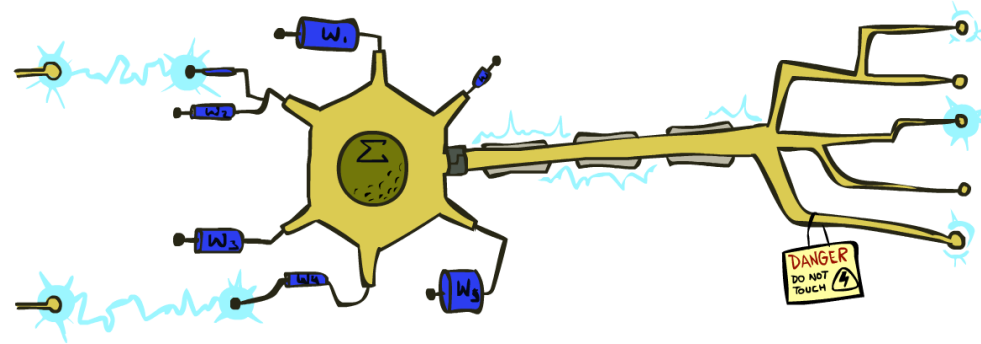
Some (Simplified) Biology

- Very loose inspiration: human neurons



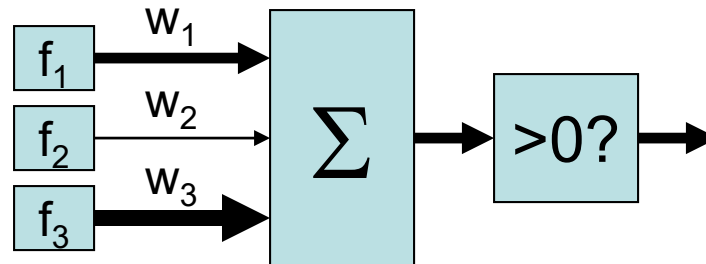
Linear Classifiers

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



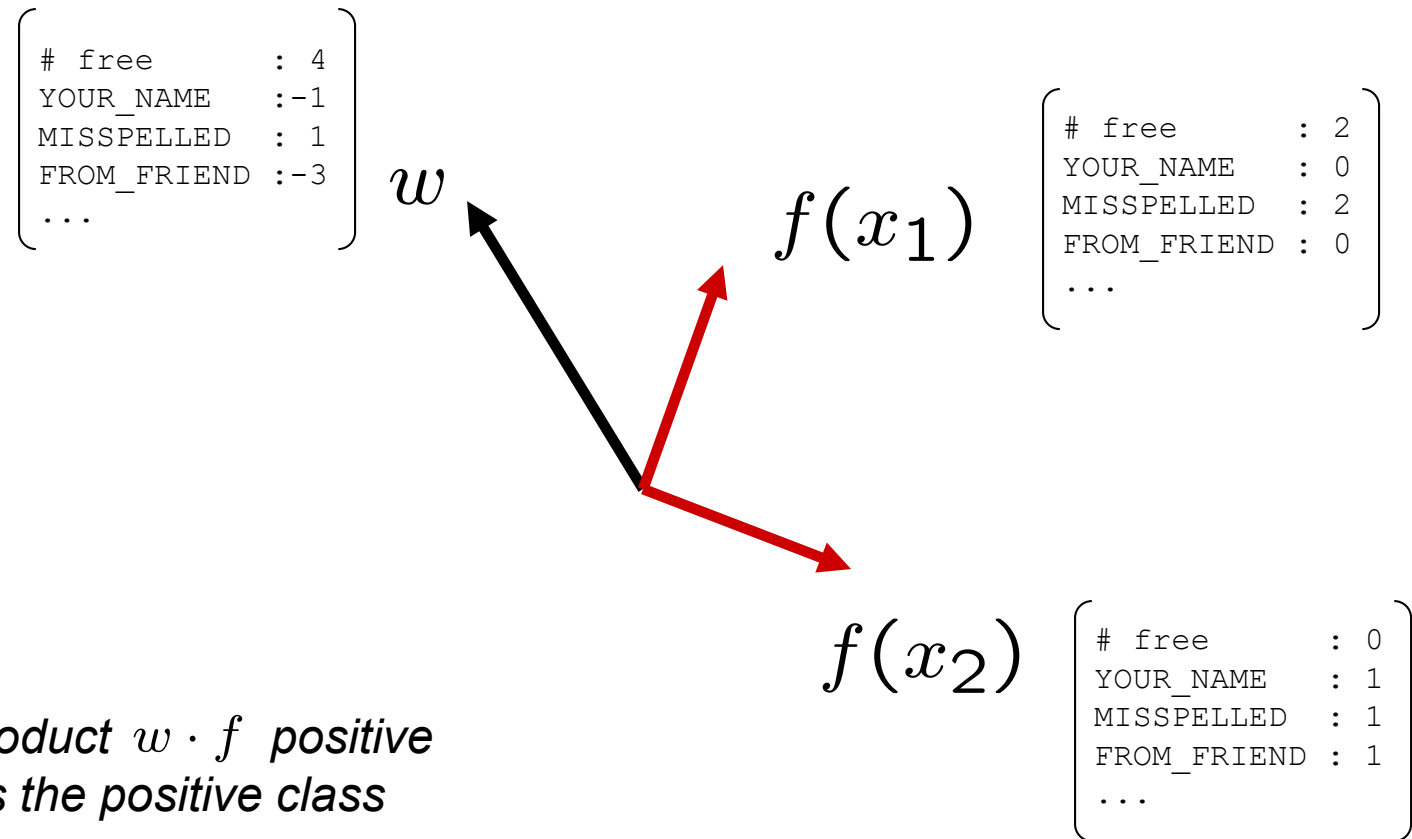
$$\text{activation}_w(x) = \sum_i w_i \cdot f_i(x) = w \cdot f(x)$$

- 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

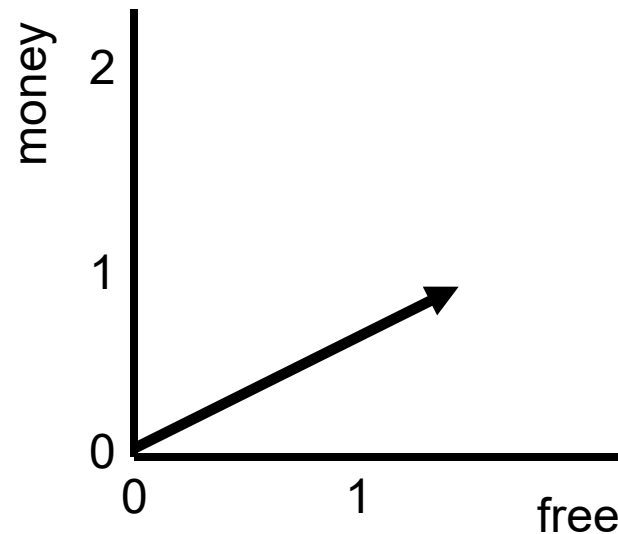
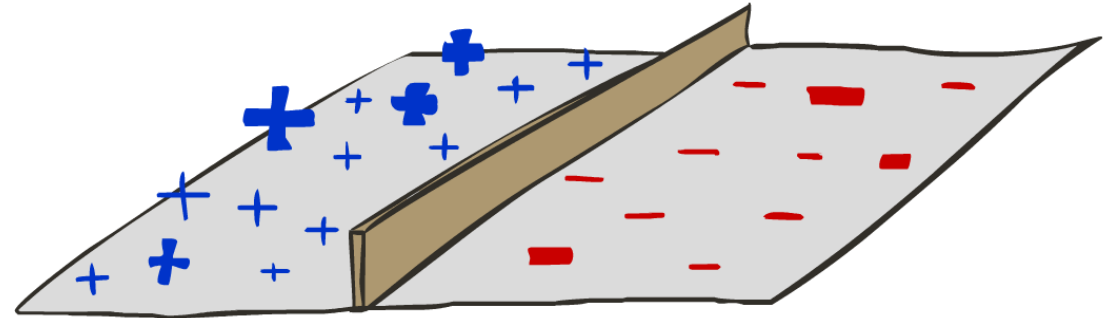


Binary Decision Rule

- 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 |
| ... | | |

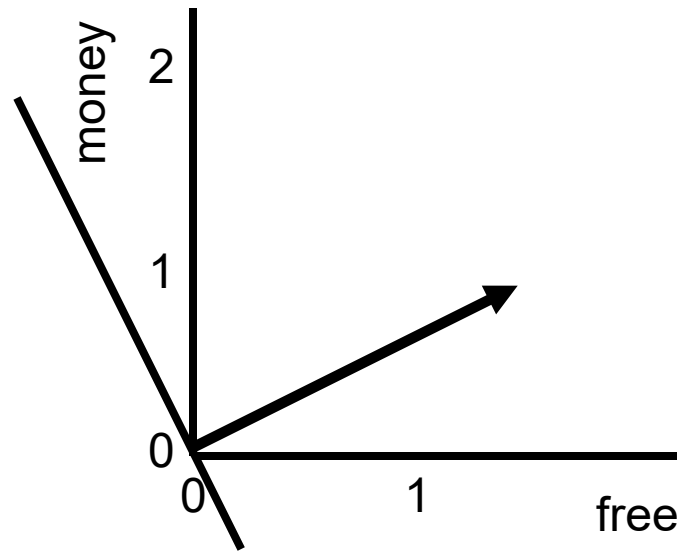
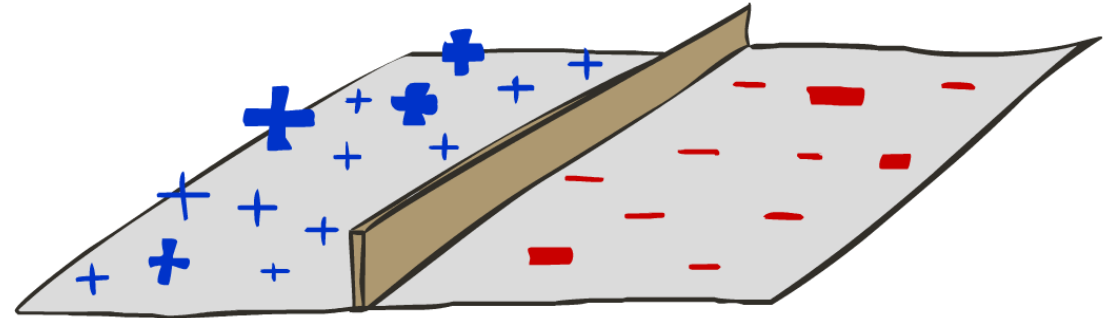


