Neural Networks

Wei Xu

(many slides from Greg Durrett and Philipp Koehn)

This Lecture

- Neural network history
- Neural network basics
- Feedforward neural networks + backpropagation
- Applications
- Implementing neural networks (if time)

A Bit of History

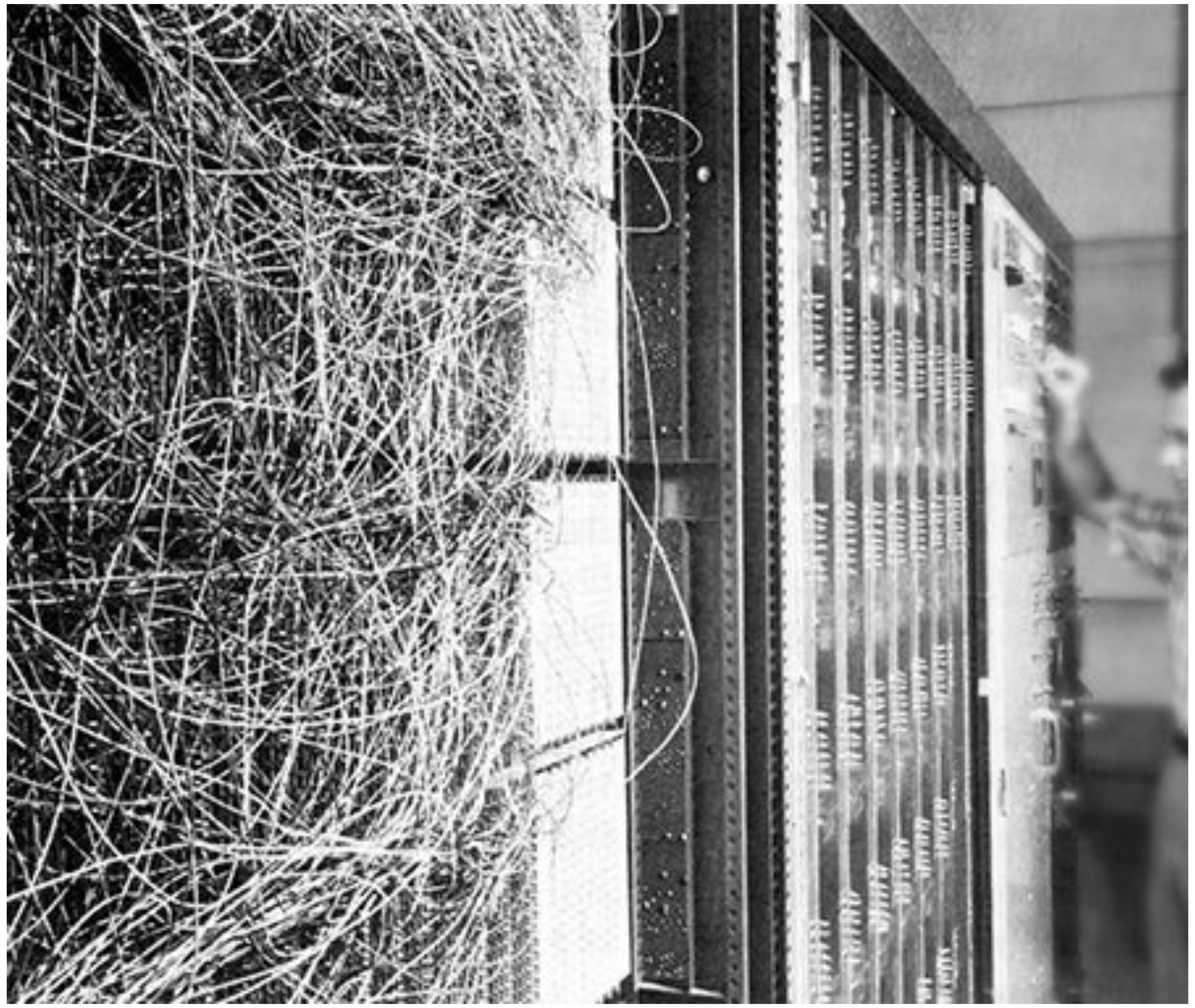
- The **Mark I Perceptron** machine was the first implementation of the perceptron algorithm.
- Perceptron (Frank Rosenblatt, 1957)
- Artificial Neuron (McCulloch & Pitts, 1943)

McCulloch Pitts Neuron
(assuming no inhibitory inputs)

$$y = 1 \quad if \sum_{i=0}^{n} x_i \geq 0$$
$$= 0 \quad if \sum_{i=0}^{n} x_i < 0$$

