
6.867 Fall 2017

Neural Networks

Introduction to neural networks

Ack: Tommi Jaakkola (for some parts)

Lecture 10: 12th Oct, 2017



Admin



Project Milestone 2: overdue today!

Exam: **10/19, 7:30pm-9:00pm**

Exam review: during recitation

Noted additional reading material for *logistic regression, kernels, svms*.

Introduction

Recap

Explicit nonlinear features: **prespecified map**

Implicit nonlinear features via **kernels**

Generic guideline: Select nonlinear map or kernels according to type of prior knowledge or structure you wish to impose (e.g., RBF, Polynomial, etc.)

Intuitively: kernels capture notion of pairwise similarity

The machine learning mindset

What if we learn the nonlinear map from data instead of prespecifying it?

As MLers, you should often ask this question more broadly:
“what if we just learn **《blah》** from data instead of fixing it by hand”

Introduction: motivating view 1

Recall that when using kernels for any point 'x' we have an implicit feature map $\phi(x)$

Kernel SVM

$$w = \sum_i \alpha_i y_i \phi(x_i)$$

$$\langle w, \phi(x) \rangle = \sum_i \alpha_i y_i k(x_i, x)$$

$$\Phi(x) := [y_1 k(x_1, x), \dots, y_n k(x_n, x)]$$

$$\langle w, \phi(x) \rangle = \sum_i \alpha_i [\Phi(x)]_i = \langle \alpha, \Phi(x) \rangle$$

Thus, our classifier is parameterized by α and computed on the nonlinear features $\Phi(x)$

These features not defined in advance but they depend on the training data since $\Phi(x)$ depends on similarity of 'x' to each training data point x_i

Introduction: motivating view 1

However, these nonlinear features Φ **not** “learned” from data

Why?

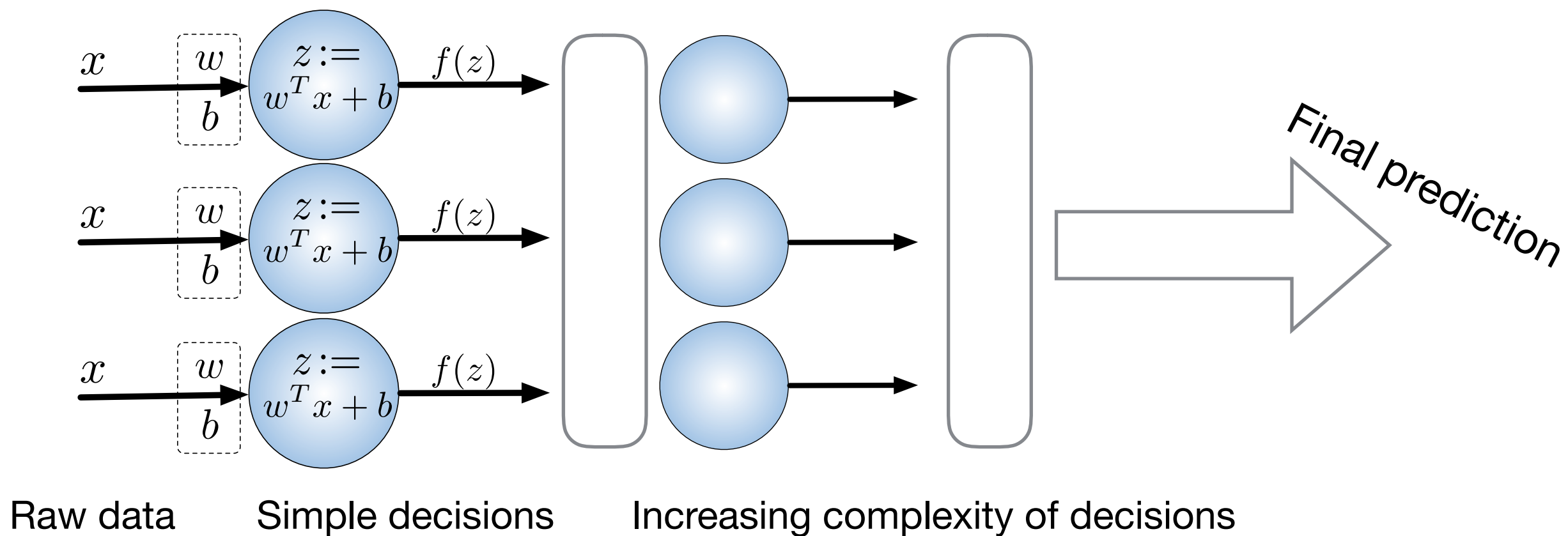
Because we do not construct Φ to optimize classification performance.
We just learn α in an SVM for instance.

- Thus, if we could learn Φ from data while optimizing classification performance, we may obtain a more power classifier.
- This task seems enormously more difficult computationally.
- Today we will talk about one such approach: *neural networks*, where we jointly learn the features and the classifier.

Introduction: motivating view 2

Building complex models

- ▶ Simple decisions feed into next stage for making more complex decisions
- ▶ These feed into next stage for even more high-level decisions and so on
- ▶ Think of decomposing a big-problem into smaller parts
- ▶ The small parts then (we hope) solve increasingly sophisticated problems
- ▶ More broadly, each “computational unit” does a simple task that feeds into next stage whose output is a more sophisticated function of the raw input



Neural networks

- * An **Artificial Neural Network (ANN)** contains several such units (aka “neurons”; we prefer the term “units”)
- * Each unit performs a task such as linear classification, detection of maxima, linear transformations of its input, scaling, normalization, etc.
- * Simplest setup: units arranged in layers, each layer consumes output of previous layer (thus, one person’s output is another’s input) 🤪

Jargon

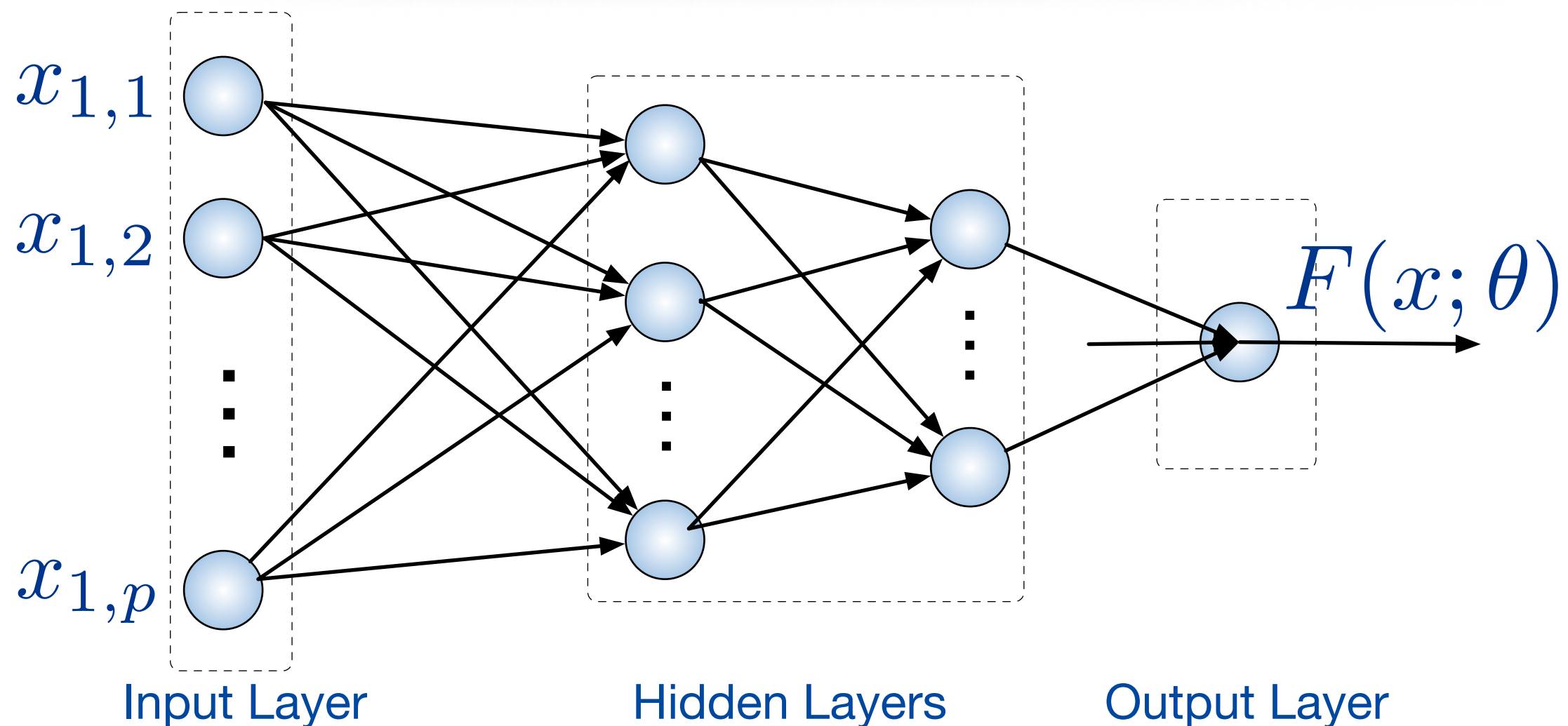
Input layer: Synonym for input data. The units store the input data eg. vectors

Hidden layers: The bulk of the neural network architecture. Think of each subsequent hidden layer generating more complex nonlinear features

Output layer: Can be a single unit (eg when doing binary classification) or multiple units (eg for multiclass classification). This layer receives features from penultimate layer and outputs predictions for instance

Convention: 2-layer network will refer to 2 layers, 1-hidden, 1 output; we don’t count the input layer.