Binary Decision Rule

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

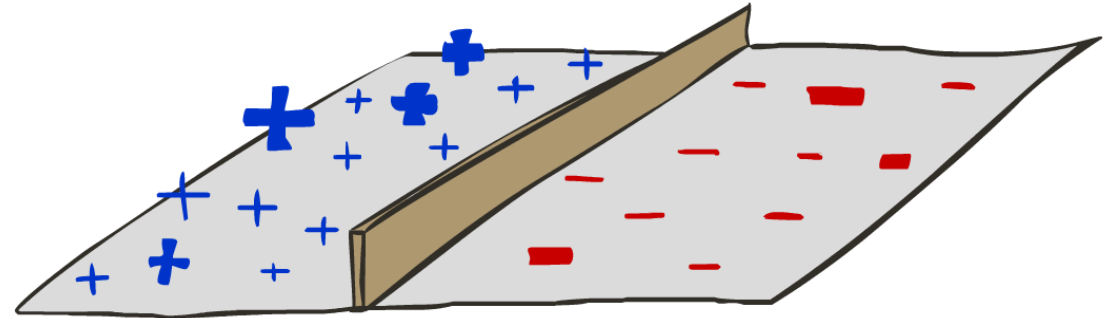
w

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



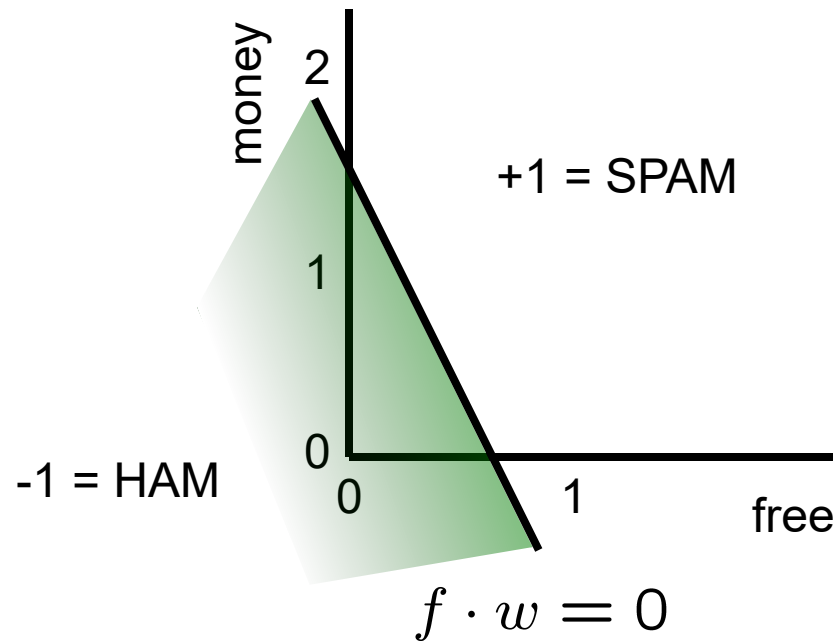
Binary Decision Rule

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

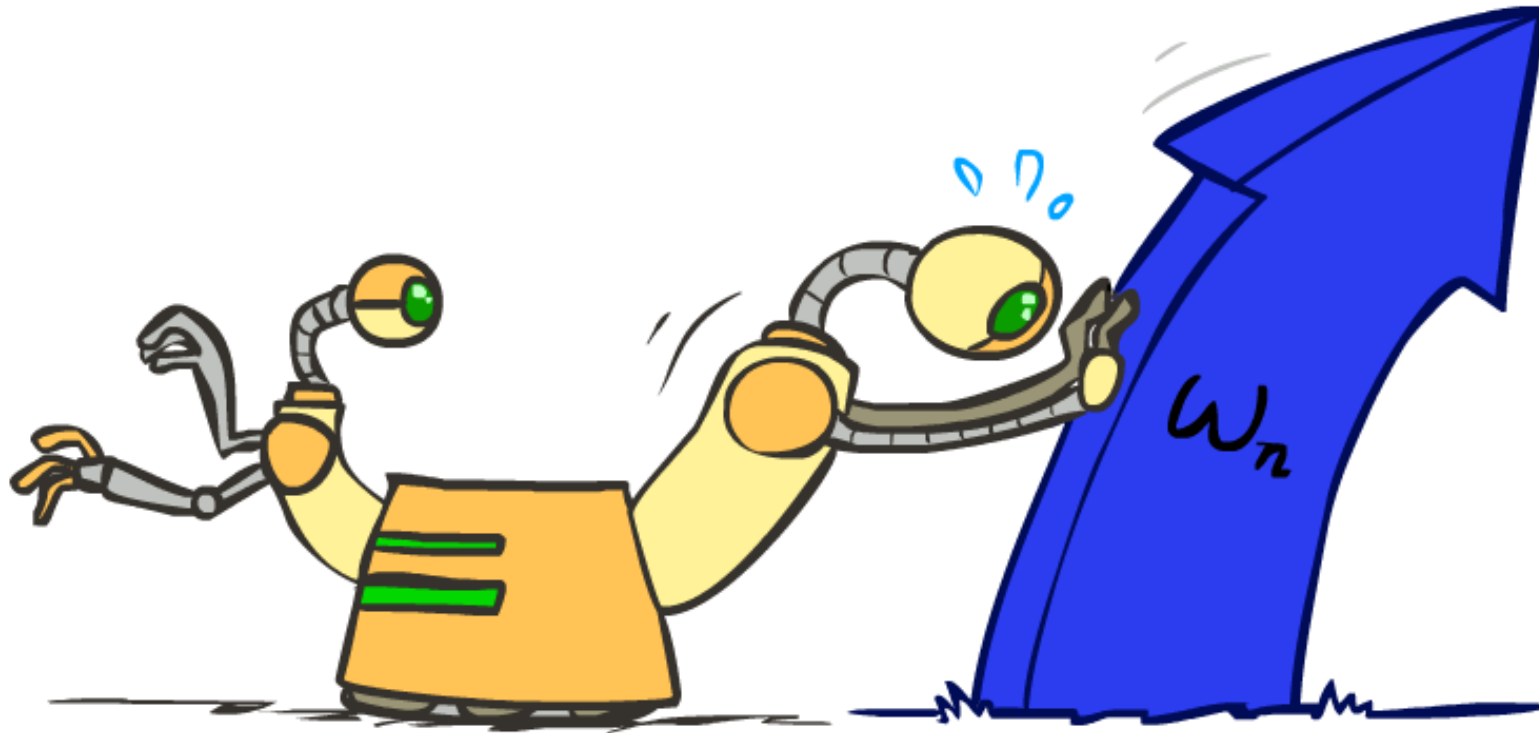


w

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

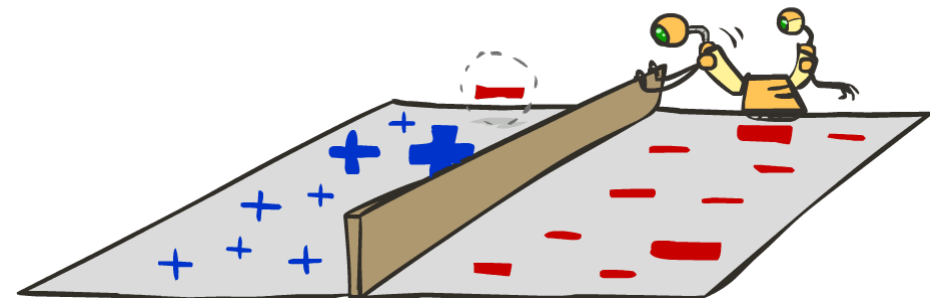
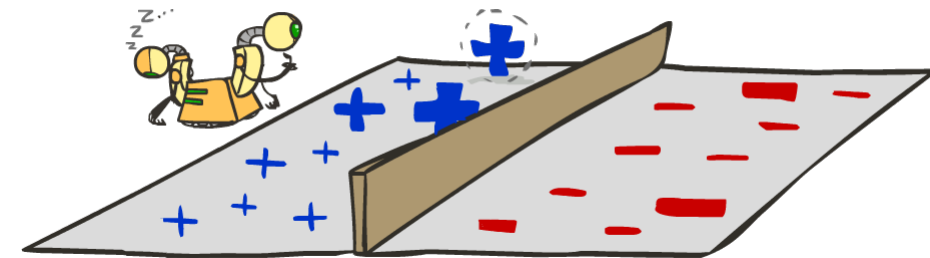
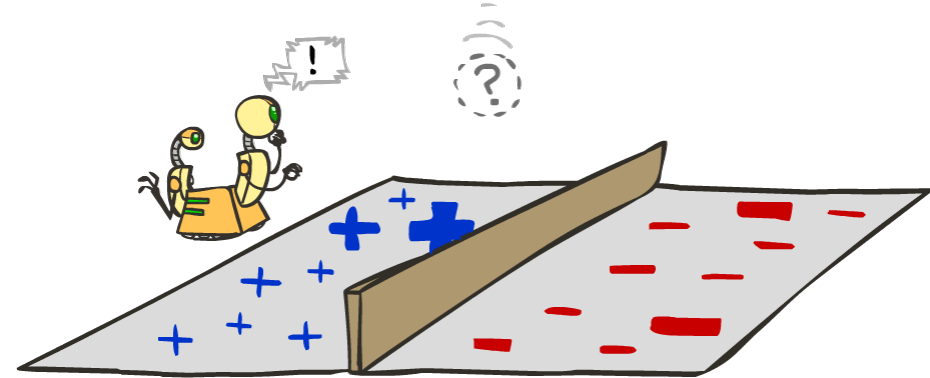


Weight Updates



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



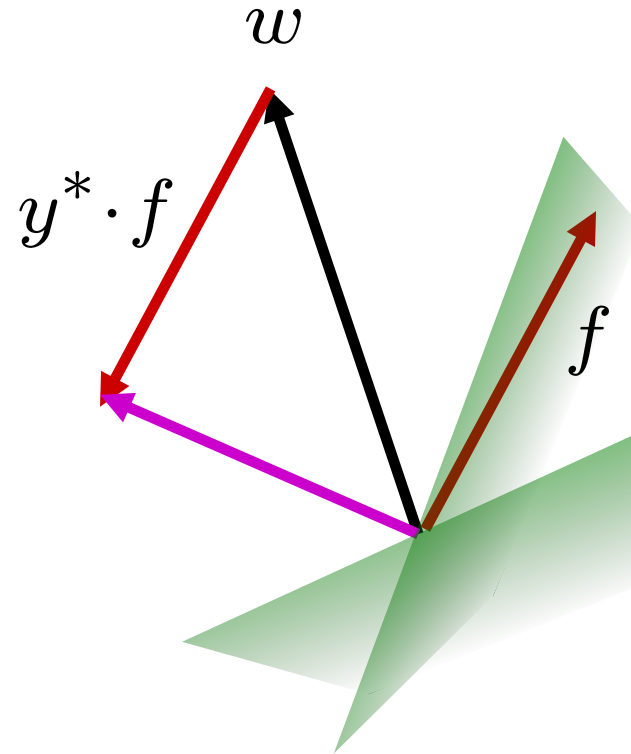
Learning: Binary Perceptron

- 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.

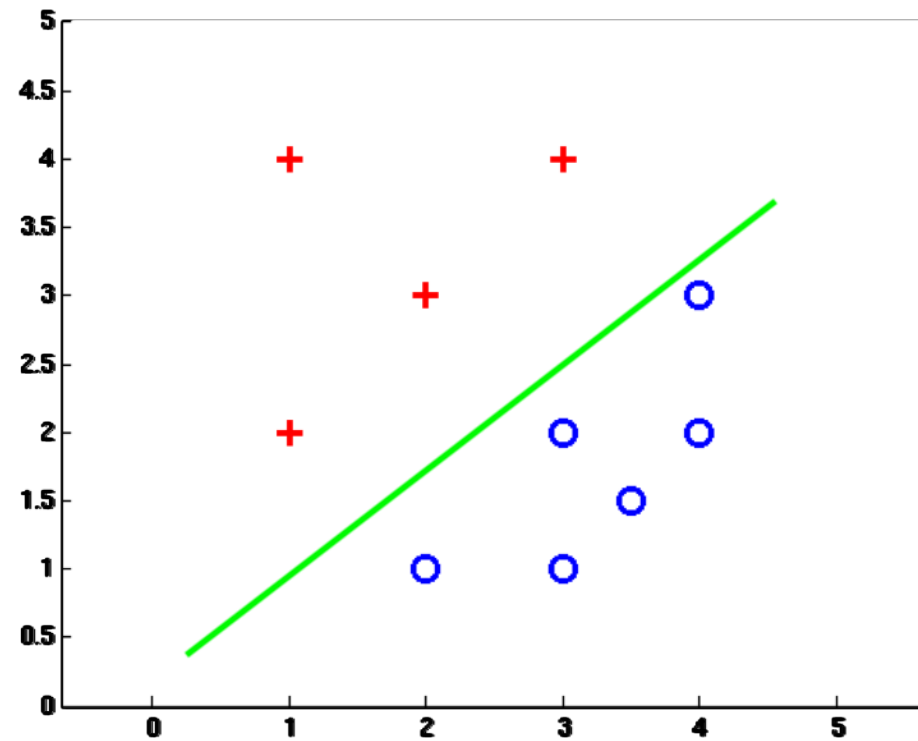
$$w = w + y^* \cdot f$$



Before: $w \cdot f$
After: $w \cdot f + y^* \cdot f \cdot f$
 $f \cdot f \geq 0$

Examples: Perceptron

- Separable Case



Multiclass Decision Rule

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

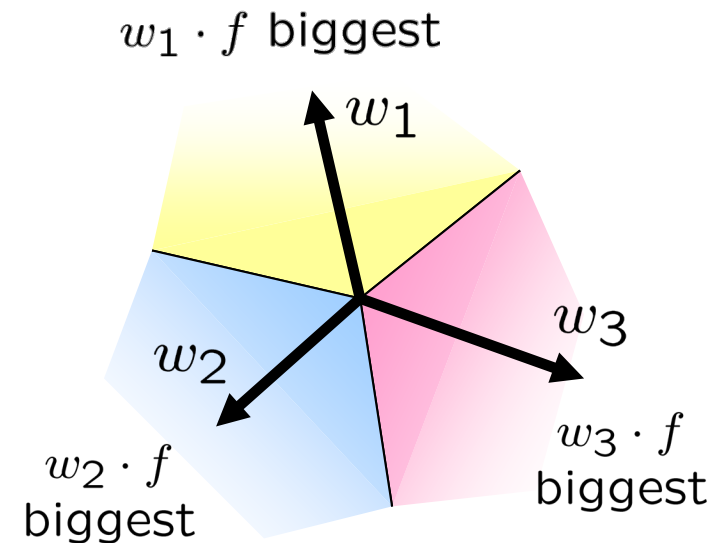
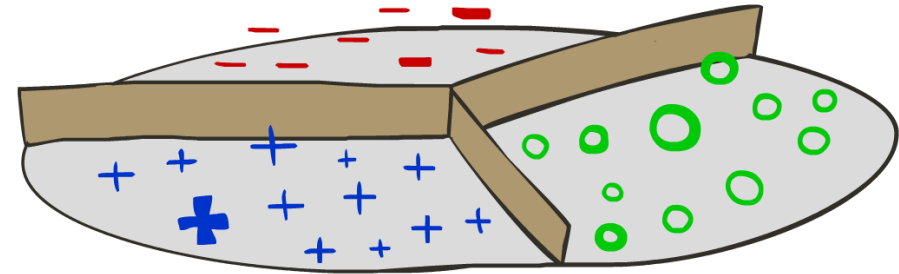
$$w_y$$