Perceptron

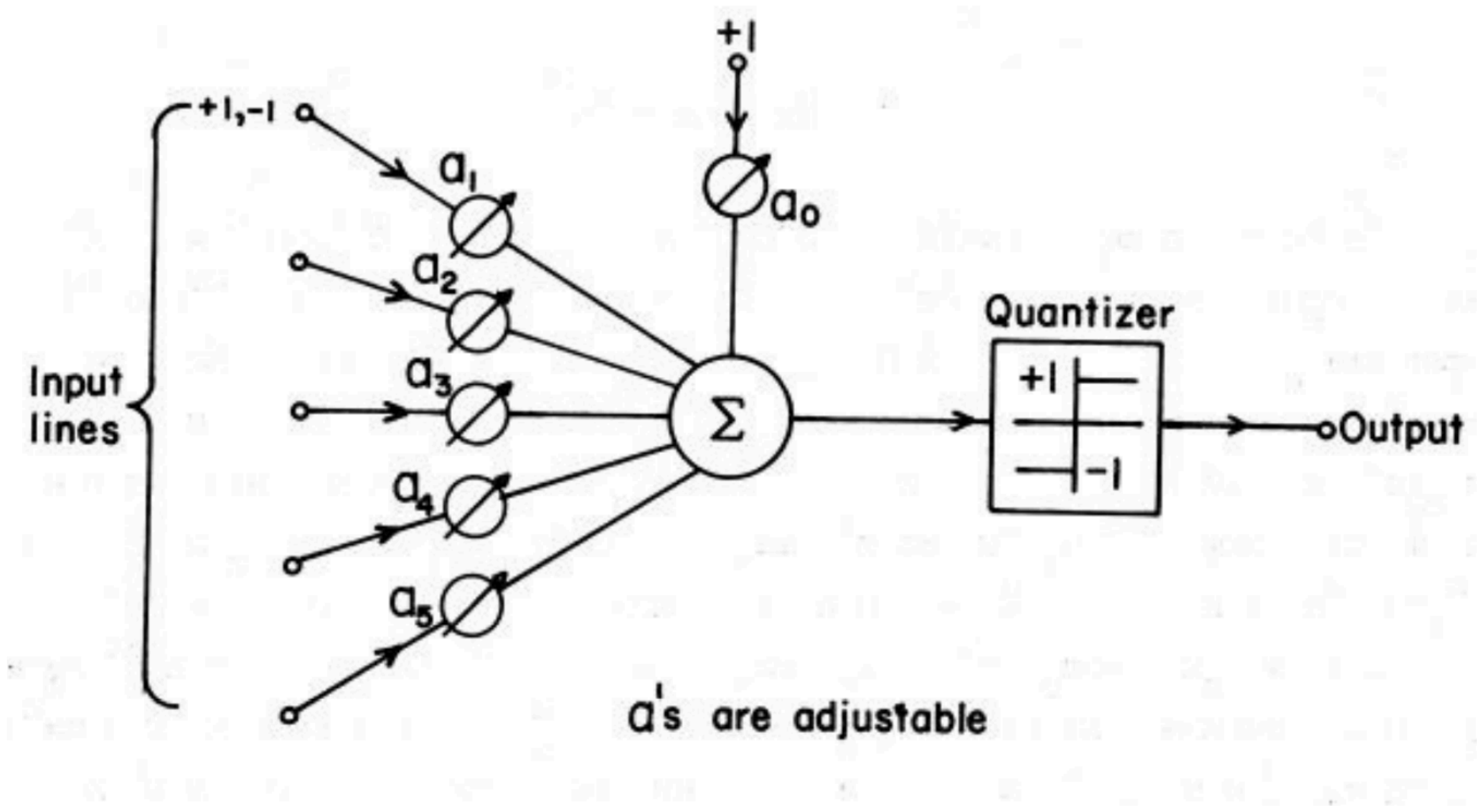
$$y = 1 \quad if \sum_{i=0}^{n} w_i * x_i \geq 0$$
$$= 0 \quad if \sum_{i=0}^{n} w_i * x_i < 0$$