Neural networks



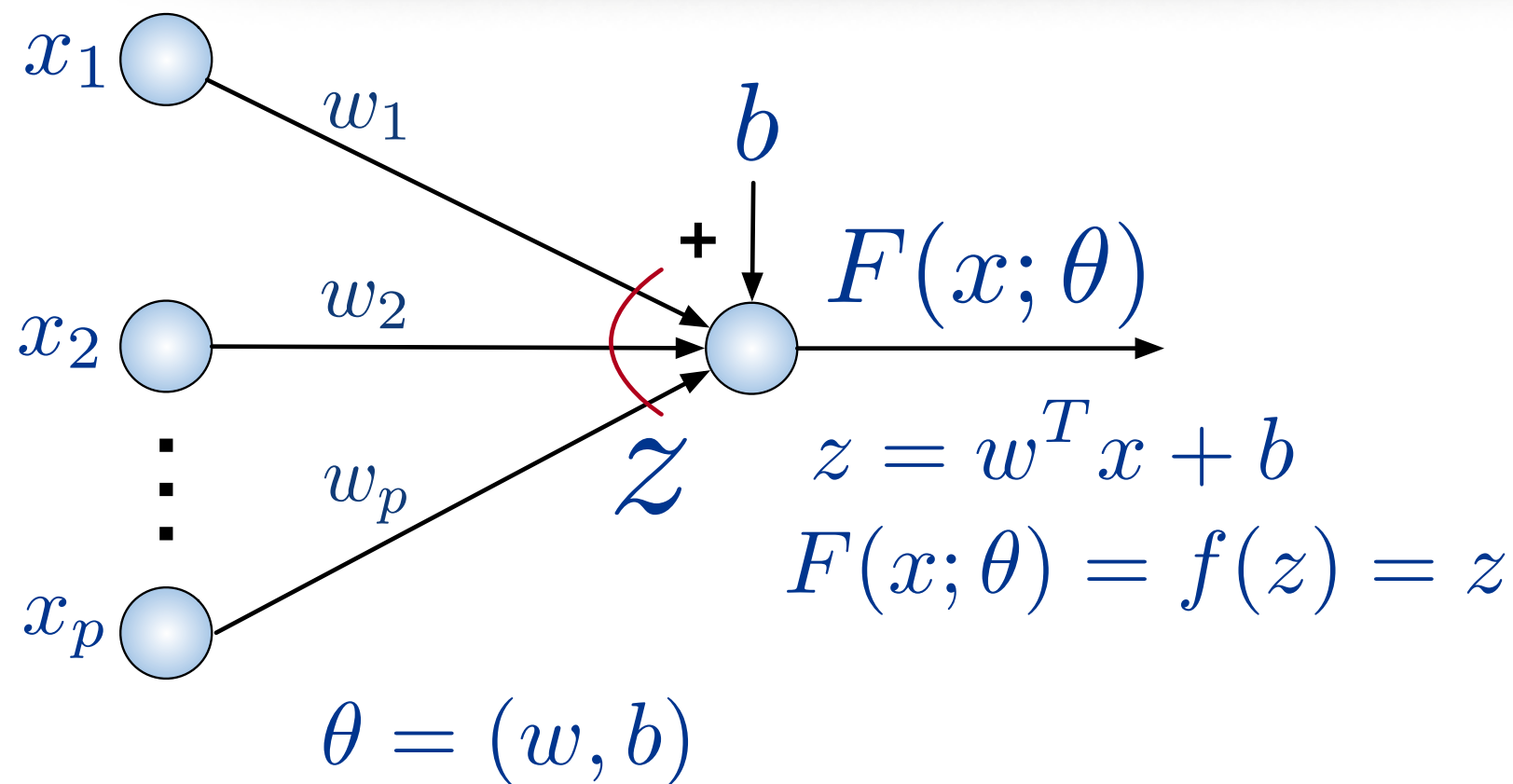
Input: vector 'x' (p-dimensional vector, say)

Hidden layers: Have corresponding 'w' weights, aka weight vectors / matrices

Output: $F(x; \theta)$ network output is a function of input 'x' and parameters θ

Some people call this “multilayer perceptron” (MLP), but that is a misnomer (**why?**), so we will just call them ANNs, feedforward NN, etc.

Toy example: no hidden layers

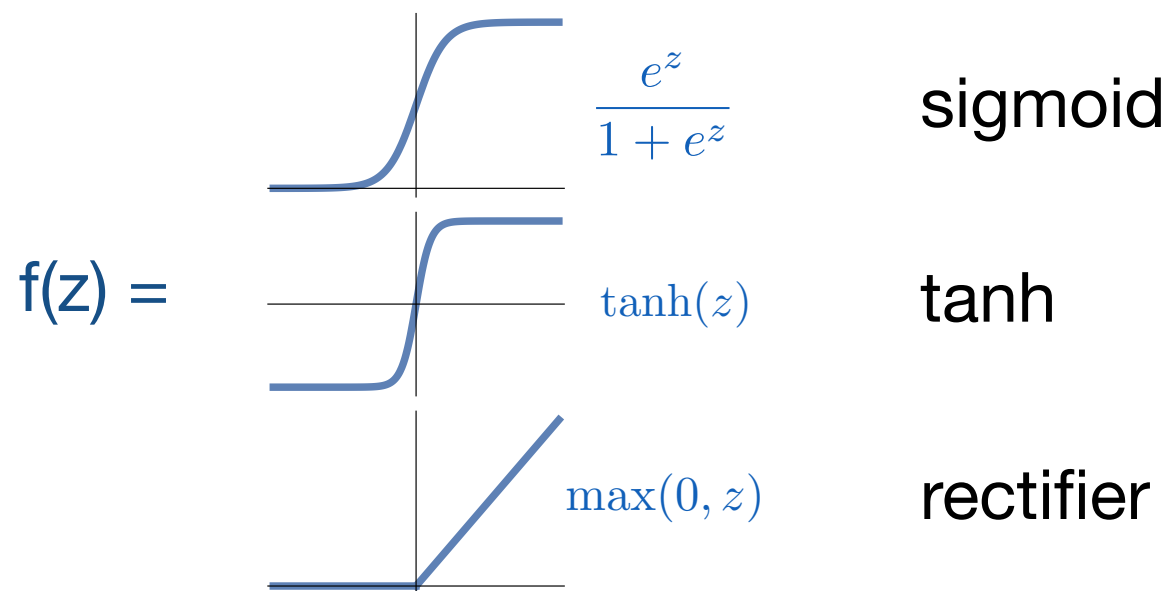


1-layer NN (no hidden layer, just one output layer and an input layer)

This network just aggregates its inputs, computed a weighted-sum adds a bias and outputs.