- 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

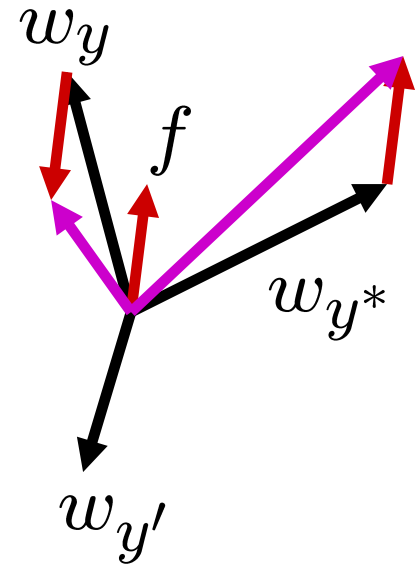
- Start with all weights = 0
- Pick up training examples one by one
- Predict with current weights

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

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

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

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

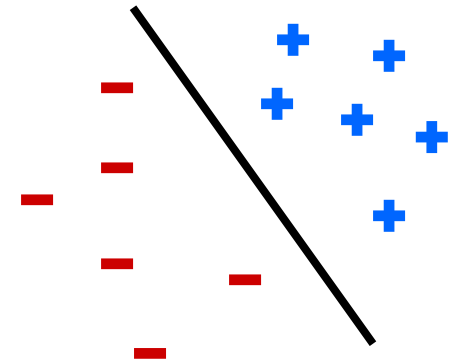


Properties of Perceptrons

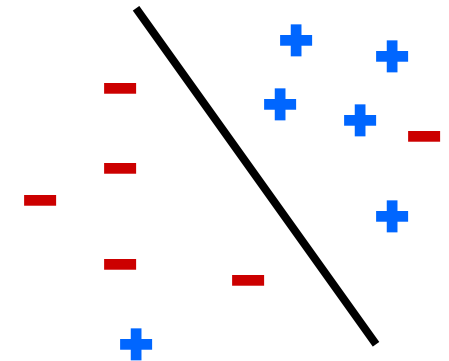
- Separability: true if some parameters get the training set perfectly correct
- Convergence: if the training is separable, perceptron will eventually converge (binary case)
- Mistake Bound: the maximum number of mistakes (binary case) related to the *margin* or degree of separability

$$\text{mistakes} < \frac{k}{\delta^2}$$

Separable

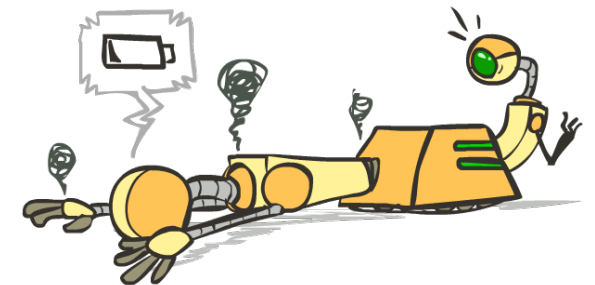
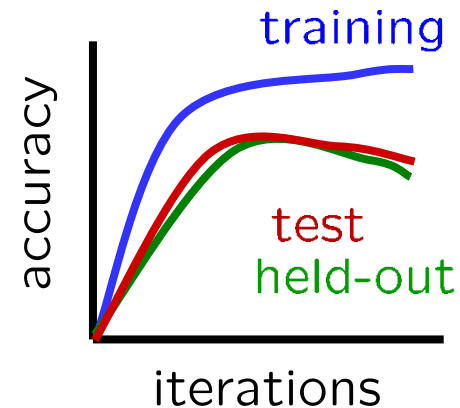
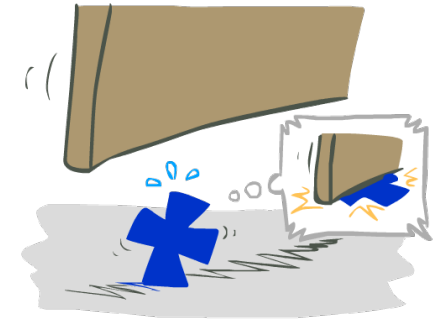
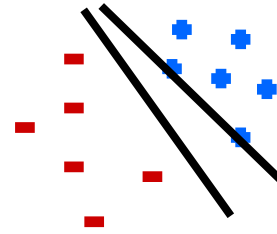
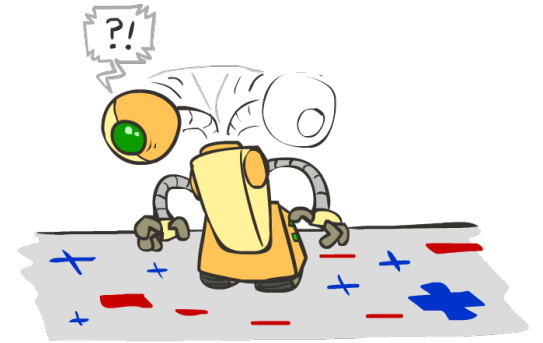
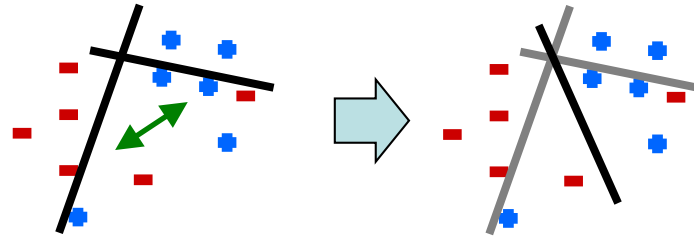