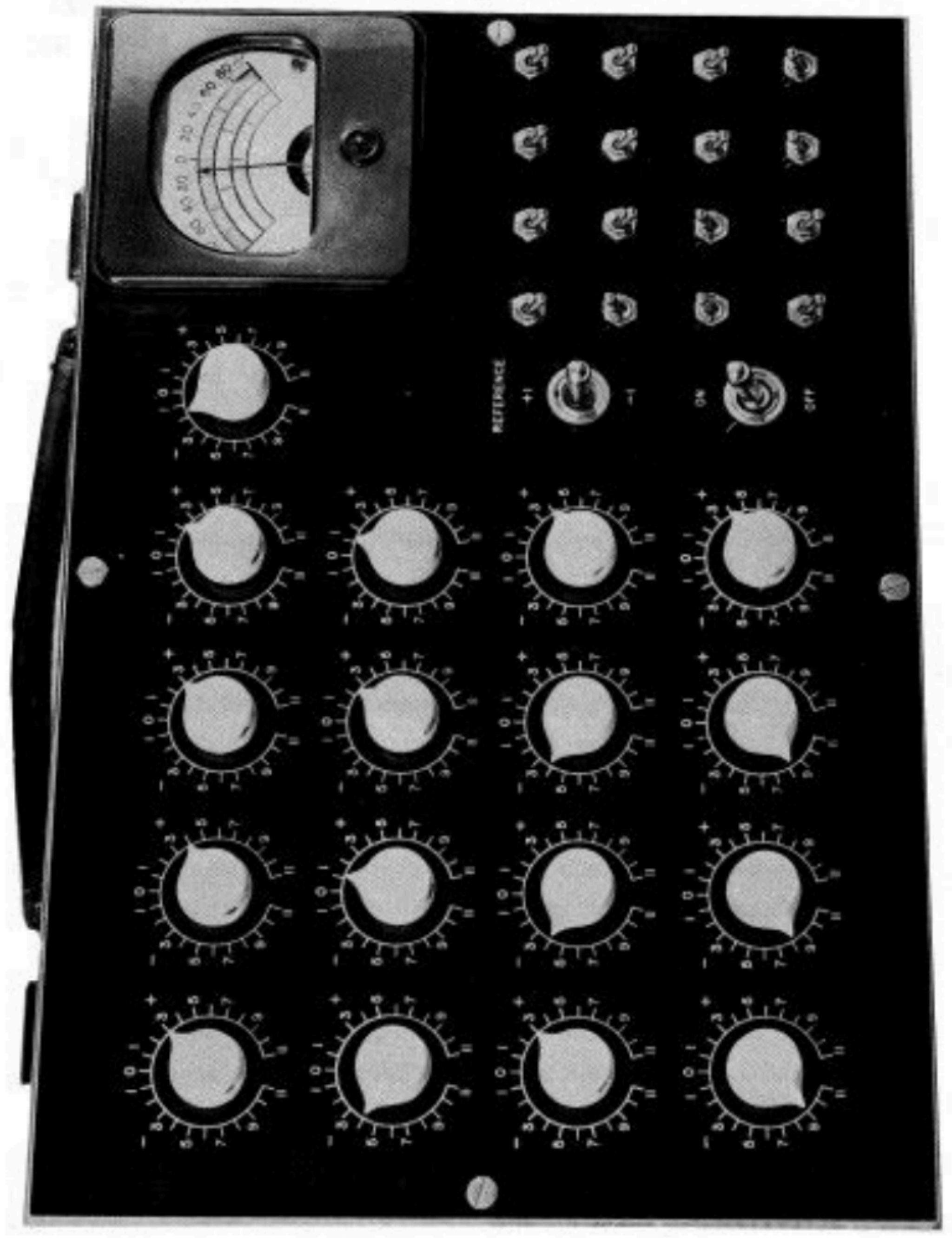


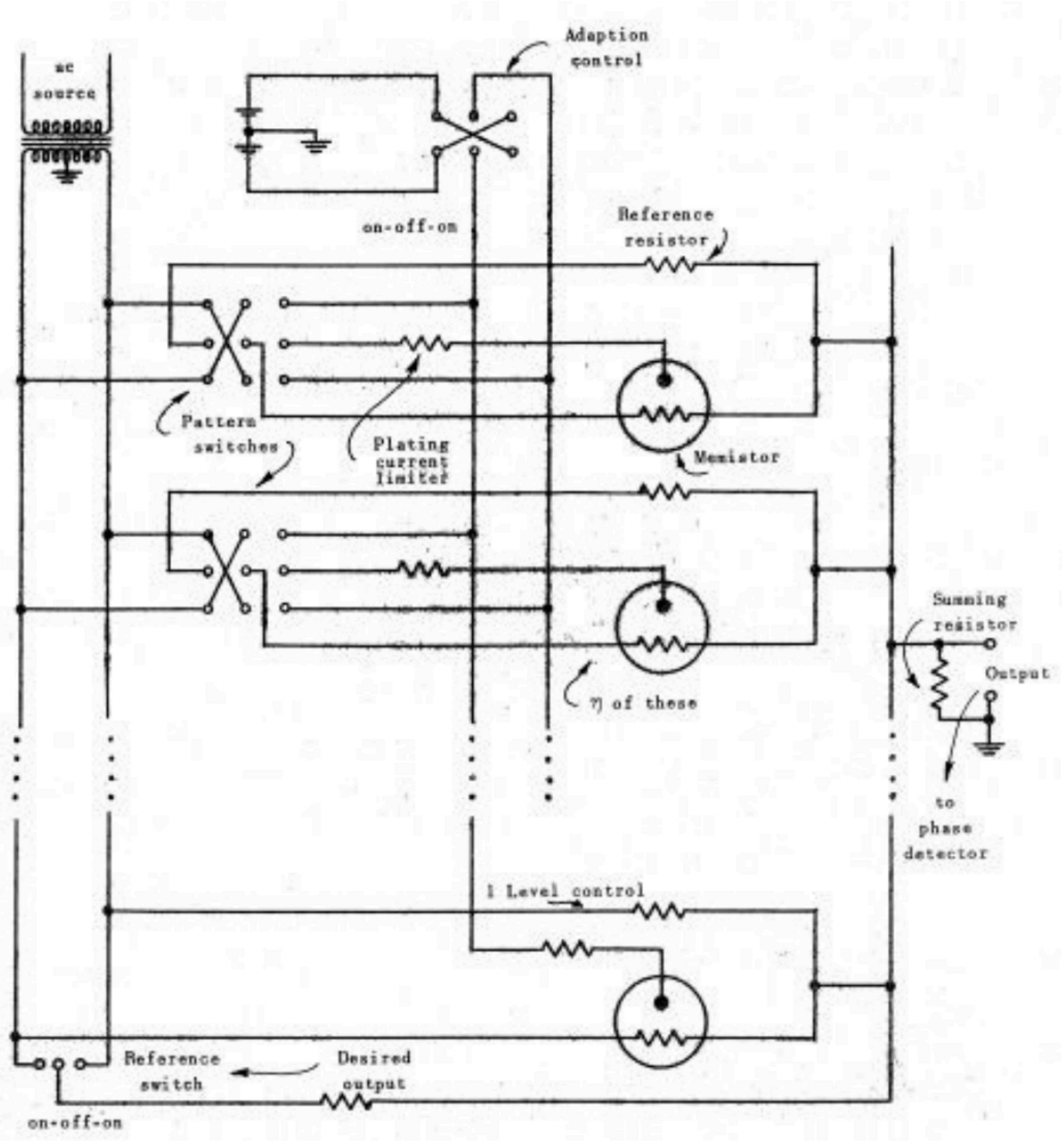
The IBM Automatic Sequence Controlled Calculator, called Mark I by Harvard University's staff. It was designed for image recognition: it had an array of 400 photocells, randomly connected to the "neurons". Weights were encoded in potentiometers, and weight updates during learning were performed by electric motors.
https://www.youtube.com/watch?time_continue=71&v=cNxadbrN_aI&feature=emb_logo

A Bit of History

- Adaline/Madeline - single and multi-layer “artificial neurons” (Widrow and Hoff, 1960)







A Bit of History

- First time back-propagation became popular (Rumbelhart et al, 1986)

Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton† & Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure[1].

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired output vectors[2]. Learning becomes more interesting but more difficult when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input–output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, x_j, to unit j is a linear function of the outputs, y_i, of the units that are connected to j and of the weights, w_{ji}, on these connections

$$x_j = \sum_i y_i w_{ji} \tag{1}$$

Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

A unit has a real-valued output, y_j, which is a non-linear function of its total input