If we put $f(z) = \text{sgn}(z)$, this will become a linear classifier

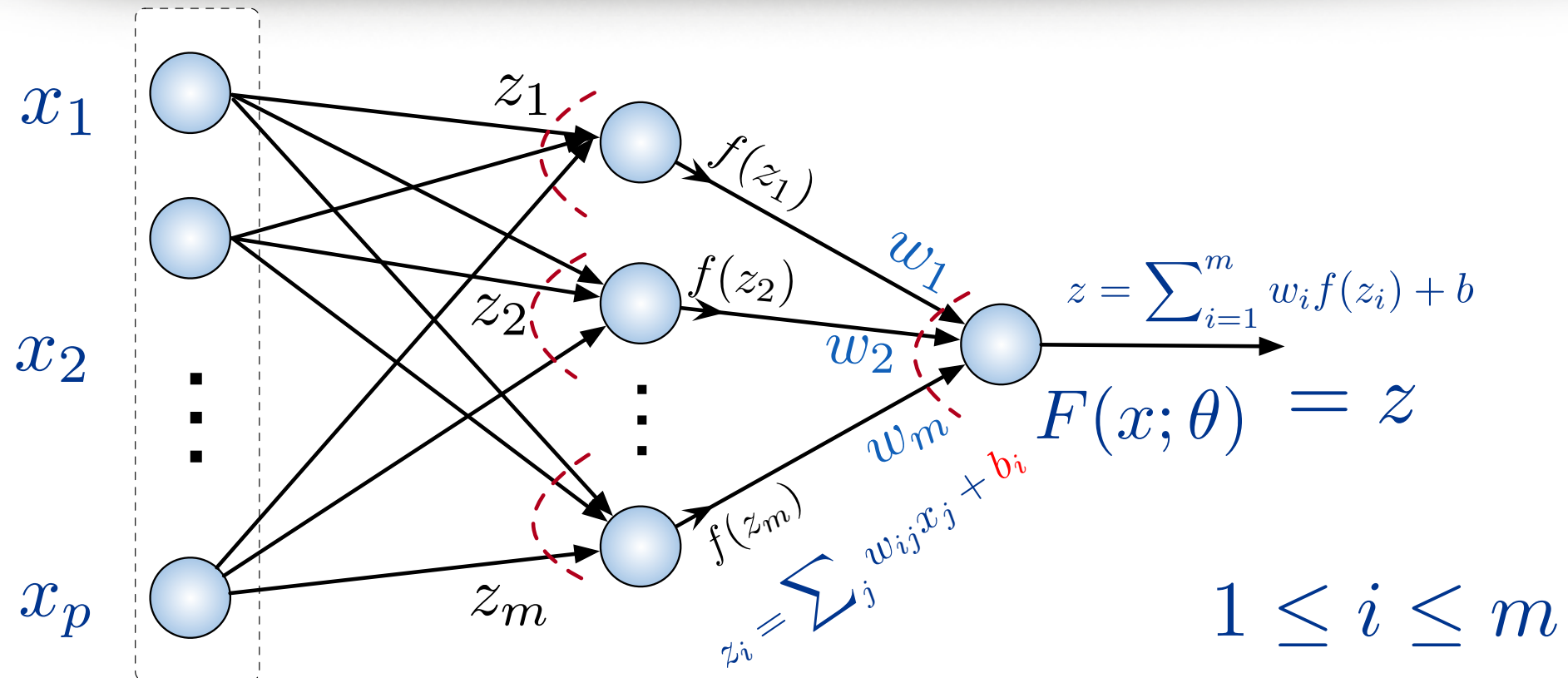
Activation functions



Henceforth, we'll assume that $f(z) = \max(0, z)$ is the activation function. This is called a **rectifier**, and a unit that uses this activation is called a Rectified Linear Unit (better known as **ReLU**).

Exercise: What governs what an activation function should be?

1 hidden layer network



Hidden unit ‘i’ computes its weighted input z_i by aggregating

$$z_i = \sum_j w_{ij} x_j + b_i$$

Ex: what is θ ?

Then it outputs the feature $f(z_i)$, i.e., it “activates”. We write

$$a_i = f(z_i)$$

Output unit does not see input data directly but operates on the new features

$$[f(z_1), \dots, f(z_m)]$$

How powerful are these models?

Answer: Seemingly omnipotent!

Theorem (informal): Let f be a sigmoid activation. Given **any** continuous function 'h' on a compact set C in \mathbb{R}^d , there exists a NN with 1 hidden layer and a choice of parameters such that the output

$$F(x) = \sum_{i=1}^m \nu_i f(w_i^T x + b_i)$$

approximates 'h' to any desired accuracy $\varepsilon > 0$, i.e., $|F(x) - h(x)| < \varepsilon$ for all x in C .

(This is Cybenko's result from 1989; one of many such "universality results" on NNs)

more generally

Hornik (1991) result is even more general: essentially, any bounded nonconstant activation allows 1-hidden layer net be a universal approximator

even more

Leshno et al. (1993). NN can approximate any continuous function arbitrarily well **if and only if** the activation function is **not** a polynomial

What's the catch?

Related work on representation; learnability

Recent interest

B. Hanin (Aug 2017). [*Universal Function Approximation by Deep Neural Nets with Bounded Width and ReLU Activations*](#)

M. Telgarsky (2016)— [*Benefits of depth in neural nets.*](#)

Petersen, Voigtländer (Sep. 2017). [*Optimal approximation of piecewise smooth functions using deep ReLU neural networks*](#)

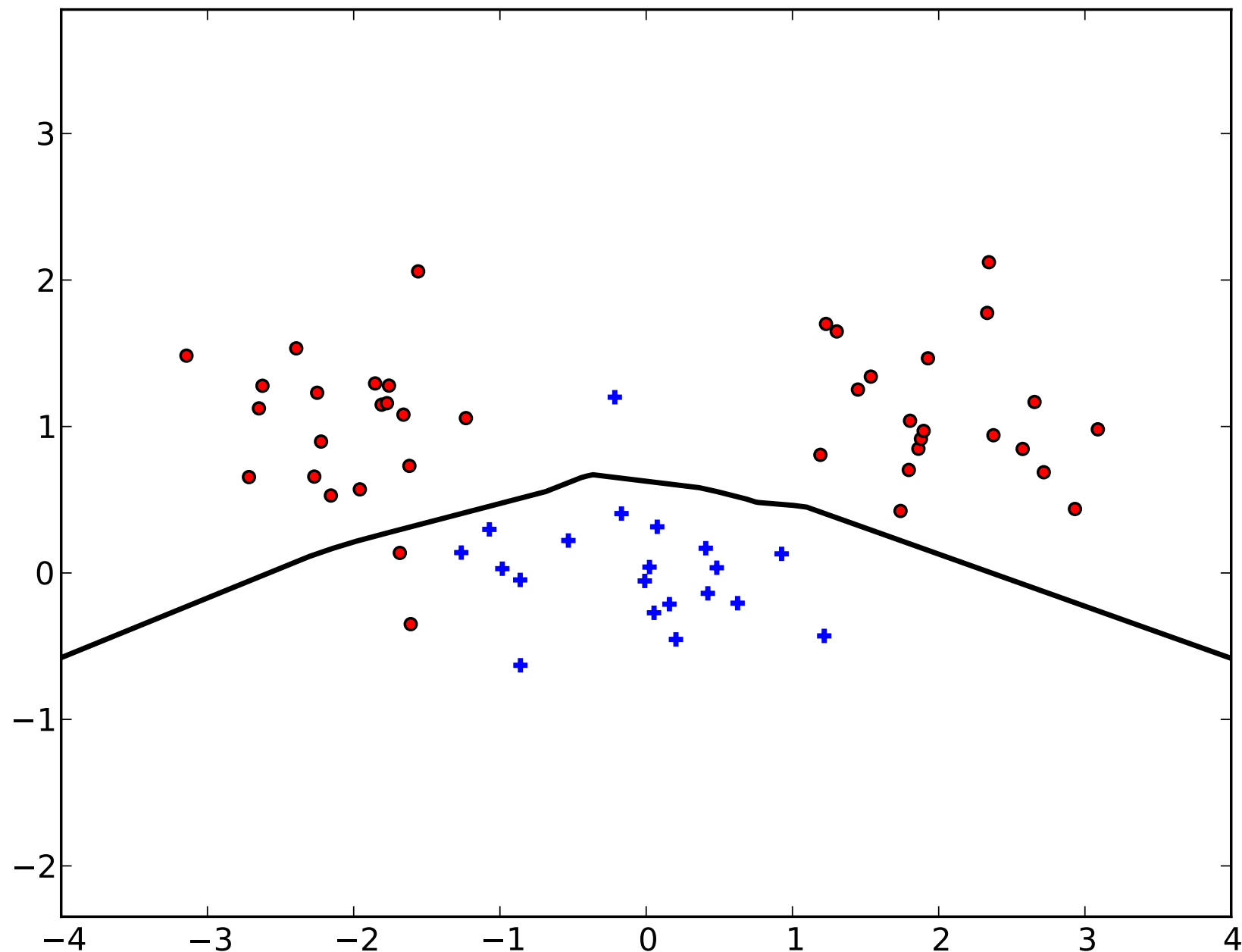
Negative results

S. Shalev-Shwartz et al. (ICML 2017). [*Failures of gradient based deep learning*](#)

Song et al. (2017). [*On the complexity of learning neural networks*](#)

And tons of other papers trying to understand specialized settings in which one can do learn the parameters, or some hypothesis class, and so on. Beyond the scope of our class; interested folks should chase above links