Non-Separable

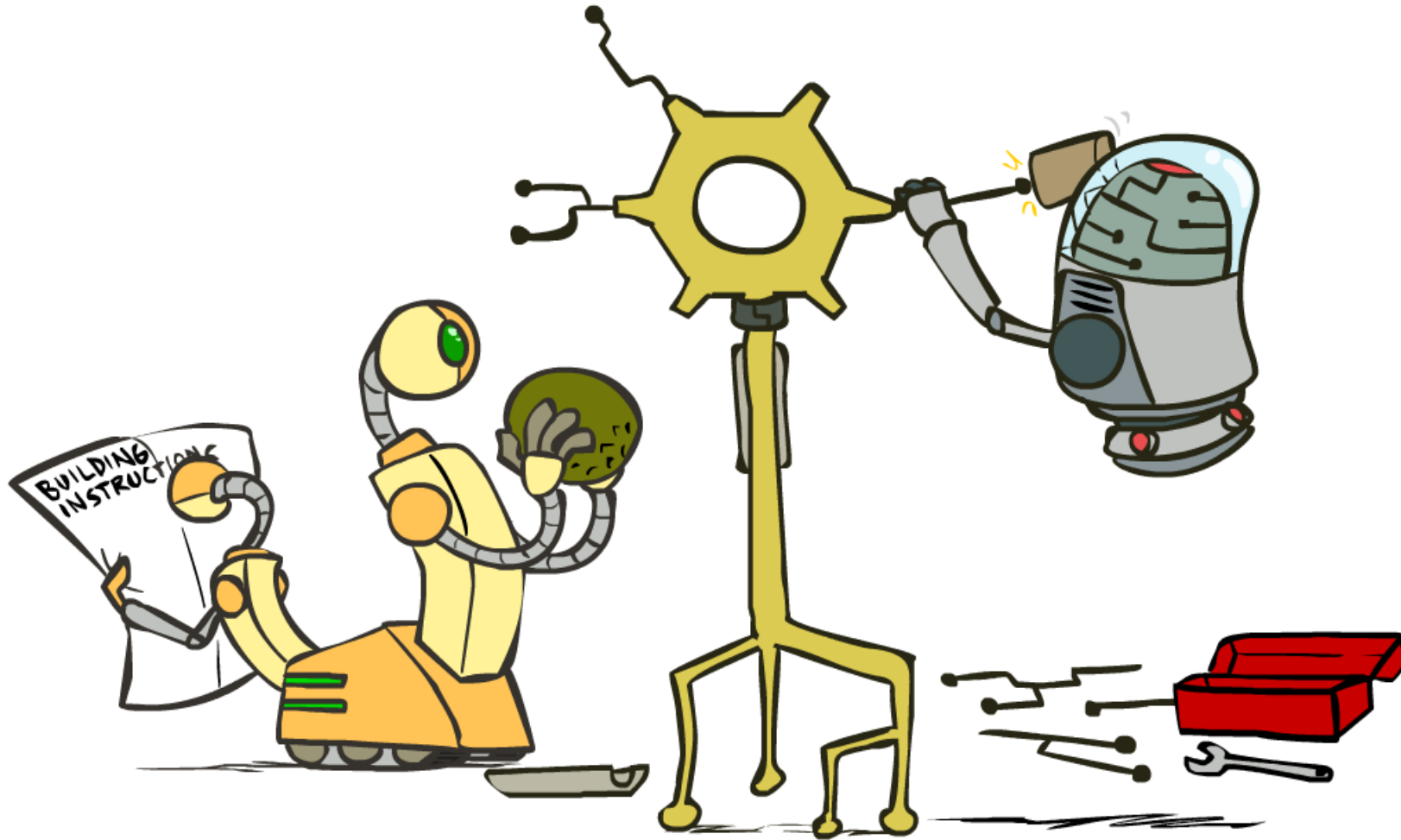


Problems with the Perceptron

- 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

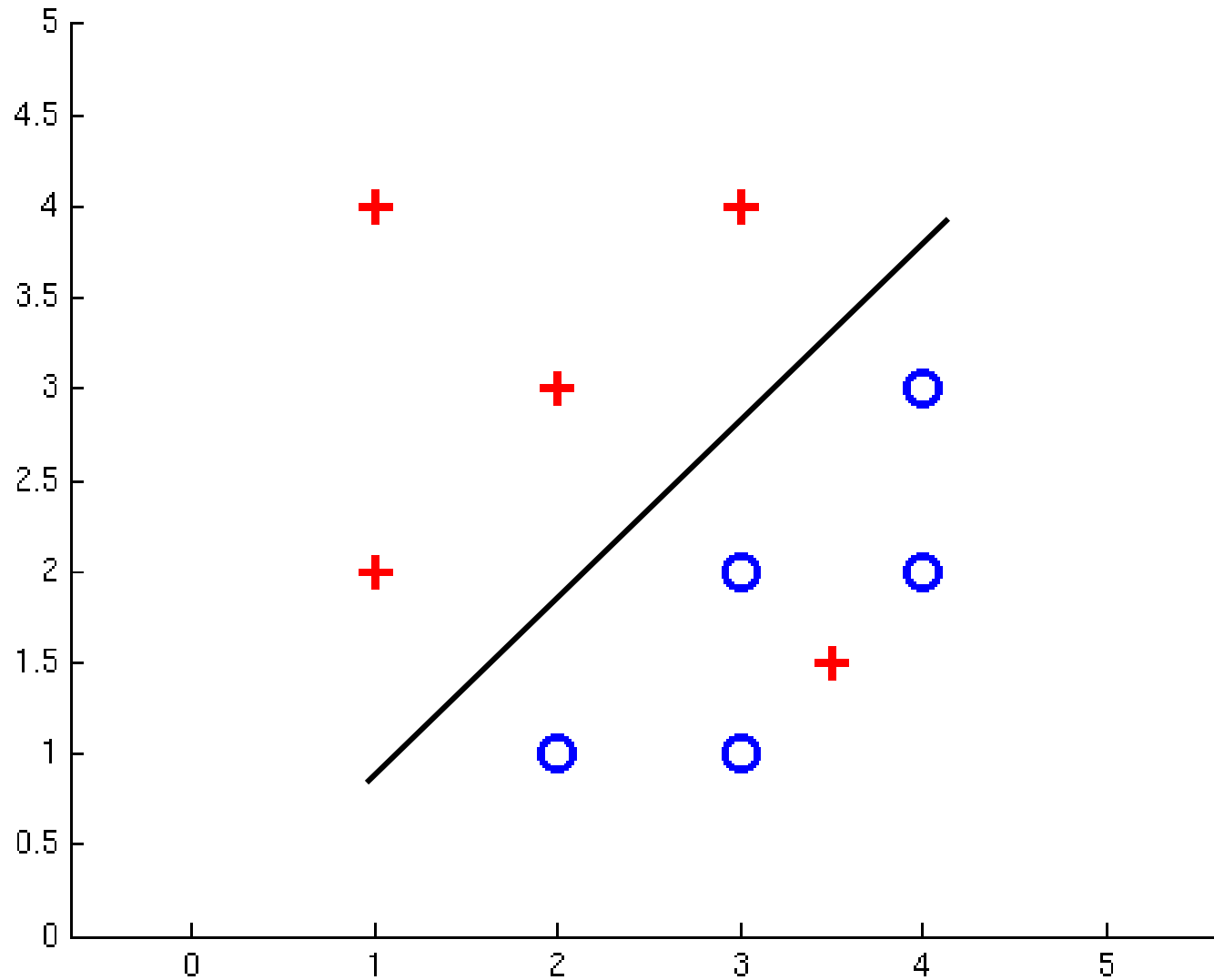


Improving the Perceptron

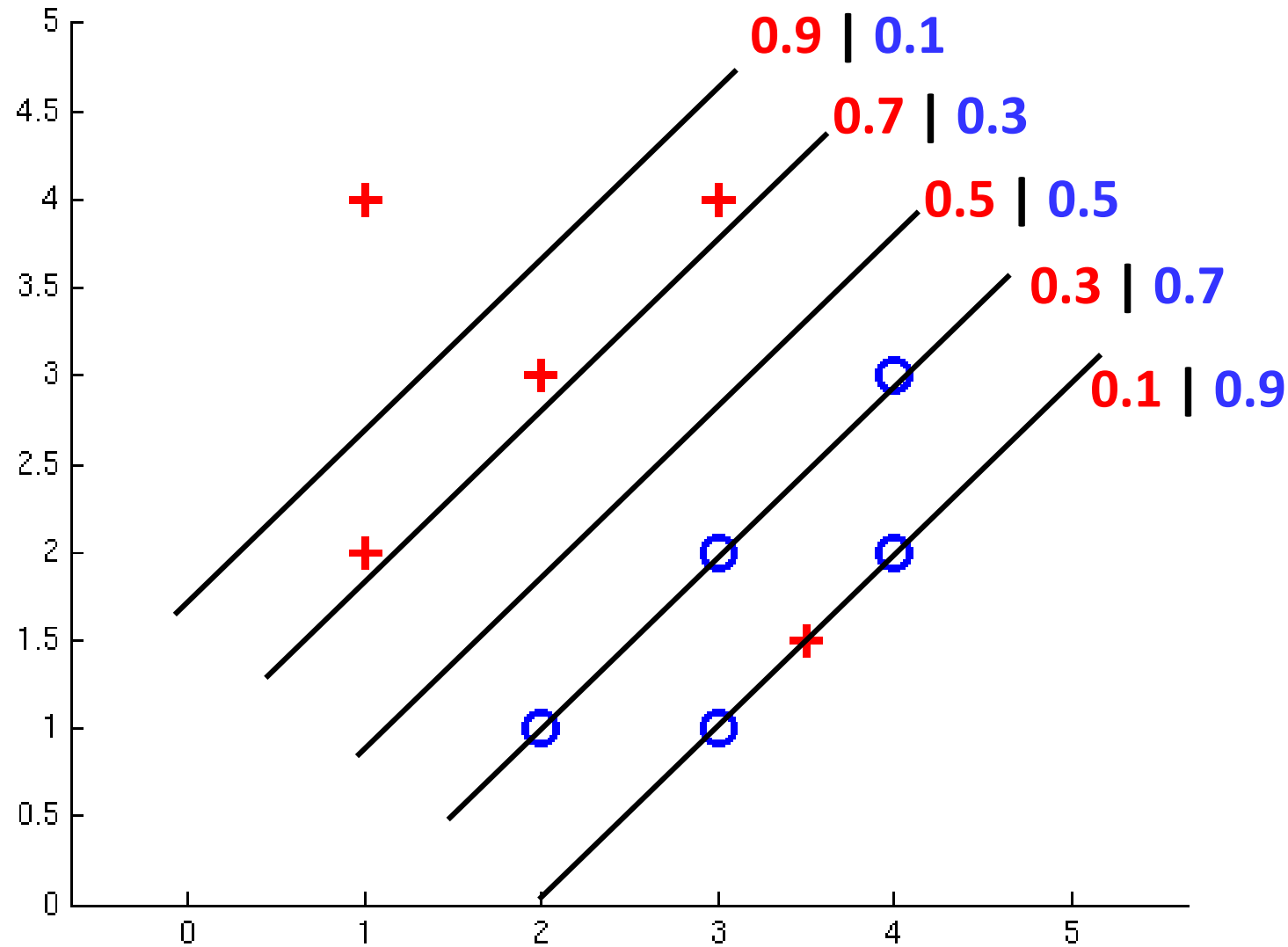


Non-Separable Case: Deterministic Decision

Even the best linear boundary makes at least one mistake



Non-Separable Case: Probabilistic Decision

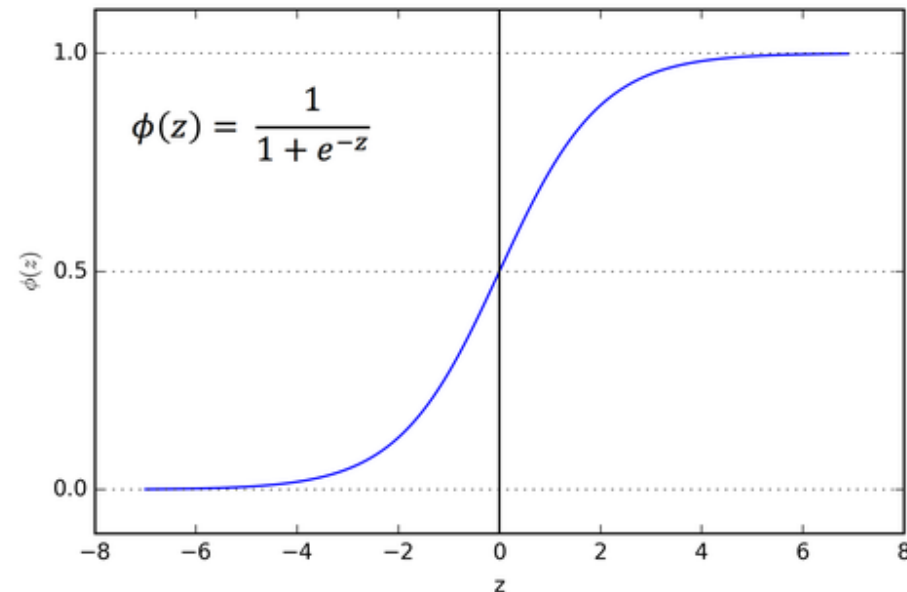


How to get probabilistic decisions?

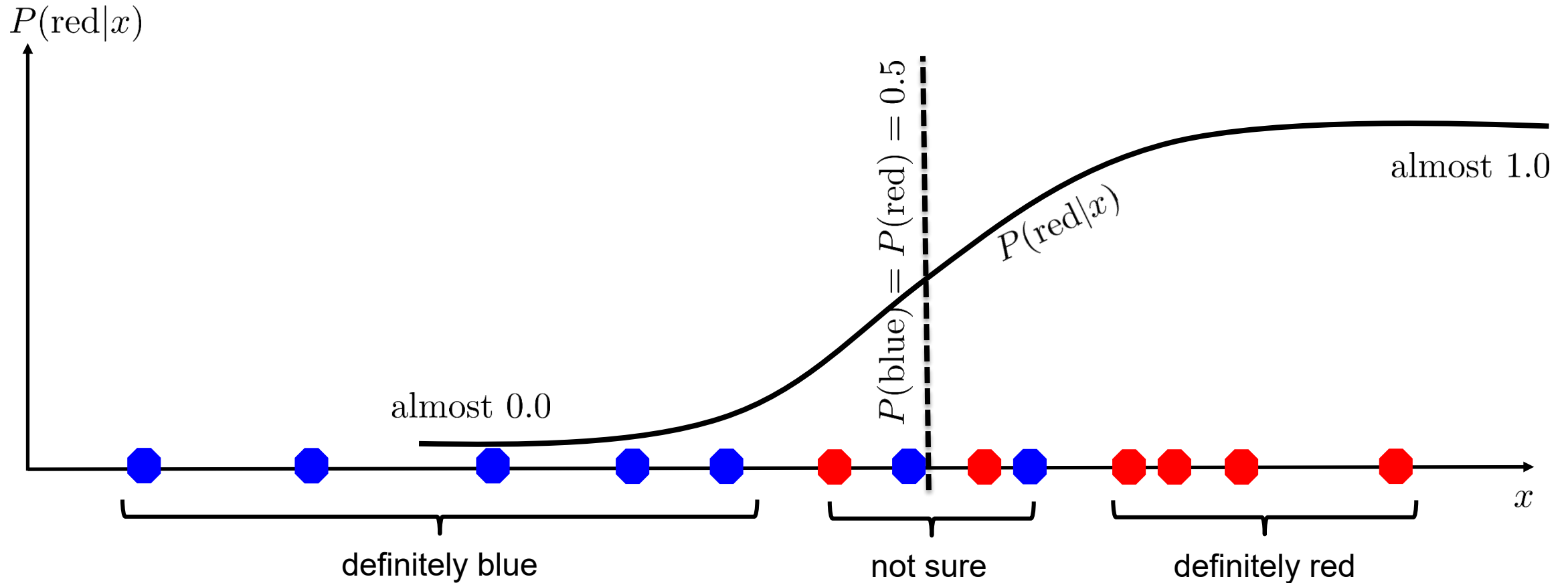
- Perceptron scoring: $z = w \cdot f(x)$
- If $z = w \cdot f(x)$ very positive \rightarrow want probability going to 1
- If $z = w \cdot f(x)$ very negative \rightarrow want probability going to 0

- Sigmoid function

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



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

Best w ?