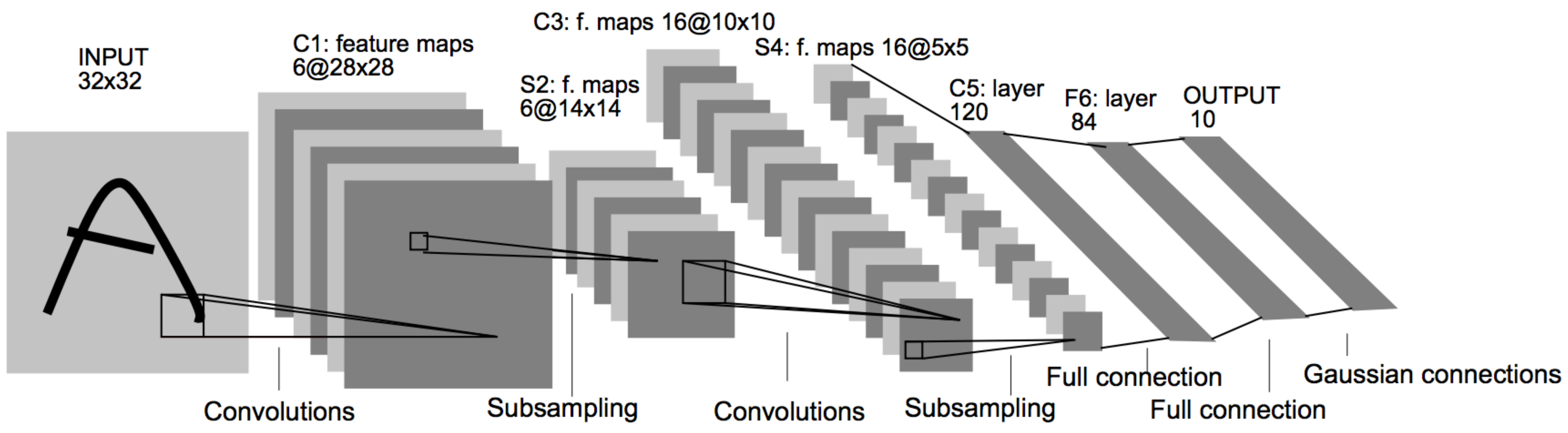
$$y_j = \frac{1}{1+e^{-x_j}} \tag{2}$$

† To whom correspondence should be addressed.

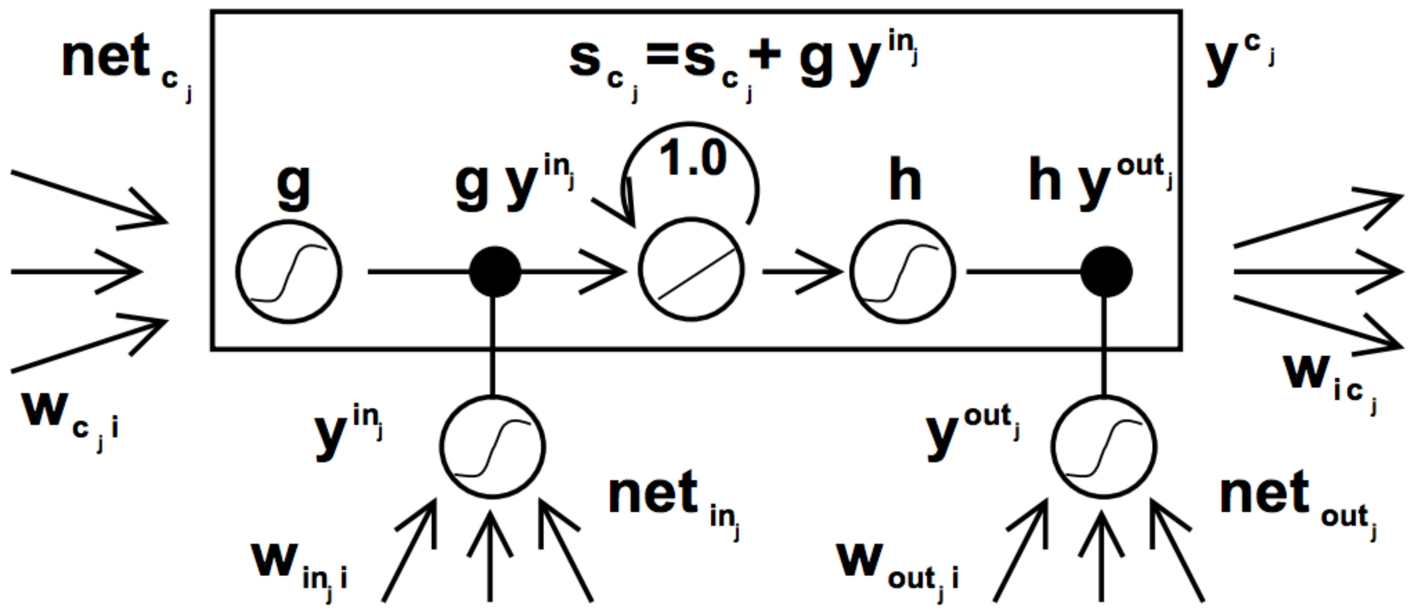
©1986 Nature Publishing Group

History: NN "dark ages"

- ConvNets: applied to MNIST by LeCun in 1998



- LSTMs: Hochreiter and Schmidhuber (1997)