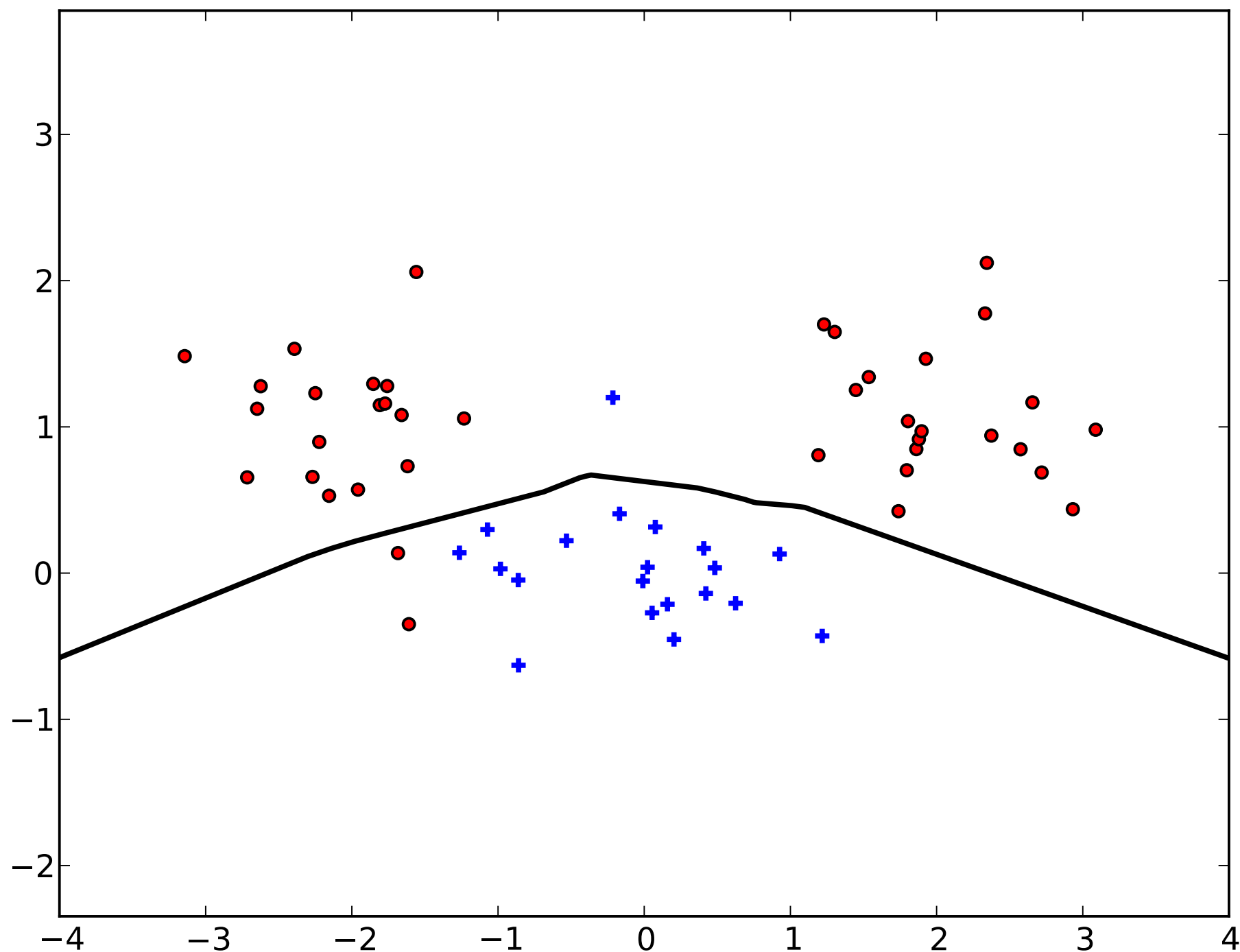
How powerful?



This data is separable using 2 hidden units (**why?**)

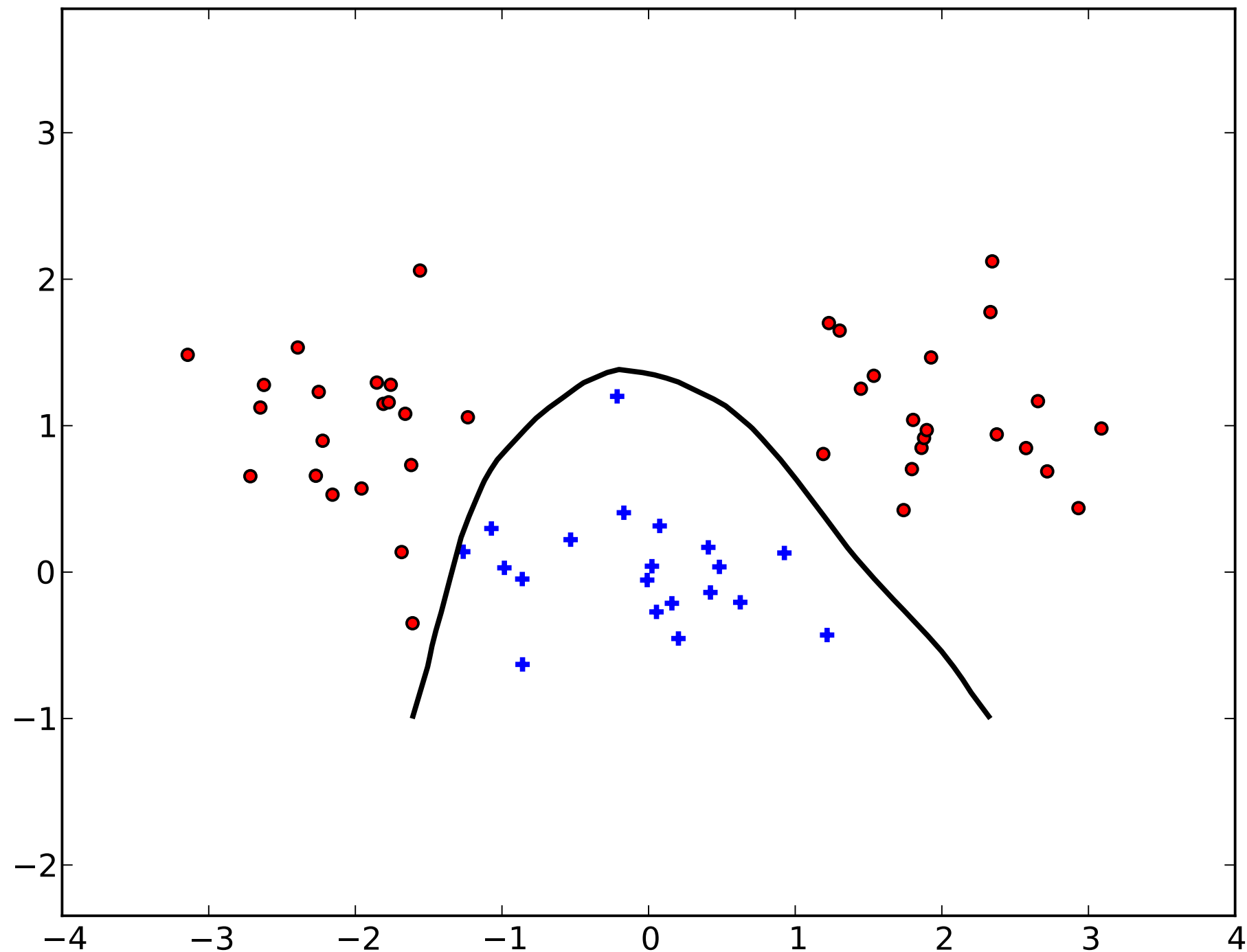
[Image credit: Tommi Jaakkola]