- Maximum likelihood estimation:

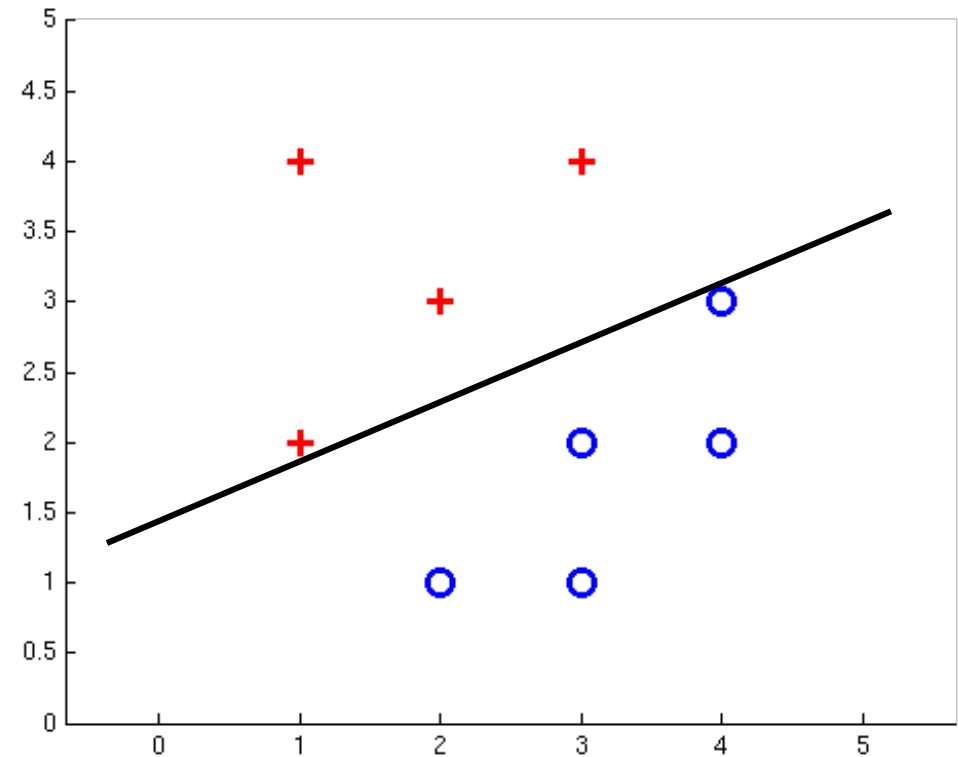
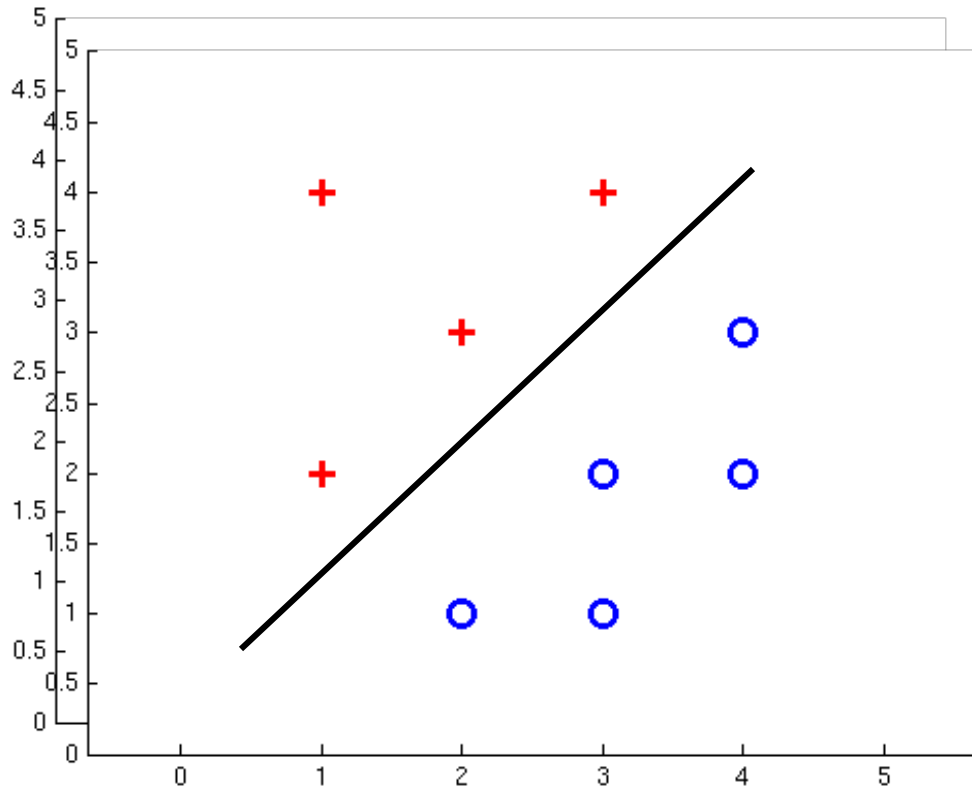
$$\max_w ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

with:

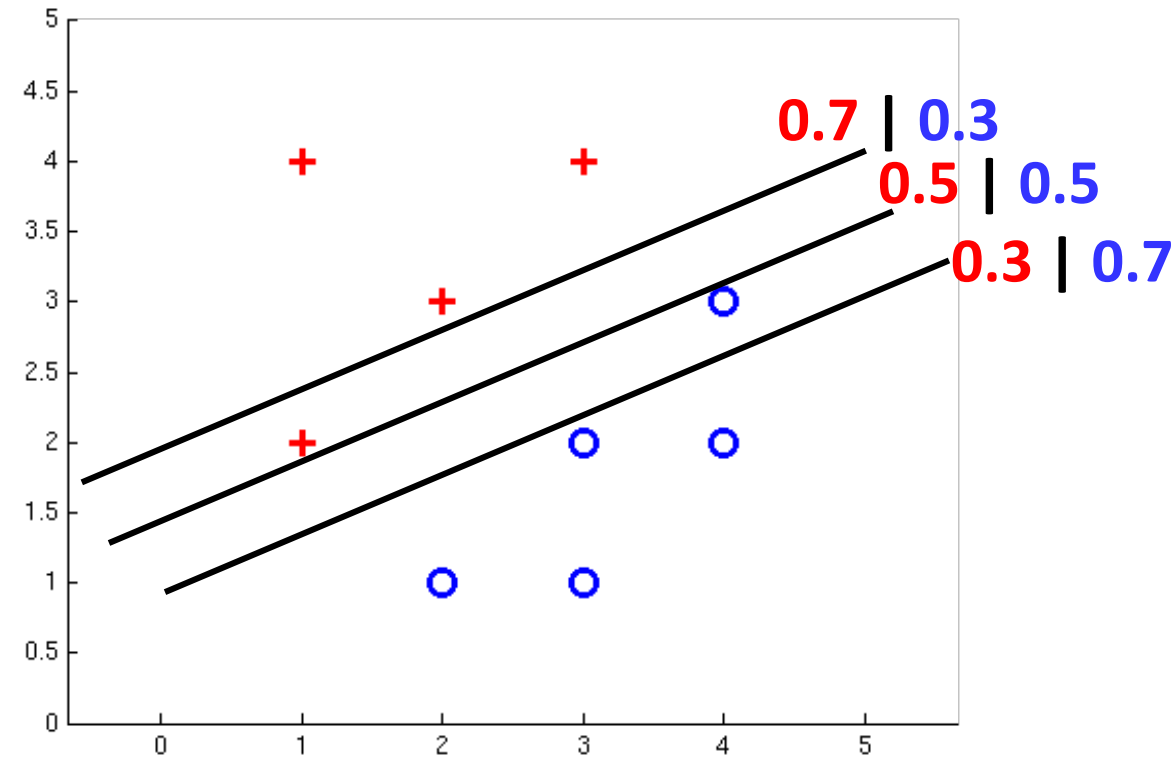
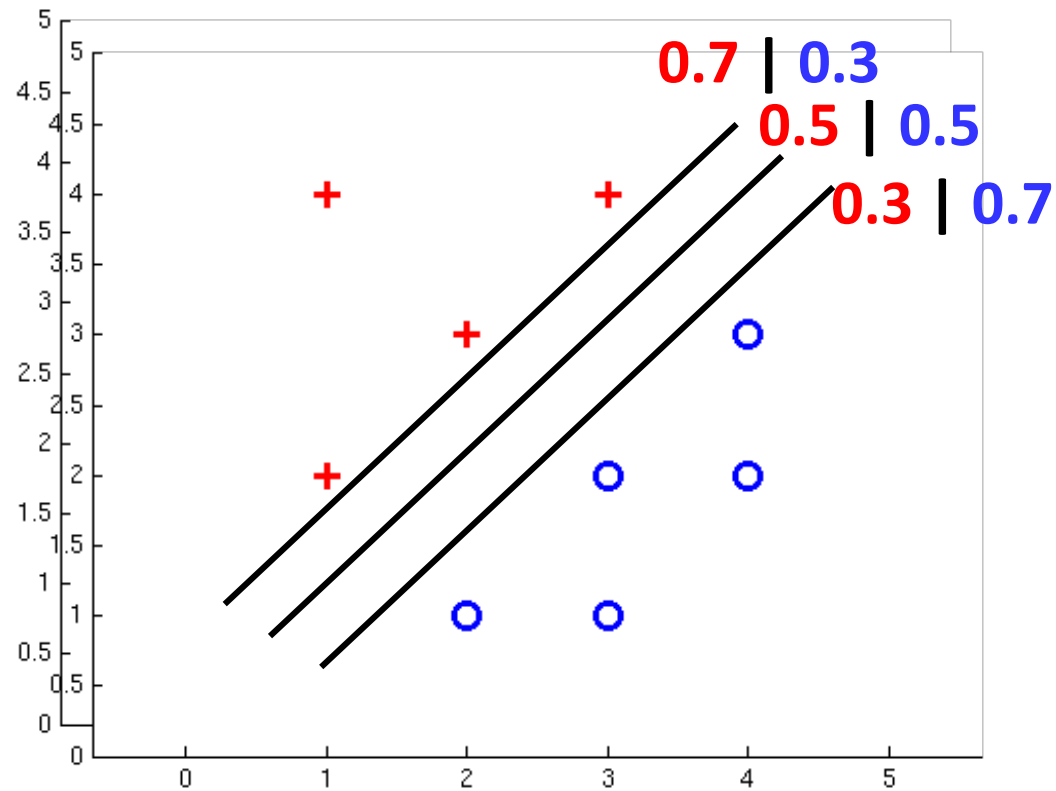
$$P(y^{(i)} = +1 | x^{(i)}; w) = \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}$$
$$P(y^{(i)} = -1 | x^{(i)}; w) = 1 - \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}$$

= Logistic Regression

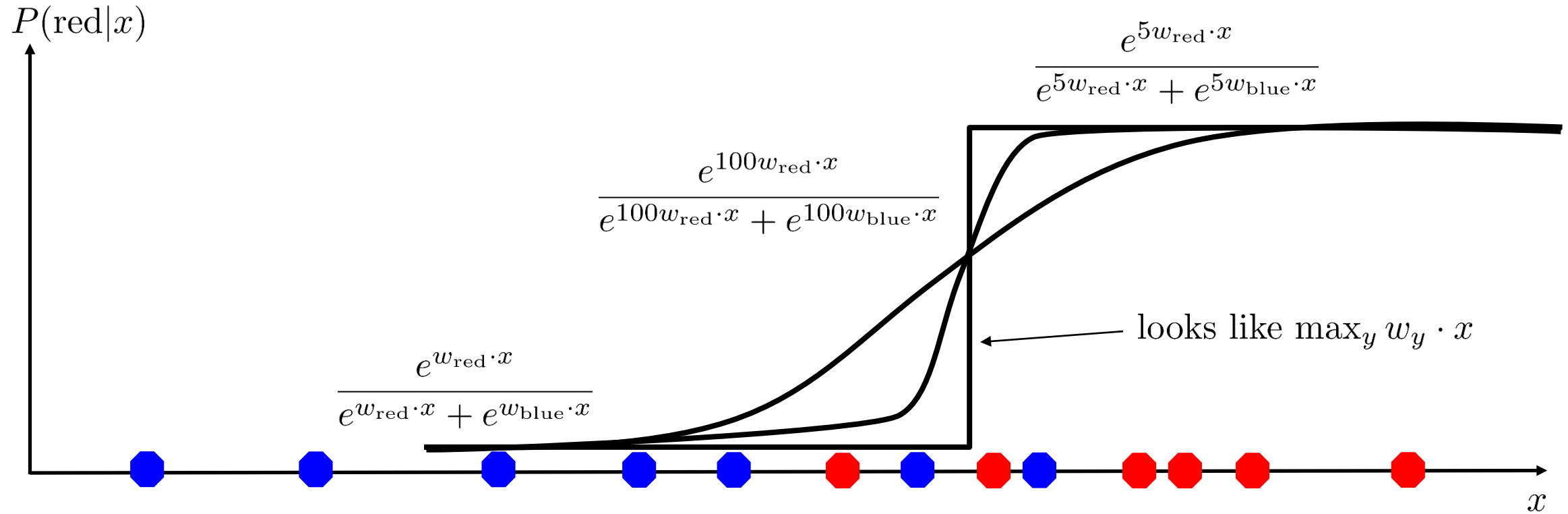
Separable Case: Deterministic Decision – Many Options



Separable Case: Probabilistic Decision – Clear Preference



The Soft Max

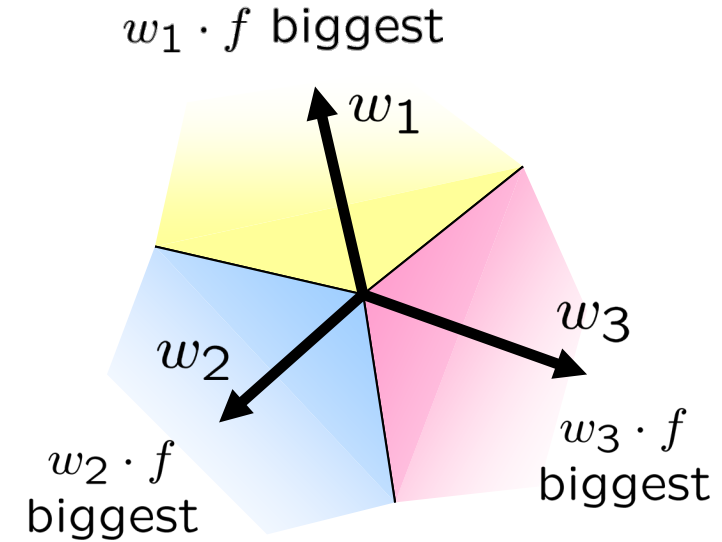


$$P(\text{red}|x) = \frac{e^{w_{\text{red}} \cdot x}}{e^{w_{\text{red}} \cdot x} + e^{w_{\text{blue}} \cdot x}}$$