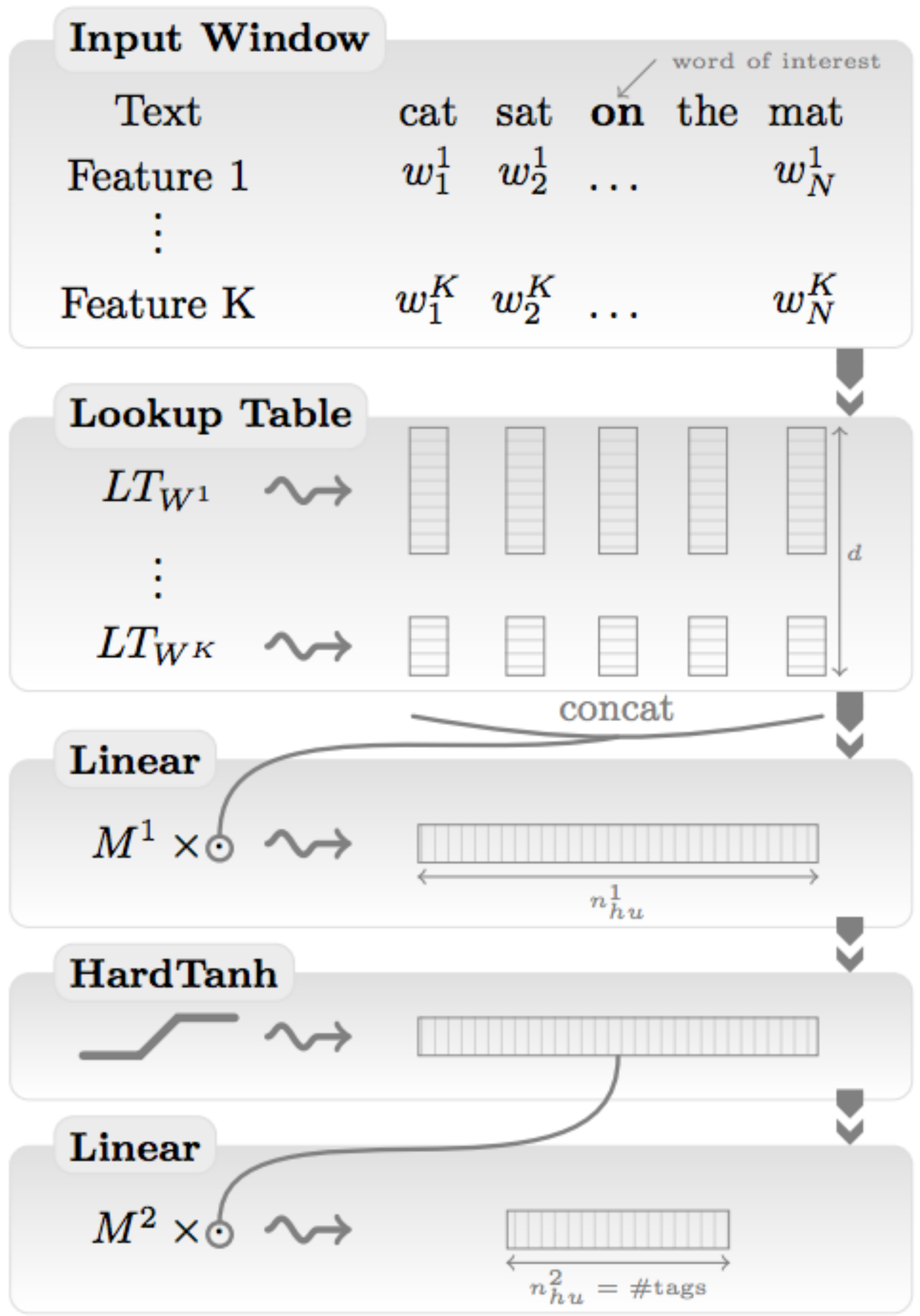


- Henderson (2003): neural shift-reduce parser, not SOTA

https://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning/
https://www.youtube.com/watch?v=FwFduRA_L6Q&feature=youtu.be

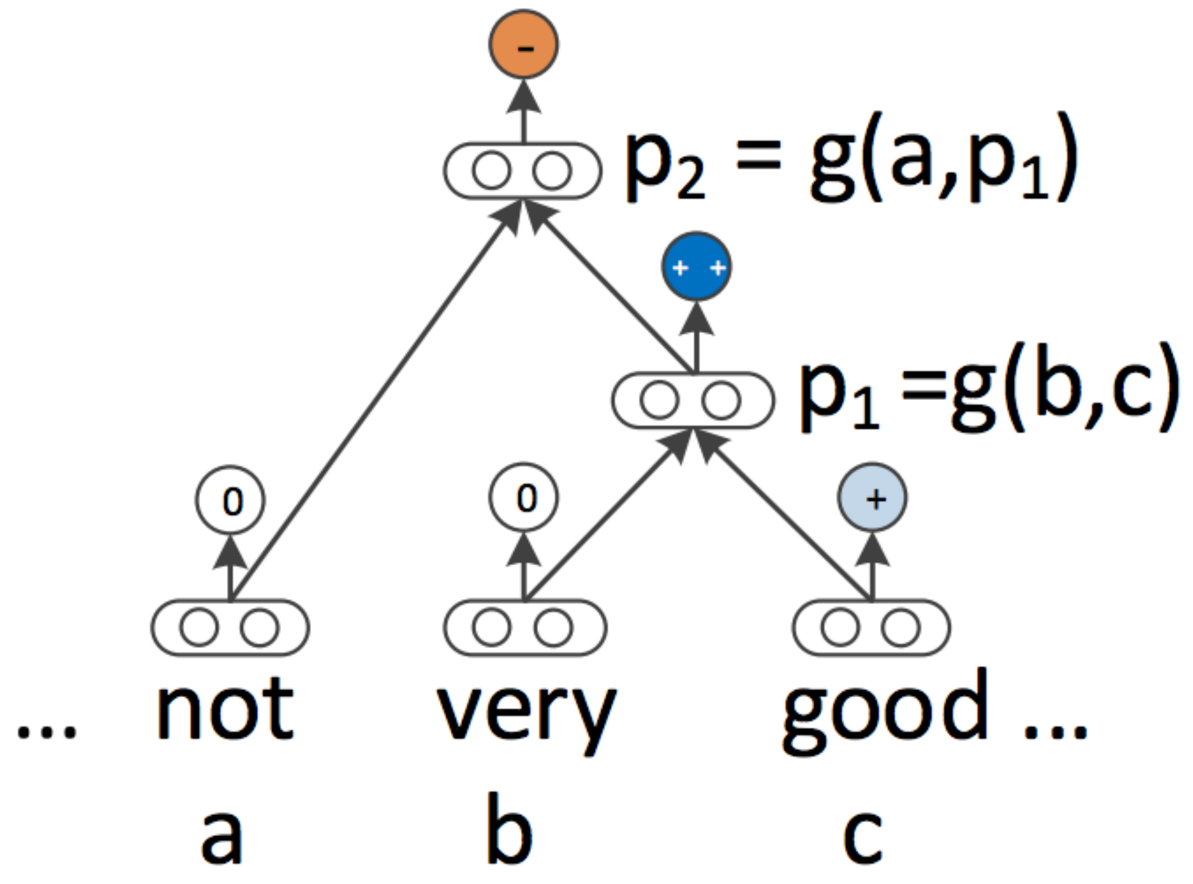
2008-2013: A glimmer of light…

- Collobert and Weston 2011: "NLP (almost) from scratch"
  - Feedforward neural nets induce features for sequential CRFs ("neural CRF")
  - 2008 version was marred by bad experiments, claimed SOTA but wasn't, 2011 version tied SOTA



- Krizhevskey et al. (2012): AlexNet for vision

- Socher 2011-2014: tree-structured RNNs working okay



2014: Stuff starts working

- Kim (2014) + Kalchbrenner et al. (2014): sentence classification / sentiment (convnets work for NLP?)
- Sutskever et al. (2014) + Bahdanau et al. (2015) : seq2seq + attention for neural MT (LSTMs work for NLP?)
- Chen and Manning (2014) transition-based dependency parser (even feedforward networks work well for NLP?)
- 2015: explosion of neural nets for everything under the sun

Why didn't they work before?