Network with 10 hidden neurons



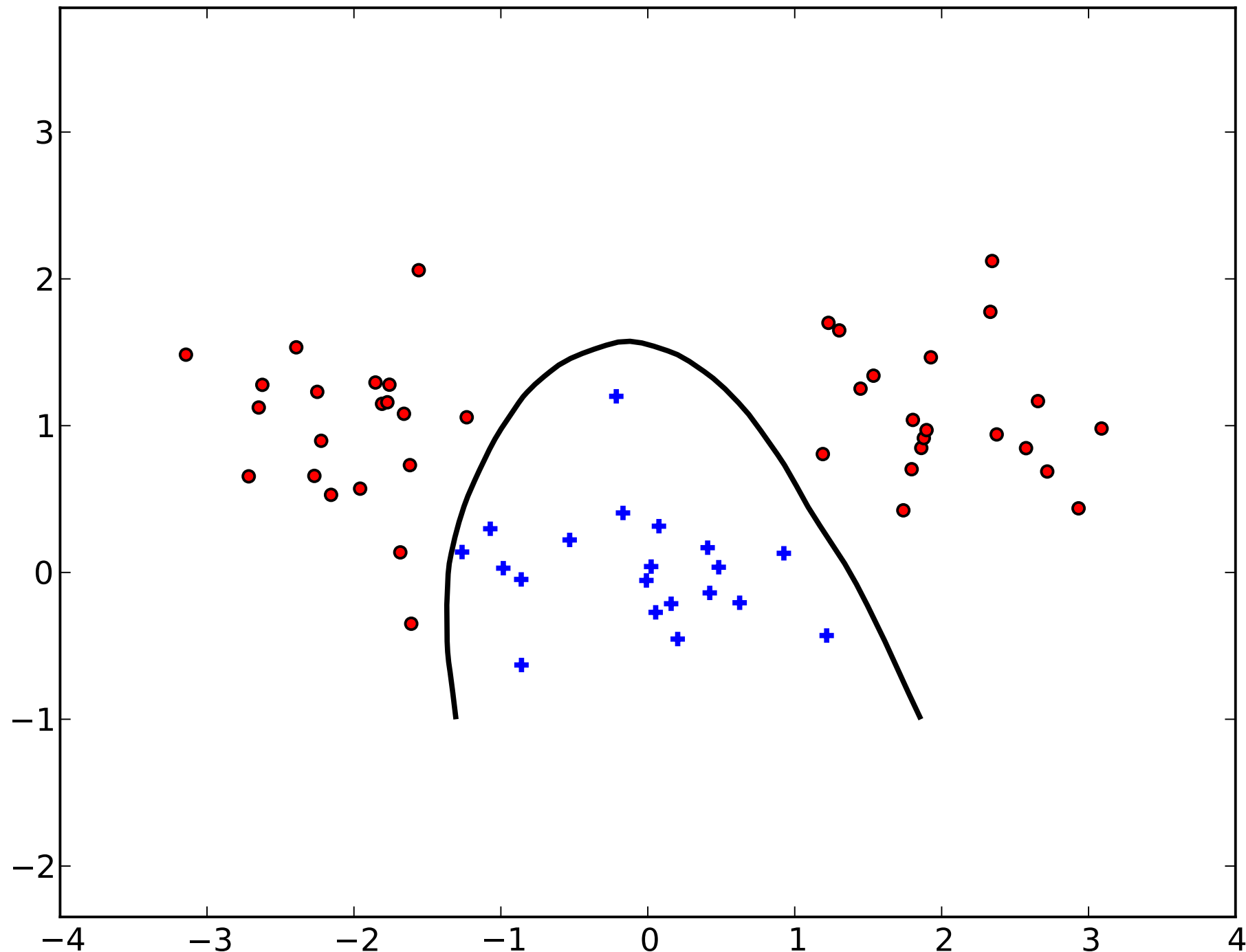
[Image credit: Tommi Jaakkola]

With 100 neurons



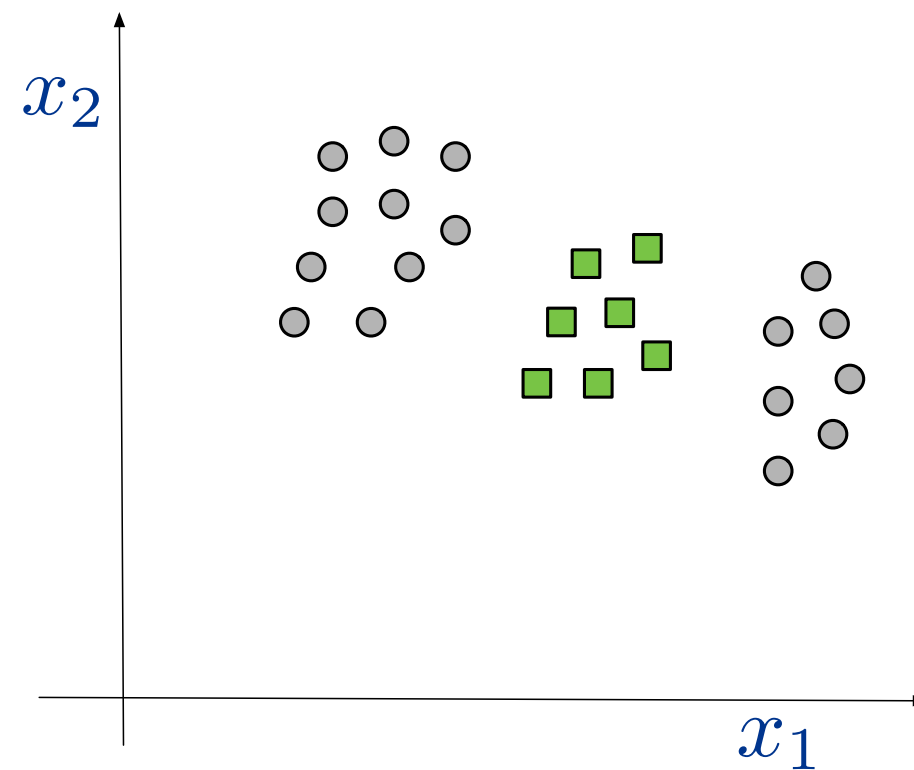
[Image credit: Tommi Jaakkola]

Now with 500 neurons



[Image credit: Tommi Jaakkola]

Nonlinear separation



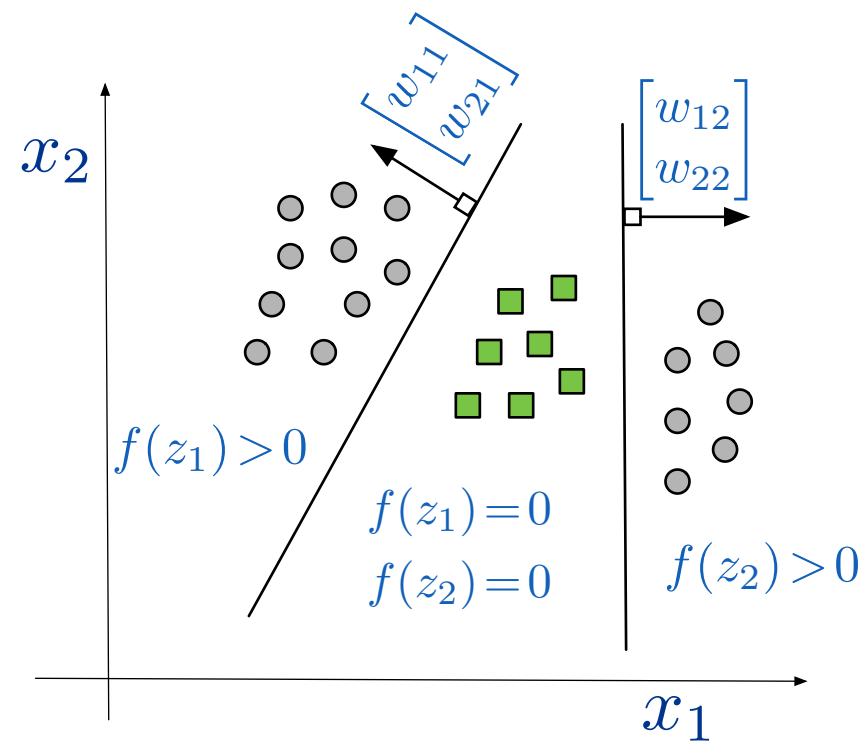
Nonlinearly separable points that we can try to classify with 2 units

$$z_1 = w_{11}x_1 + w_{21}x_2 + b_1, \quad (\text{1st unit})$$

$$z_2 = w_{12}x_1 + w_{22}x_2 + b_2, \quad (\text{1st unit})$$

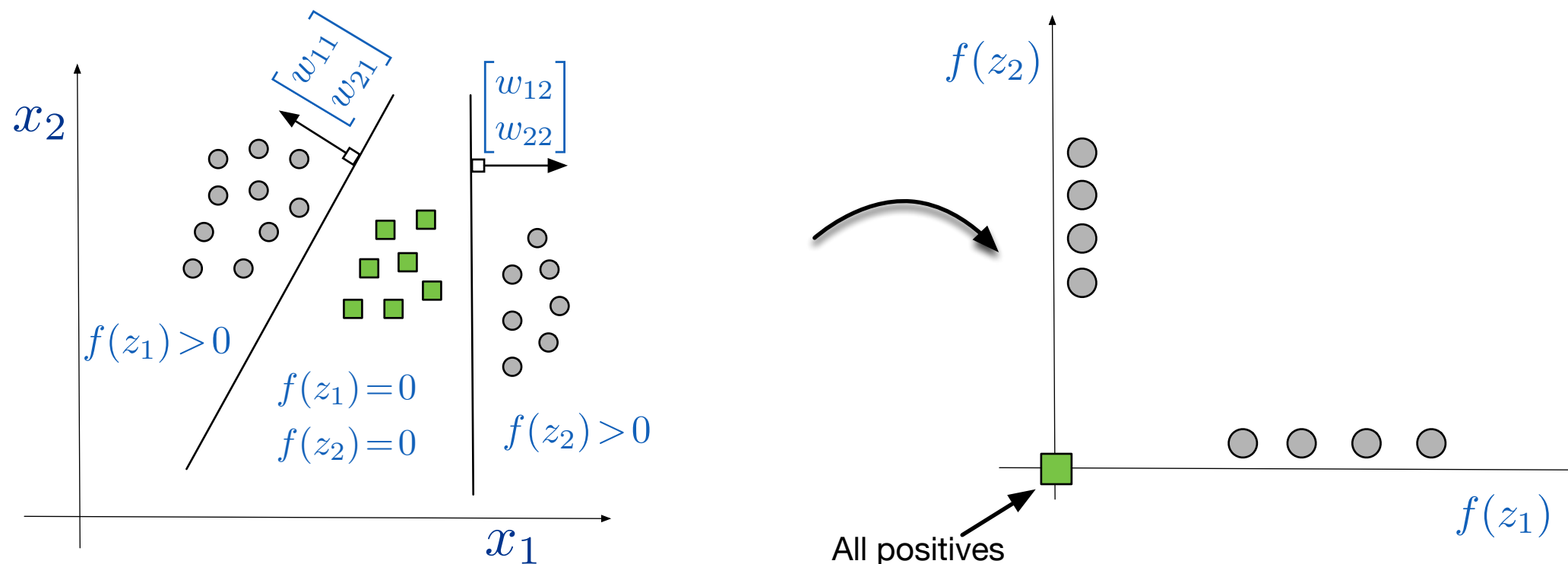
$$f(z_1) = \max(0, z_1), \quad f(z_2) = \max(0, z_2).$$

Nonlinear separation



Each input point with coordinates $x=(x_1,x_2)$ mapped into $[f(z_1),f(z_2)]$

Nonlinear separation



Points become easily linearly separable in the mapped space

Note: This example also indicates that it is challenging to learn the hidden representation.