Multiclass Logistic Regression

- Recall Perceptron:

- A weight vector for each class: w_y
- Score (activation) of a class y : $w_y \cdot f(x)$
- Prediction highest score wins $y = \arg \max_y w_y \cdot f(x)$



- How to make the scores into probabilities?

$$\underbrace{z_1, z_2, z_3}_{\text{original activations}} \rightarrow \underbrace{\frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}, \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}, \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}}_{\text{softmax activations}}$$

Optimization

- i.e., how do we solve:

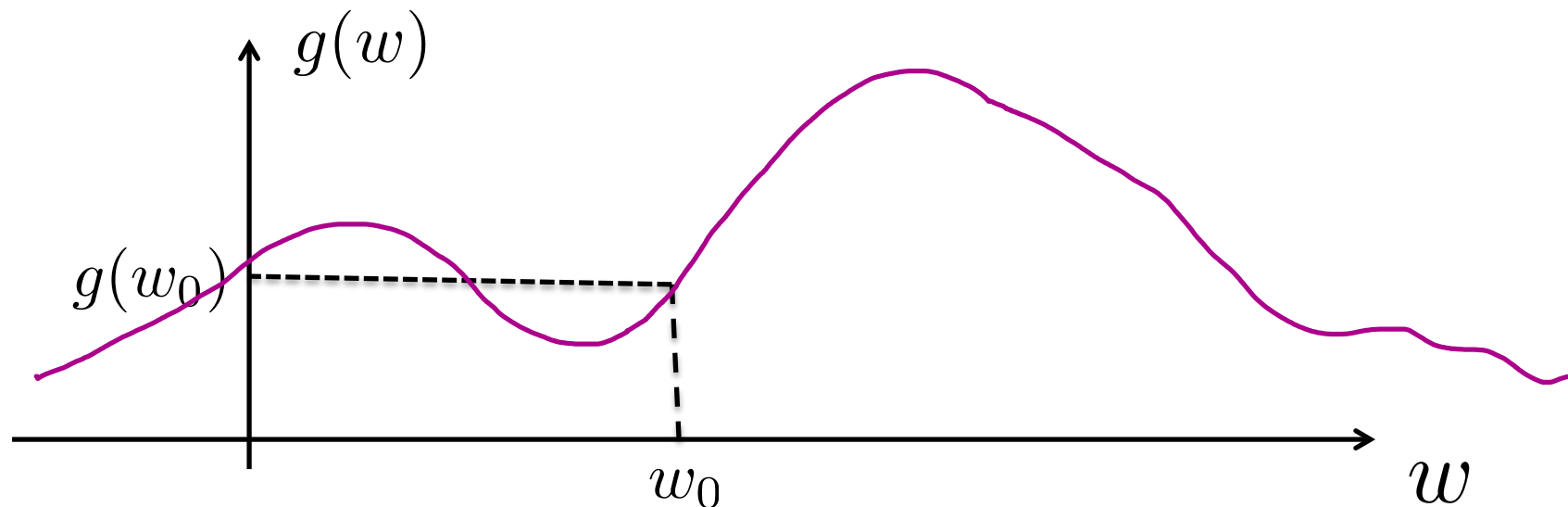
$$\max_w ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

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



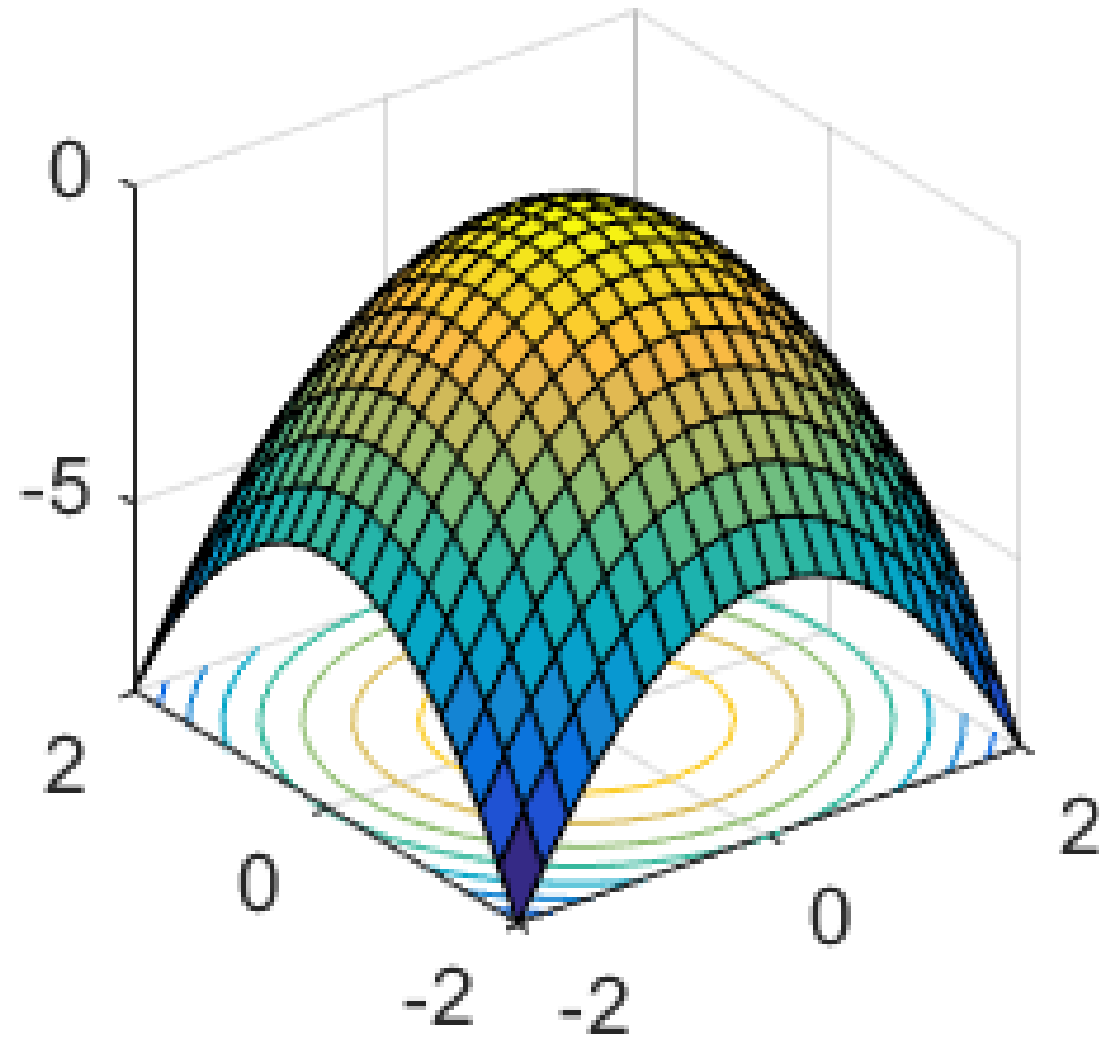
- Could evaluate $g(w_0 + h)$ and $g(w_0 - h)$

- Then step in best direction

- Or, evaluate derivative:
$$\frac{\partial g(w_0)}{\partial w} = \lim_{h \rightarrow 0} \frac{g(w_0 + h) - g(w_0 - h)}{2h}$$

- Tells which direction to step into

2-D Optimization



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:

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

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

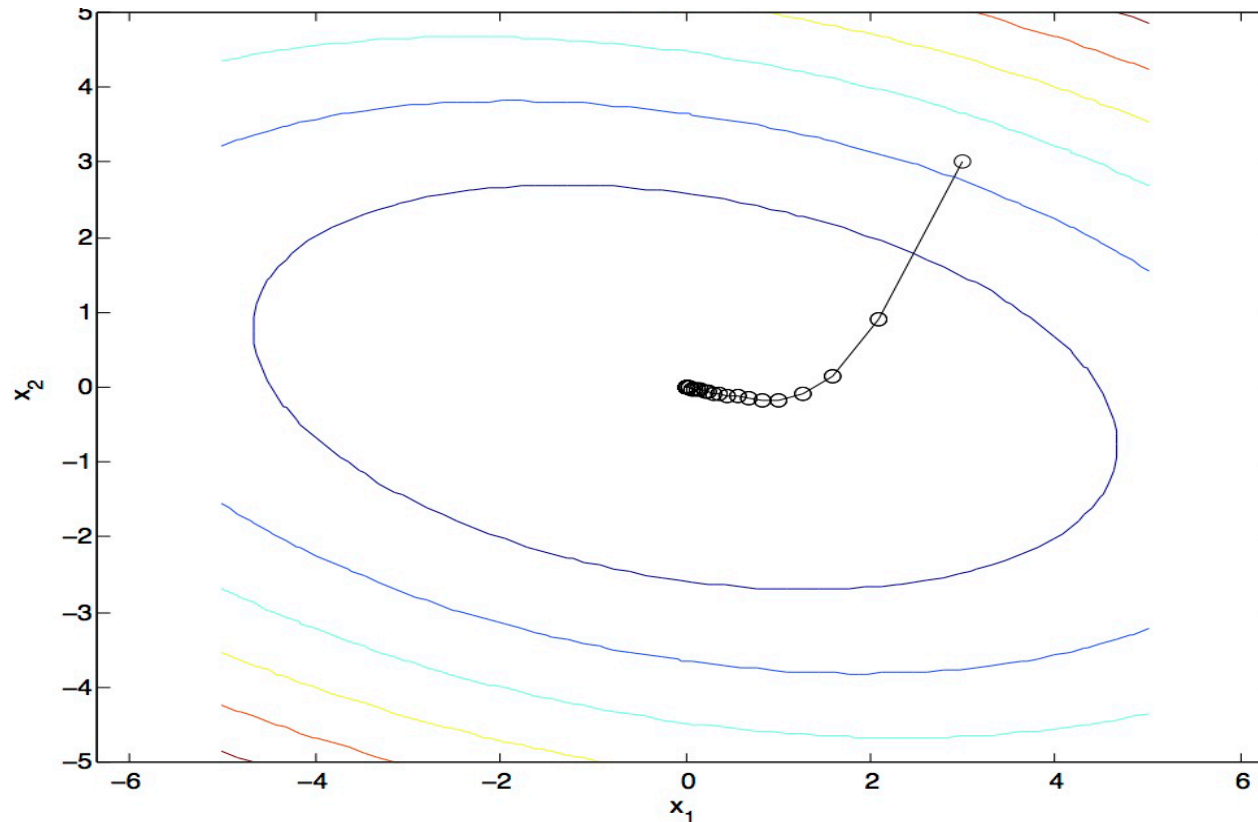
- Updates in vector notation:

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

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

Gradient Ascent

- Idea:
 - Start somewhere
 - Repeat: Take a step in the gradient direction



What is the Steepest Direction?

$$\max_{\Delta: \Delta_1^2 + \Delta_2^2 \leq \varepsilon} g(w + \Delta)$$



- First-Order Taylor Expansion:

$$g(w + \Delta) \approx g(w) + \frac{\partial g}{\partial w_1} \Delta_1 + \frac{\partial g}{\partial w_2} \Delta_2$$

- Steepest Ascent Direction:

$$\max_{\Delta: \Delta_1^2 + \Delta_2^2 \leq \varepsilon} g(w) + \frac{\partial g}{\partial w_1} \Delta_1 + \frac{\partial g}{\partial w_2} \Delta_2$$

- Recall: $\max_{\Delta: \|\Delta\| \leq \varepsilon} \Delta^\top a \rightarrow$

$$\Delta = \varepsilon \frac{a}{\|a\|}$$

- Hence, solution: $\Delta = \varepsilon \frac{\nabla g}{\|\nabla g\|}$

Gradient direction = steepest direction!

$$\nabla g = \begin{bmatrix} \frac{\partial g}{\partial w_1} \\ \frac{\partial g}{\partial w_2} \end{bmatrix}$$