- **Datasets too small**: for MT, not really better until you have 1M+ parallel sentences (and really need a lot more)
- **Optimization not well understood**: good initialization, per-feature scaling + momentum (AdaGrad / AdaDelta / Adam) work best out-of-the-box
  - **Regularization**: dropout is pretty helpful
  - **Computers not big enough**: can't run for enough iterations
- **Inputs**: need word representations to have the right continuous semantics
- **Libraries**: TensorFlow (Nov 2015), PyTorch (Sep 2016)

Neural Networks

Wei Xu

(many slides from Greg Durrett and Philipp Koehn)

Administrivia

- Problem Set 1 is due on 2/3
- Reading: Eisenstein 2.6, 3.1-3.3, J+M 7, Goldberg 1-4
- TA also has released Project 1
- PyTorch Tutorial can also be found on the course project

This Lecture

- Neural network history (last class)
- Neural network basics
- Feedforward neural networks + backpropagation
- Applications
- Implementing neural networks (if time)

Neural Net Basics

Neural Networks: motivation

- Linear classification: $\text{argmax}_y w^\top f(x, y)$

- How can we do nonlinear classification? Kernels are too slow…

- Want to learn intermediate conjunctive features of the input

the movie was ***not*** *all that* ***good***

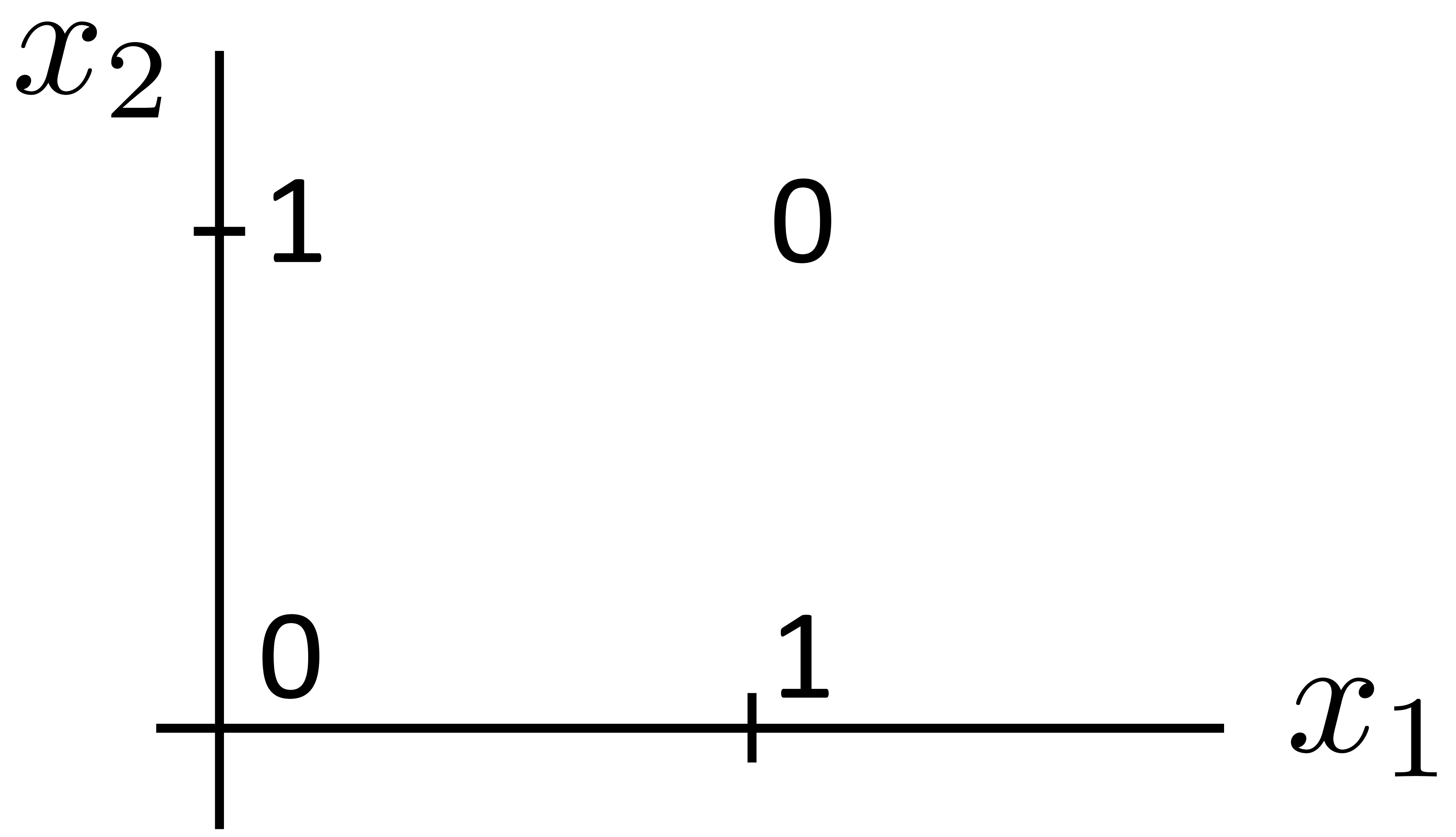
I[contains *not* & contains *good*]

Neural Networks: XOR

- Let's see how we can use neural nets to learn a simple nonlinear function
- Inputs $x_1, \; x_2$