Exercise: think about what would happen if we were to flip the signs of the the normal vectors shown in the plot: what features would we now learn?

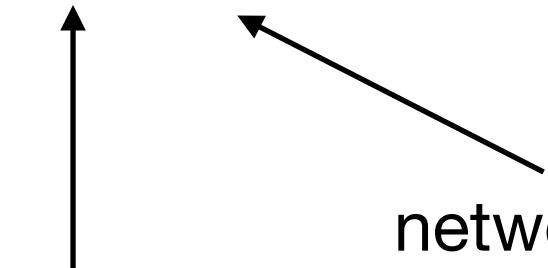
Think: Imagine now we had 'm' hidden units; how many such "sign flips"? which of those configurations may work / may not work?

Training a NN: ERM once again

$$\min_{\theta} R_N(\theta) := \frac{1}{N} \sum_{i=1}^N \ell(y_i, F(x_i; \theta)) + \lambda \|\theta\|^2$$

regularization

label network output



$$\ell(y, z) = \max(0, 1 - yz)$$

$$\ell(y, z) = \frac{1}{2}(y - z)^2$$

(several others exist, eg to handle multiclass)

SGD

$$\theta \leftarrow \theta - \eta \frac{\partial \ell(y, F(x; \theta))}{\partial \theta}$$

Notes:

In practice, SGD with momentum and gradient clipping used

Check out: <http://cs231n.github.io/neural-networks-3/#sgd> for good summary

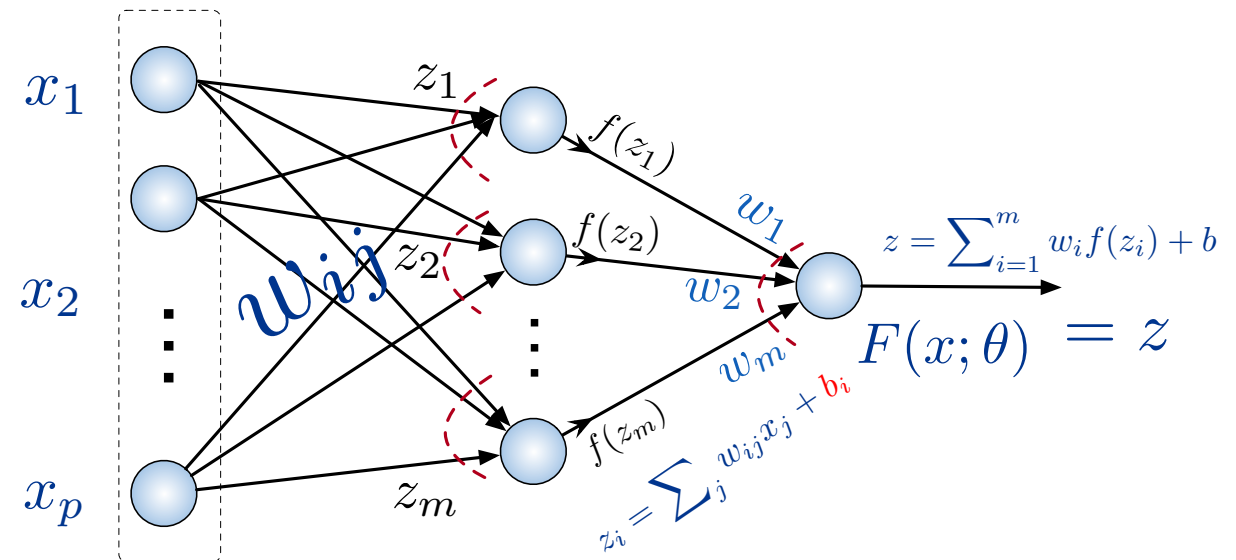
Also useful: <http://runder.io/optimizing-gradient-descent/index.html>

Computing a stochastic gradient

Key computational challenge: compute a stochastic gradient

Consider this NN with 1 hidden layer

$$w_{ij} \quad \begin{array}{l} 1 \leq i \leq m \text{ (hidden units)} \\ 1 \leq j \leq p \text{ (input features)} \end{array}$$



$$z_i = \sum_{j=1}^p w_{ij} x_j + b_i$$

$$f(z_i) = \max(0, z_i)$$

$$z = \sum_{i=1}^m w_i f(z_i) + b$$

$$f(z) = F(x; \theta) = z$$

$$\ell(y, z) = \max(0, 1 - yz)$$

input to i -th hidden unit

output of i^{th} hidden unit

input to output unit

network output

Aim: compute

$$\partial \ell / \partial \theta$$

Computing a stochastic gradient

$$z_i = \sum_{j=1}^p w_{ij} x_j + b_i \quad \text{input to } i\text{-th hidden unit}$$

$$f(z_i) = \max(0, z_i) \quad \text{output of } i^{\text{th}} \text{ hidden unit}$$

$$z = \sum_{i=1}^m w_i f(z_i) + b \quad \text{input to output unit}$$

$$f(z) = F(x; \theta) = z \quad \text{network output}$$

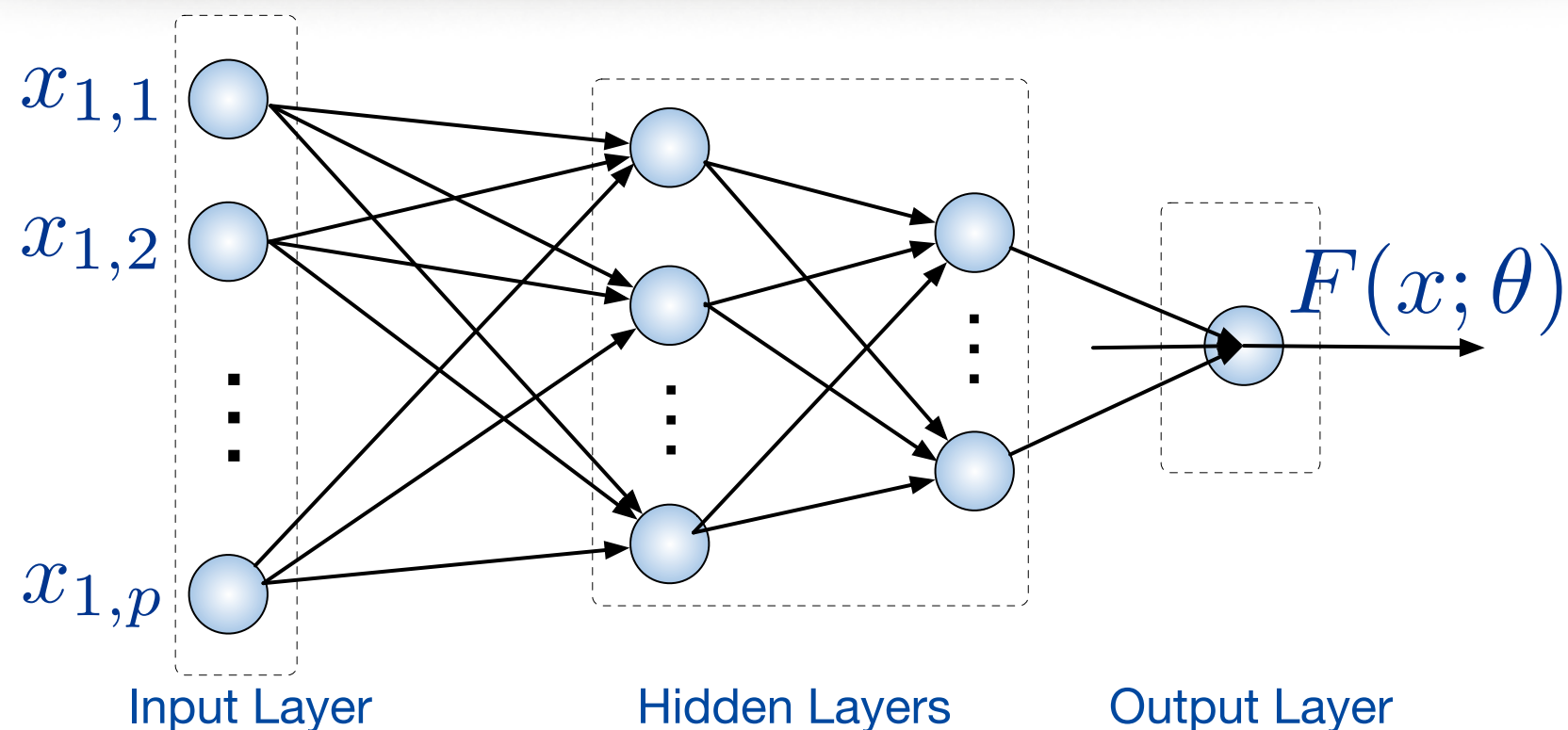
$$\ell(y, z) = \max(0, 1 - yz)$$

Appeal to the chain-rule of calculus: observe that a change to w_{ij} changes z_i , which changes $f(z_i)$, which eventually changes z and thus the loss ℓ .