Gradient in n dimensions

$$\nabla g = \begin{bmatrix} \frac{\partial g}{\partial w_1} \\ \frac{\partial g}{\partial w_2} \\ \dots \\ \frac{\partial g}{\partial w_n} \end{bmatrix}$$

Optimization Procedure: Gradient Ascent

```
■ init  $w$   
■ for iter = 1, 2, ...  

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

```

- α : learning rate --- tweaking parameter that needs to be chosen carefully
- How? Try multiple choices
 - Crude rule of thumb: update changes w about 0.1 – 1 %

Batch Gradient Ascent on the Log Likelihood Objective

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

- `init w`
- `for iter = 1, 2, ...`

$$w \leftarrow w + \alpha * \sum_i \nabla \log P(y^{(i)} | x^{(i)}; w)$$

What will gradient ascent do?

$$w \leftarrow w + \alpha * \sum_i \nabla \log P(y^{(i)} | x^{(i)}; w)$$

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

$$\nabla w_{y^{(i)}} f(x^{(i)}) - \nabla \log \sum_y e^{w_y f(x^{(i)})}$$

adds f to the correct
class weights

$$\frac{1}{\sum_y e^{w_y f(x^{(i)})}} \sum_y \left(e^{w_y f(x^{(i)})} [0^T f(x^{(i)})^T 0^T]^T \right)$$

$$\text{for } y' \text{ weights: } \frac{1}{\sum_y e^{w_y f(x^{(i)})}} e^{w_{y'} f(x^{(i)})} f(x^{(i)})$$

$P(y' | x^{(i)}; w) f(x^{(i)})$ subtracts f from y' weights in proportion to
the probability current weights give to y'

Stochastic Gradient Ascent on the Log Likelihood Objective

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

Observation: once gradient on one training example has been computed, might as well incorporate before computing next one

- `init w`
- `for iter = 1, 2, ...`
 - pick random j

$$w \leftarrow w + \alpha * \nabla \log P(y^{(j)} | x^{(j)}; w)$$

Mini-Batch Gradient Ascent on the Log Likelihood Objective

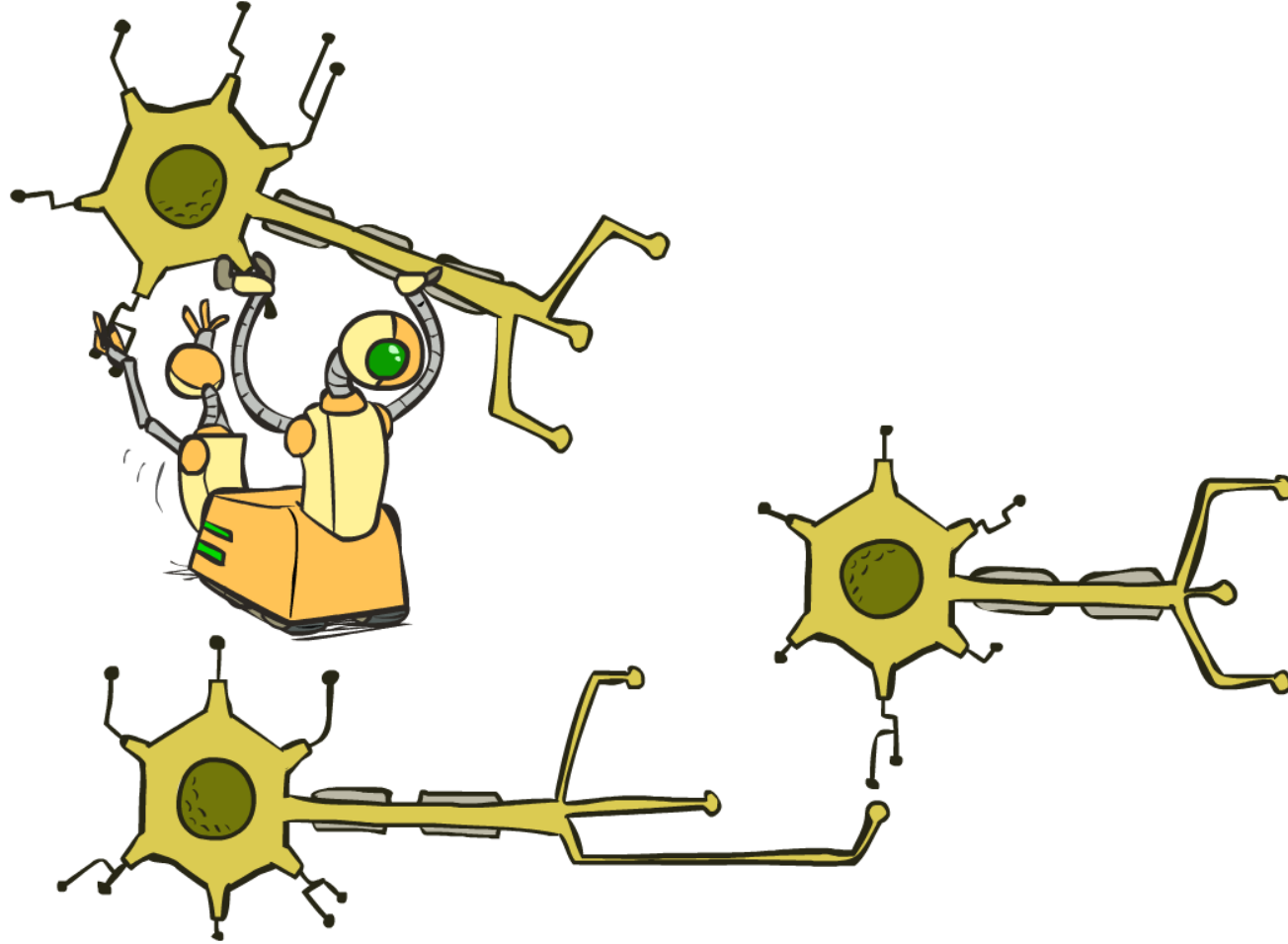
$$\max_w ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

Observation: gradient over small set of training examples (=mini-batch) can be computed in parallel, might as well do that instead of a single one

- `init` w
- `for` `iter` = 1, 2, ...
 - pick random subset of training examples J

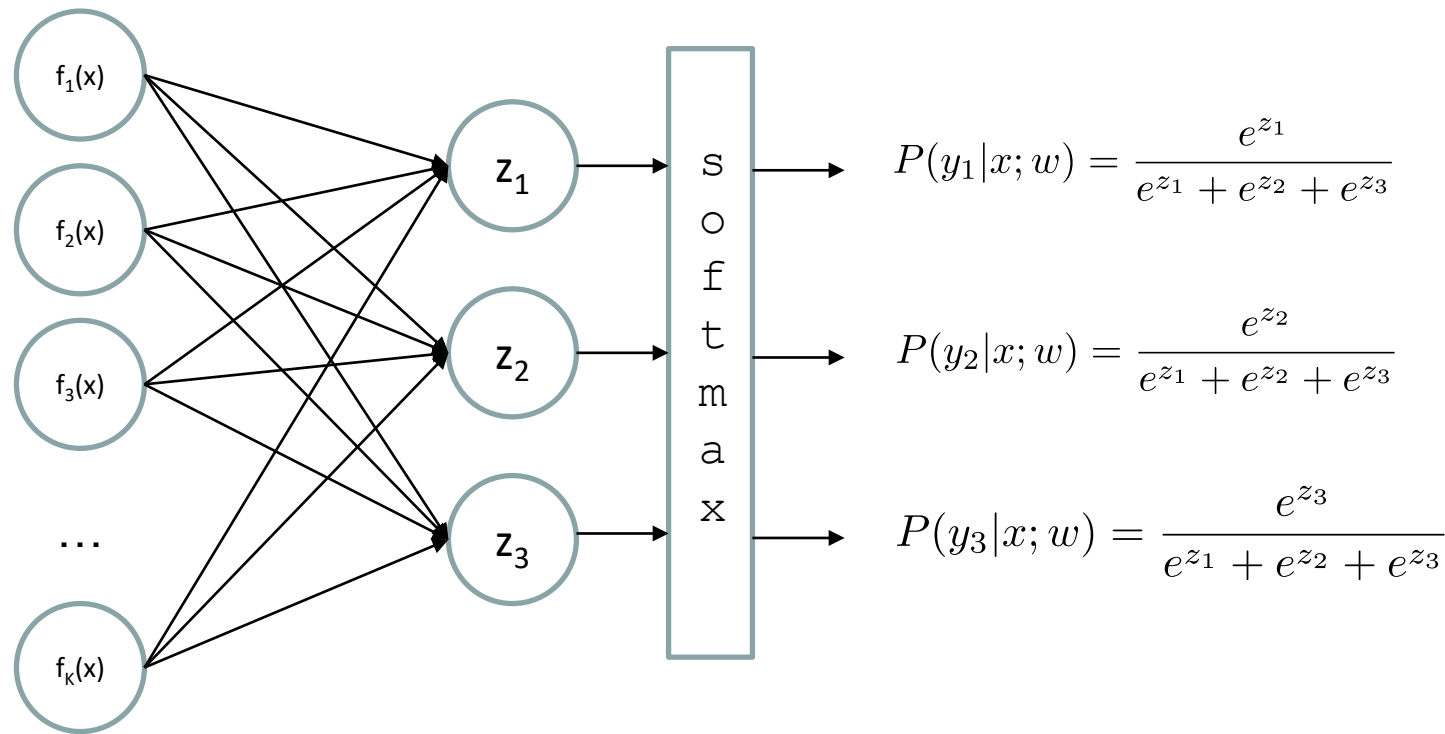
$$w \leftarrow w + \alpha * \sum_{j \in J} \nabla \log P(y^{(j)} | x^{(j)}; w)$$