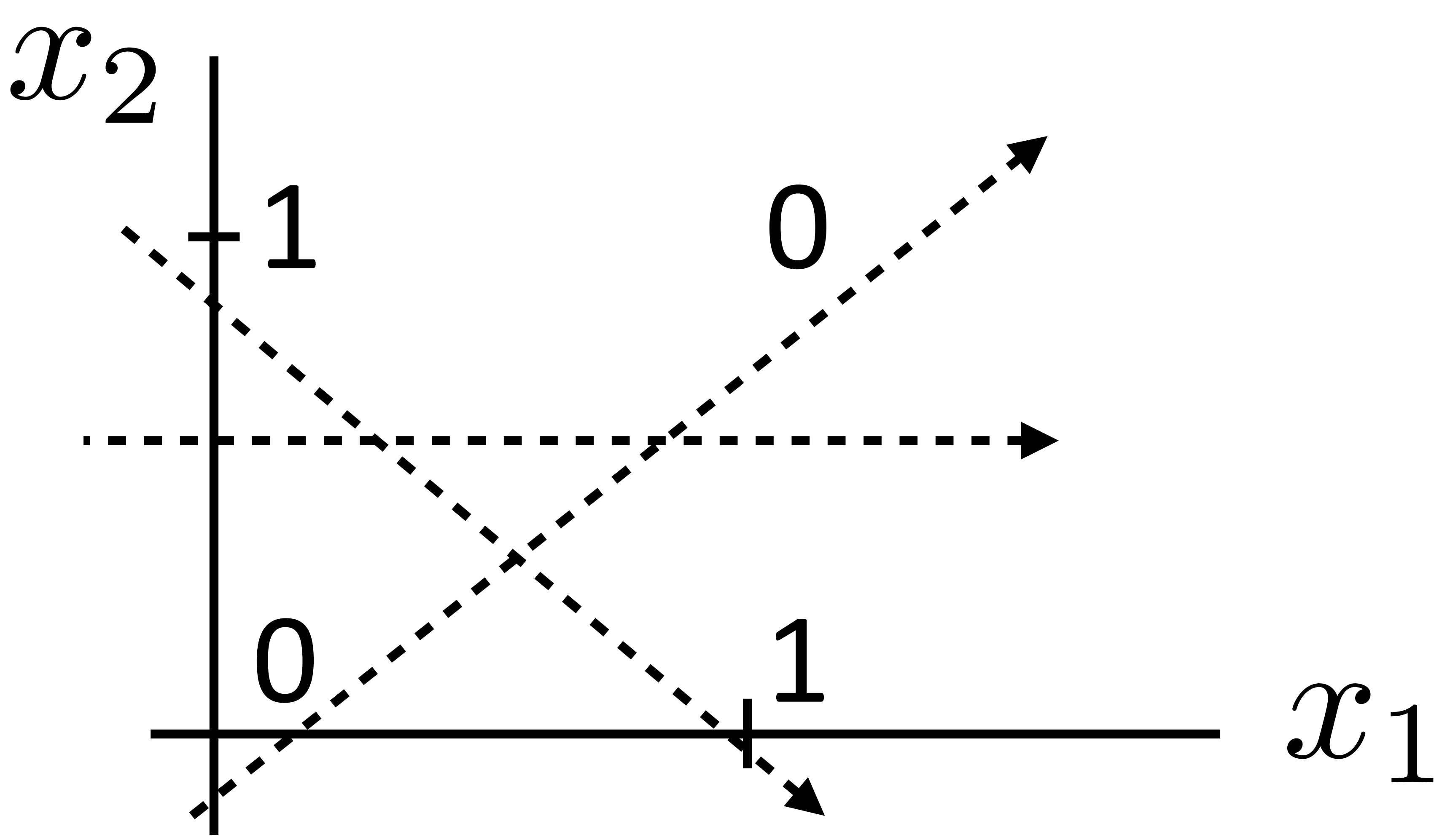
  $(\text{generally } \mathbf{x} = (x_1, \ldots, x_m))$
- Output y

  $(\text{generally } \mathbf{y} = (y_1, \ldots, y_n))$



x_1	x_2	$y = x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Neural Networks: XOR