Chain-rule of calculus

$$\begin{aligned} \frac{\partial \ell(y, z)}{\partial w_{ij}} &= \left[\frac{\partial z_i}{\partial w_{ij}} \right] \left[\frac{\partial f(z_i)}{\partial z_i} \right] \left[\frac{\partial z}{\partial f(z_i)} \right] \frac{\partial \ell}{\partial z} \\ &= [x_j] \mathbb{I}[z_i > 0] [w_i] \begin{cases} -y, & \text{if } \ell(y, z) > 0, \\ 0, & \text{otherwise.} \end{cases} \end{aligned}$$

Computing gradients: the challenge



Challenge: How to apply the chain rule?

- * A change to a weight w_{ij} at the first hidden layer will impact all subsequent layers.
- * To apply the chain-rule, must aggregate contribution from each unit to final output
- * We must chase all paths by which information can flow from first layer to last!
- * This is clearly a combinatorial explosion, not good.

Exercise: Why not do finite-differences $[F(\theta + \epsilon e_i) - F(\theta)] / \epsilon$ for some small ϵ ?

Backpropagation

A simple, brilliant idea dating back to 1960s, and early 70s

Rediscovered multiple time; popularized greatly after 1986 paper of Rumelhart, Hinton, Williams

Key insight: Trade space for time (dynamic programming).

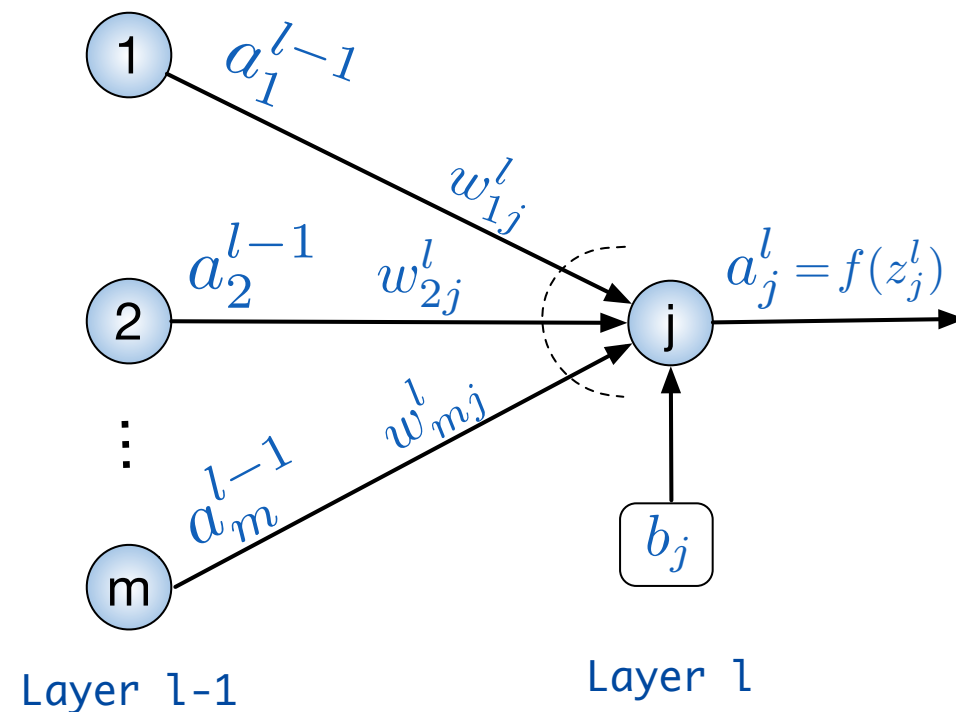
Idea: Keep track of how a change to the input of one layer impacts its output, and **use extra storage to save that**. Possibly “chain” these savings to obtain the combined change to the loss function: in other words, a gradient of the loss function wrt change in network parameters

Observe: In a simple architecture, a layer does not depend on what comes after it but only on what comes before it.

(More generally, we can think of a Directed Acyclic Graph describing a network architecture)

Backpropagation

Suppose our network has layers 1, 2, ..., L; layer 1: input; layer L: output



$$a^l := (a_1^l, \dots, a_{m'}^l), \quad (m' \text{ units in layer } l)$$

$$z^l := (W^l)^T a^{l-1} + b^l$$

$$a^l = f(z^l).$$

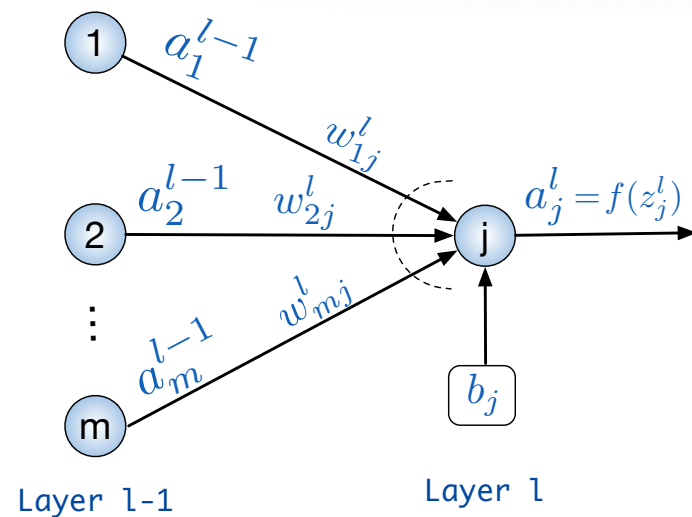
a^l, b^l, z^l vectors
 W^l weight matrix

Connection of layer $l-1$ to neuron 'j' in layer l

Aim: compute $\partial \ell / \partial \theta$

Let us consider $\partial \ell / \partial W^l$

Backpropagation



$$a^l := (a_1^l, \dots, a_{m'}^l), \quad (m' \text{ units in layer } l)$$

$$z^l := (W^l)^T a^{l-1} + b^l$$

$$a^l = f(z^l).$$

$$a^l, b^l, z^l$$

$$W^l$$

Extra storage

$$\frac{\partial \ell}{\partial W^l} = \frac{\partial z^l}{\partial W^l} \left[\frac{\partial \ell}{\partial z^l} \right] = \frac{\partial z^l}{\partial W^l} (\delta^l)^T$$

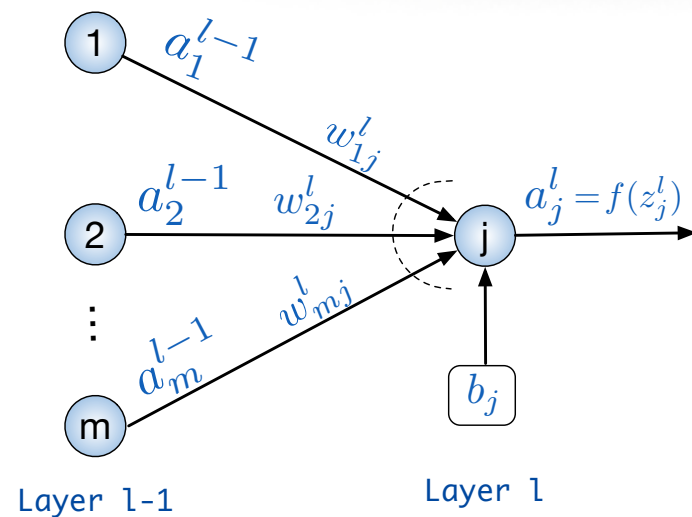
a^{l-1} →

The crucial idea:
Keep track layerwise,
how loss varies with
inputs each layer.

Since we store activations, we have access to the first part, what about the δ term?