Neural Networks

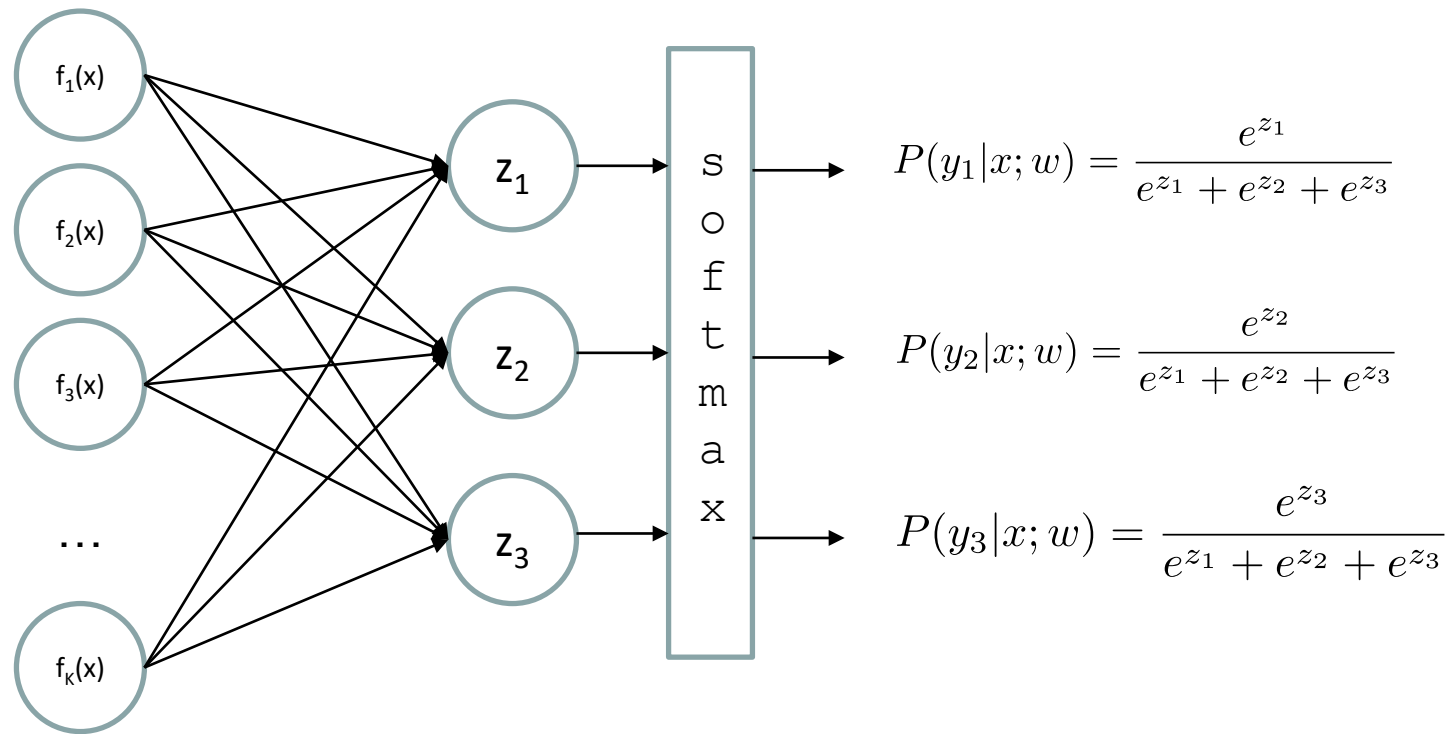


Multi-class Logistic Regression

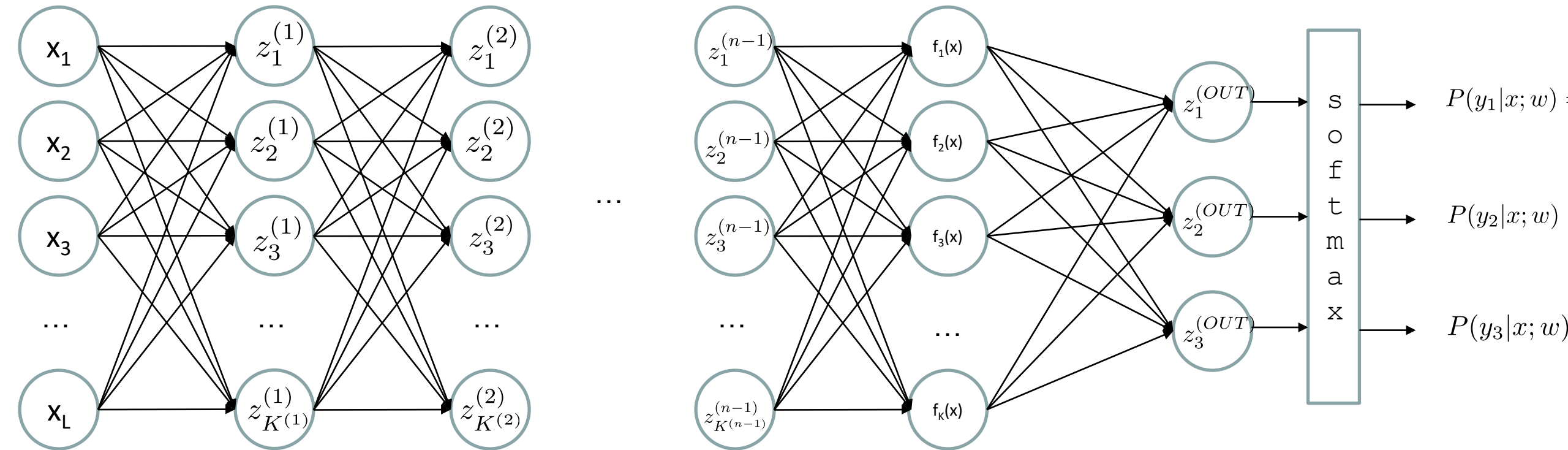
- = special case of neural network



Deep Neural Network = Also learn the features!



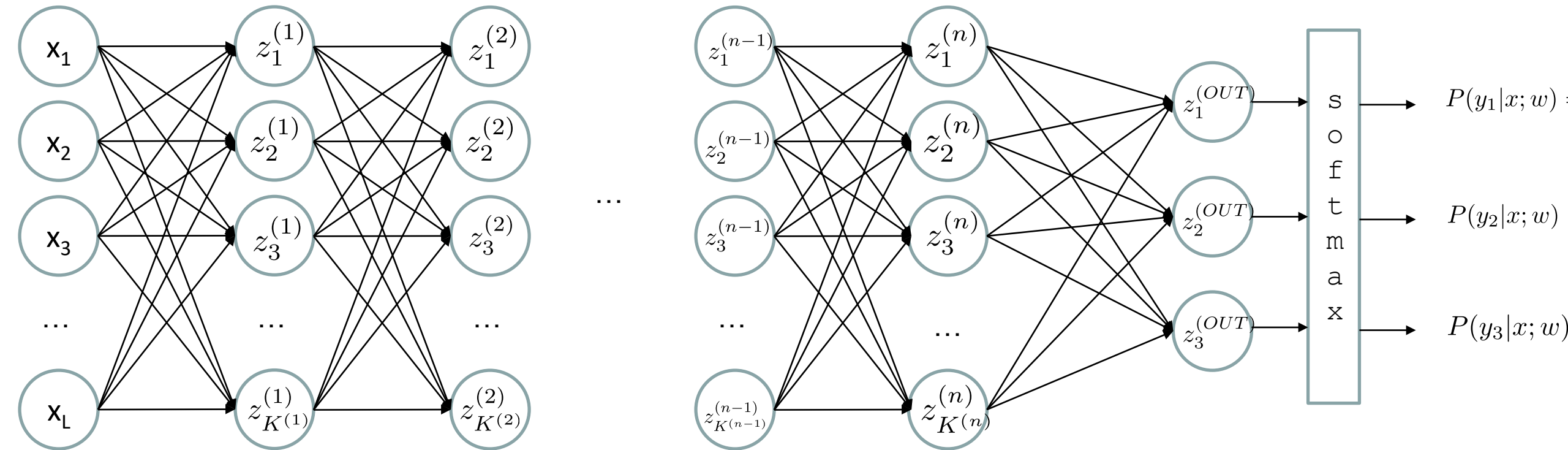
Deep Neural Network = Also learn the features!



$$z_i^{(k)} = g\left(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\right)$$

g = nonlinear activation function

Deep Neural Network = Also learn the features!

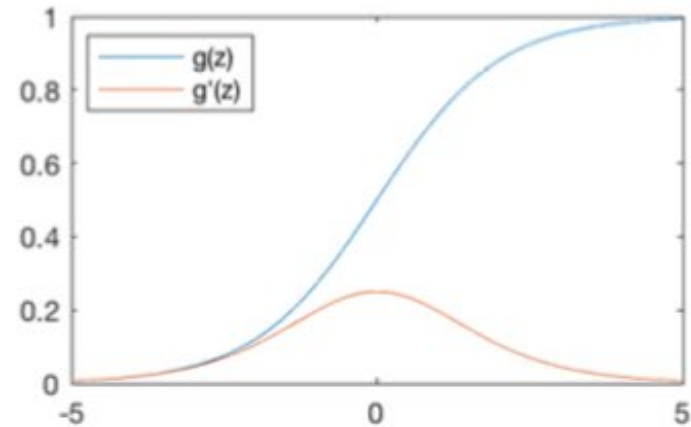


$$z_i^{(k)} = g\left(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\right)$$

g = nonlinear activation function

Common Activation Functions

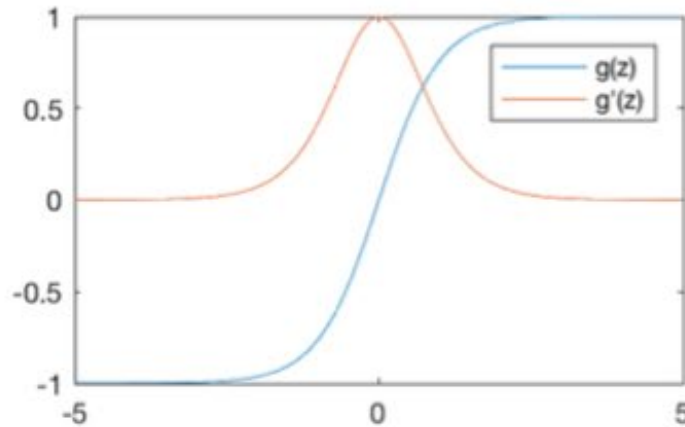
Sigmoid Function



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

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

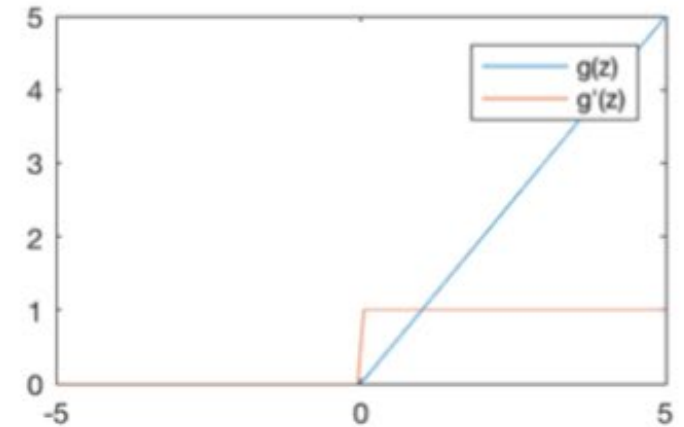
Hyperbolic Tangent



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

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

Rectified Linear Unit (ReLU)



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

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

Deep Neural Network: Also Learn the Features!

- Training the deep neural network is just like logistic regression:

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

just w tends to be a much, much larger vector 😊

→ just run gradient ascent

+ stop when log likelihood of hold-out data starts to decrease

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!)

How about computing all the derivatives?

- Derivatives tables:

$$\frac{d}{dx}(a) = 0$$

$$\frac{d}{dx}(x) = 1$$

$$\frac{d}{dx}(au) = a \frac{du}{dx}$$

$$\frac{d}{dx}(u + v - w) = \frac{du}{dx} + \frac{dv}{dx} - \frac{dw}{dx}$$

$$\frac{d}{dx}(uv) = u \frac{dv}{dx} + v \frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{1}{v} \frac{du}{dx} - \frac{u}{v^2} \frac{dv}{dx}$$

$$\frac{d}{dx}(u^n) = nu^{n-1} \frac{du}{dx}$$

$$\frac{d}{dx}(\sqrt{u}) = \frac{1}{2\sqrt{u}} \frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{1}{u}\right) = -\frac{1}{u^2} \frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{1}{u^n}\right) = -\frac{n}{u^{n+1}} \frac{du}{dx}$$

$$\frac{d}{dx}[f(u)] = \frac{d}{du}[f(u)] \frac{du}{dx}$$

$$\frac{d}{dx}[\ln u] = \frac{d}{dx}[\log_e u] = \frac{1}{u} \frac{du}{dx}$$

$$\frac{d}{dx}[\log_a u] = \log_a e \frac{1}{u} \frac{du}{dx}$$

$$\frac{d}{dx}e^u = e^u \frac{du}{dx}$$

$$\frac{d}{dx}a^u = a^u \ln a \frac{du}{dx}$$

$$\frac{d}{dx}(u^v) = vu^{v-1} \frac{du}{dx} + \ln u \cdot u^v \frac{dv}{dx}$$

$$\frac{d}{dx} \sin u = \cos u \frac{du}{dx}$$

$$\frac{d}{dx} \cos u = -\sin u \frac{du}{dx}$$

$$\frac{d}{dx} \tan u = \sec^2 u \frac{du}{dx}$$

$$\frac{d}{dx} \cot u = -\csc^2 u \frac{du}{dx}$$

$$\frac{d}{dx} \sec u = \sec u \tan u \frac{du}{dx}$$

$$\frac{d}{dx} \csc u = -\csc u \cot u \frac{du}{dx}$$

How about computing all the derivatives?

- But neural net f is never one of those?
 - No problem: CHAIN RULE:

If $f(x) = g(h(x))$

Then $f'(x) = g'(h(x))h'(x)$

→ Derivatives can be computed by following well-defined procedures

Automatic Differentiation

- Automatic differentiation software
 - e.g. PyTorch, TensorFlow, JAX
 - Only need to program the function $g(x,y,w)$
 - Can automatically compute all derivatives w.r.t. all entries in w
 - This is typically done by caching info during forward computation pass of f , and then doing a backward pass = “backpropagation”
 - Autodiff / Backpropagation can often be done at computational cost comparable to the forward pass
- Need to know this exists
- How this is done? -- outside of scope

Summary of Key Ideas

- Optimize probability of label given input $\max_w ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$
- Continuous optimization
 - Gradient ascent:
 - Compute steepest uphill direction = gradient (= just vector of partial derivatives)
 - Take step in the gradient direction
 - Repeat (until held-out data accuracy starts to drop = “early stopping”)
- Deep neural nets
 - Last layer = still logistic regression
 - Now also many more layers before this last layer
 - = computing the features
 - → the features are learned rather than hand-designed
 - Universal function approximation theorem
 - If neural net is large enough
 - Then neural net can represent any continuous mapping from input to output with arbitrary accuracy
 - But remember: need to avoid overfitting / memorizing the training data → early stopping!
 - Automatic differentiation gives the derivatives efficiently (how? = outside of scope of 188)