As we observed before, a previous layer impacts the next one, not vice-versa. So as z^l changes, it impacts z^{l+1}

Backpropagation



$$a^l := (a_1^l, \dots, a_{m'}^l), \quad (m' \text{ units in layer } l)$$

$$z^l := (W^l)^T a^{l-1} + b^l$$

$$a^l = f(z^l).$$

$$a^l, b^l, z^l$$

$$W^l$$

Extra storage

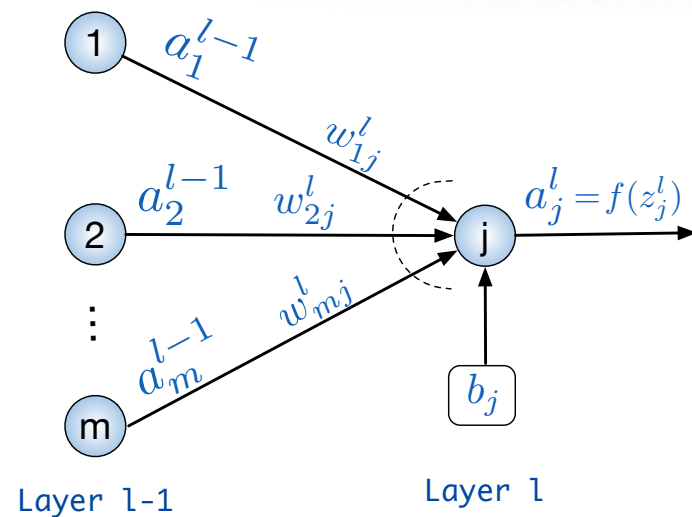
$$\frac{\partial \ell}{\partial W^l} = \frac{\partial z^l}{\partial W^l} \left[\frac{\partial \ell}{\partial z^l} \right] = \frac{\partial z^l}{\partial W^l} (\delta^l)^T$$

a^{l-1} \nearrow

As we observed before, a previous layer impacts the next one, not vice-versa. So as z^l changes, it impacts z^{l+1} . This is the most important substep!

$$\delta^l = \frac{\partial \ell}{\partial z^l} = \frac{\partial z^{l+1}}{\partial z^l} \cdot \frac{\partial \ell}{\partial z^{l+1}} = \frac{\partial z^{l+1}}{\partial z^l} \cdot \delta^{l+1}$$

Backpropagation



$$a^l := (a_1^l, \dots, a_{m'}^l), \quad (m' \text{ units in layer } l)$$

$$z^l := (W^l)^T a^{l-1} + b^l$$

$$a^l = f(z^l).$$

$$a^l, b^l, z^l$$

$$W^l$$

As we observed before, a previous layer impacts the next one, not vice-versa. So as z^l changes, it impacts z^{l+1}

$$\delta^l = \frac{\partial \ell}{\partial z^l} = \frac{\partial z^{l+1}}{\partial z^l} \cdot \frac{\partial \ell}{\partial z^{l+1}} = \frac{\partial z^{l+1}}{\partial z^l} \cdot \delta^{l+1}$$

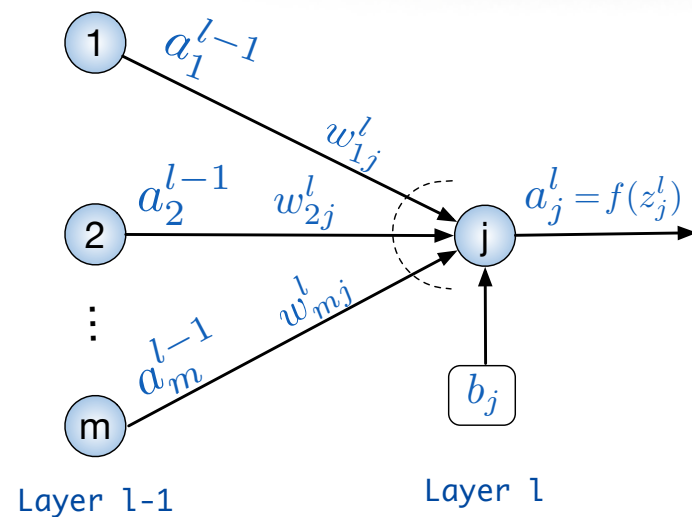
If we could somehow handle this term, smells like progress, right?



But hey!

$$z^{l+1} = (W^{l+1})^T a^l + b^{l+1} = (W^{l+1})^T f(z^l) + b^{l+1}$$

Backpropagation



$$a^l := (a_1^l, \dots, a_{m'}^l), \quad (m' \text{ units in layer } l)$$

$$z^l := (W^l)^T a^{l-1} + b^l$$

$$a^l = f(z^l).$$

$$a^l, b^l, z^l$$

$$W^l$$

As we observed before, a previous layer impacts the next one, not vice-versa. So as z^l changes, it impacts z^{l+1}

$$\delta^l = \frac{\partial \ell}{\partial z^l} = \frac{\partial z^{l+1}}{\partial z^l} \cdot \frac{\partial \ell}{\partial z^{l+1}} = \frac{\partial z^{l+1}}{\partial z^l} \cdot \delta^{l+1}$$

storage

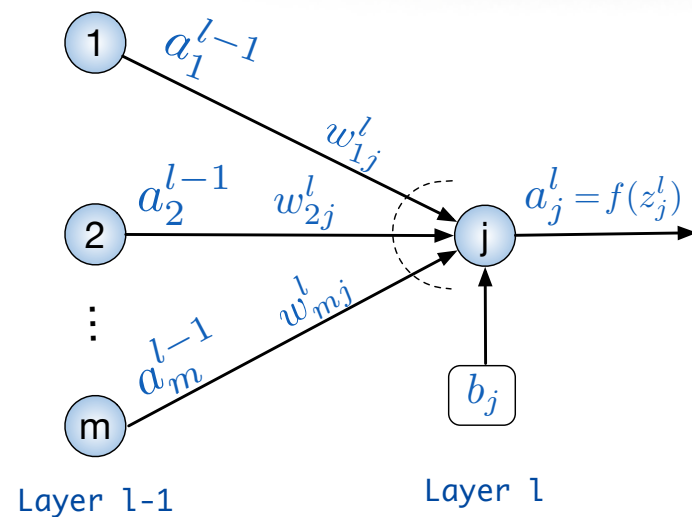
But hey! $z^{l+1} = (W^{l+1})^T a^l + b^{l+1} = (W^{l+1}) f(z^l) + b^{l+1}$

$$\frac{\partial z^{l+1}}{\partial z^l} = \text{Diag}[f'(z^l)] W^{l+1}$$

Great? $\delta^l = \frac{\partial \ell}{\partial z^l} = \text{Diag}[f'(z^l)] W^{l+1} \delta^{l+1}$

Recursive identity

Backpropagation



$$a^l := (a_1^l, \dots, a_{m'}^l), \quad (m' \text{ units in layer } l)$$

$$z^l := (W^l)^T a^{l-1} + b^l$$

$$a^l = f(z^l).$$

$$a^l, b^l, z^l$$

$$W^l$$

Great? $\delta^l = \frac{\partial \ell}{\partial z^l} = \text{Diag}[f'(z^l)] W^{l+1} \delta^{l+1}.$

$$\delta^l = \text{Diag}[f'(z^l)] W^{l+1} \text{Diag}[f'(z^{l+1})] W^{l+2} \dots W^L \delta^L$$

Are we done?

Also, now it is clear, why we call it “backprop”

Yes, we are because δ^L is trivial to compute

Verify $\delta^L = \frac{\partial \ell}{\partial z^L} = \frac{\partial \ell}{\partial a^L} \frac{\partial a^L}{\partial z^L} = \text{Diag}[f'(z^L)] \frac{\partial \ell}{\partial a^L}$

Backprop, pseudocode

1. **Input:** (x, y) set activations for a^1 input layer

2. **Feedforward:**

for each layer $l = 2, \dots, L$ **do**

$$z^l = (W^l)^T a^{l-1} + b^l$$

$$a^l = f(z^l)$$

3. **Output layer:** $\delta^L = \mathcal{D}[f'(z^L)] \nabla_a \text{loss}$

4. **Backpropagation:**

for each $l = L - 1, \dots, 2$ **do**

$$\delta^l = \mathcal{D}[f'(z^l)] W^{l+1} \delta^{l+1}$$

5. **Final gradients:**

$$\frac{\partial \text{loss}}{\partial W^l} = a^{l-1} \delta^l, \quad \frac{\partial \text{loss}}{\partial b^l} = \delta^l$$

Automatic differentiation

Forward mode

Backward mode (Backprop a special case)

Can be expensive; checkpointing tricks with repeated re-evaluation to speedup

Optimal Jacobian Accumulation: NP-Complete

Complexity unknown if it is assumed that all edge labels unique and independent

Check out the Wikipedia page for a rough intro

Tensorflow, et al : build static compute graphs

PyTorch, Mxnet: dynamic computation graphs

All have tutorials going in further detail about autodiff