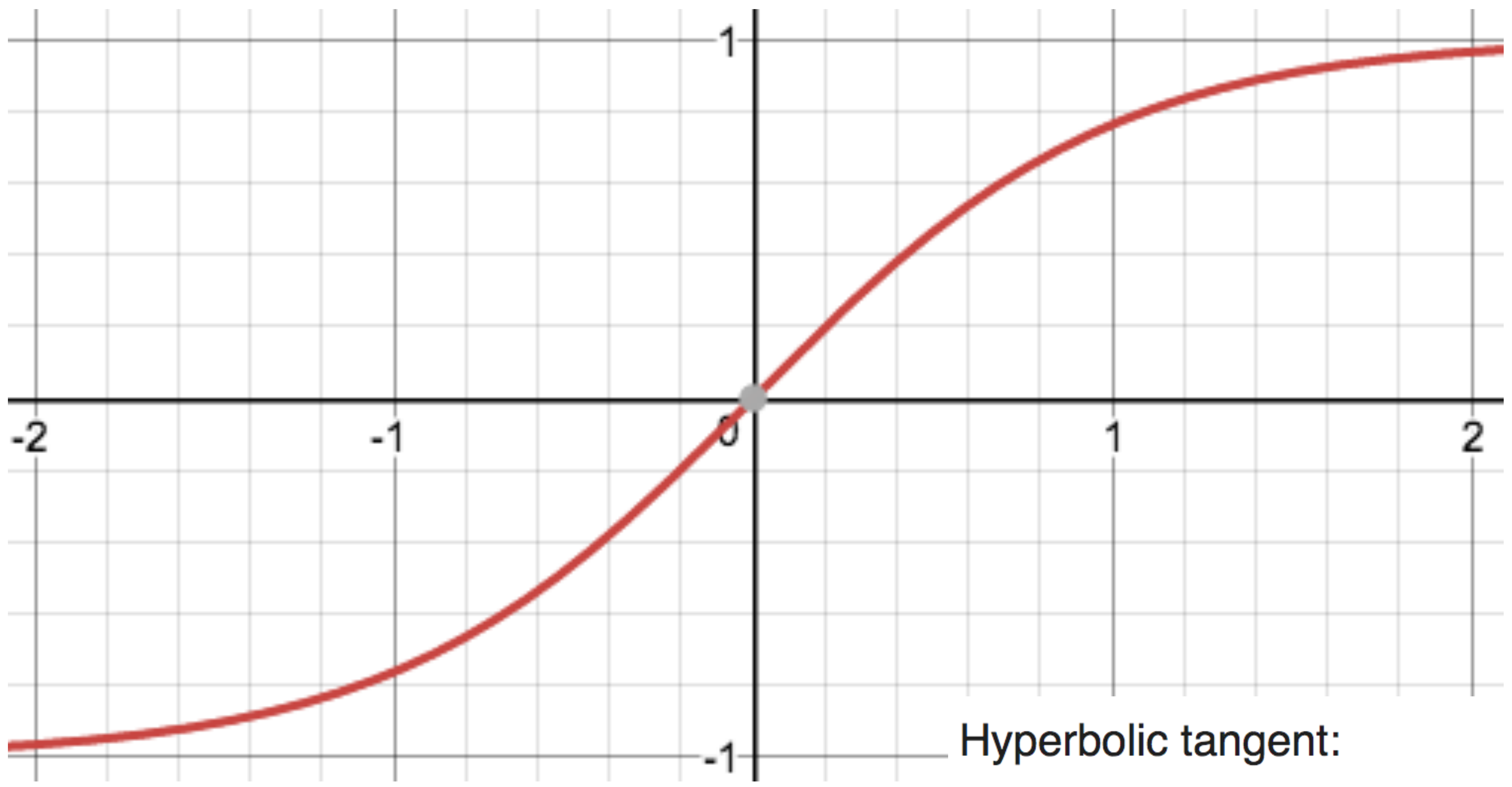
$$y = a_1 x_1 + a_2 x_2$$ ✗

$$y = a_1 x_1 + a_2 x_2 + a_3 \tanh(x_1 + x_2)$$ ✓

"or"

x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

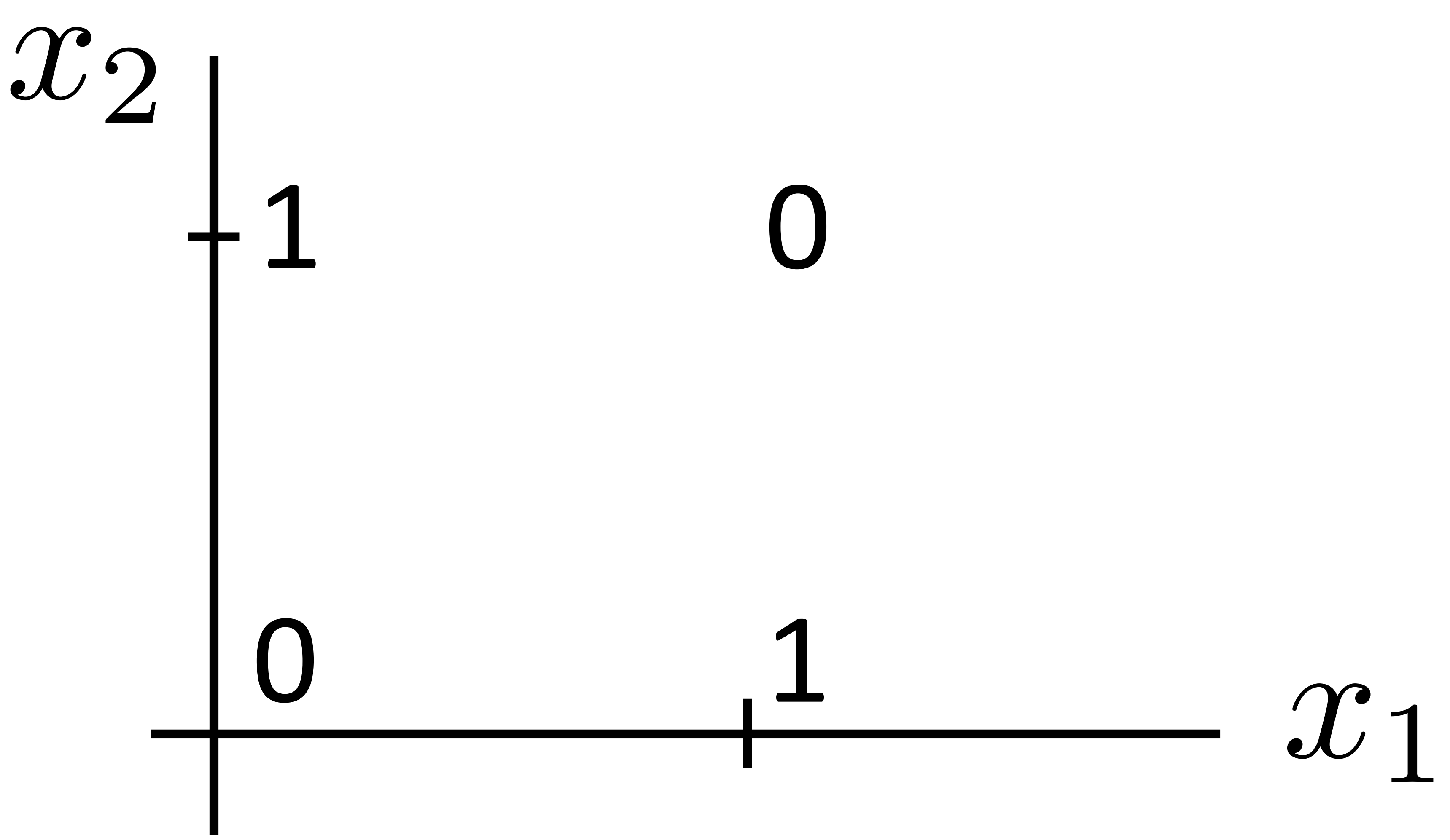
(looks like action potential in neuron)



Hyperbolic tangent:

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Neural Networks: XOR



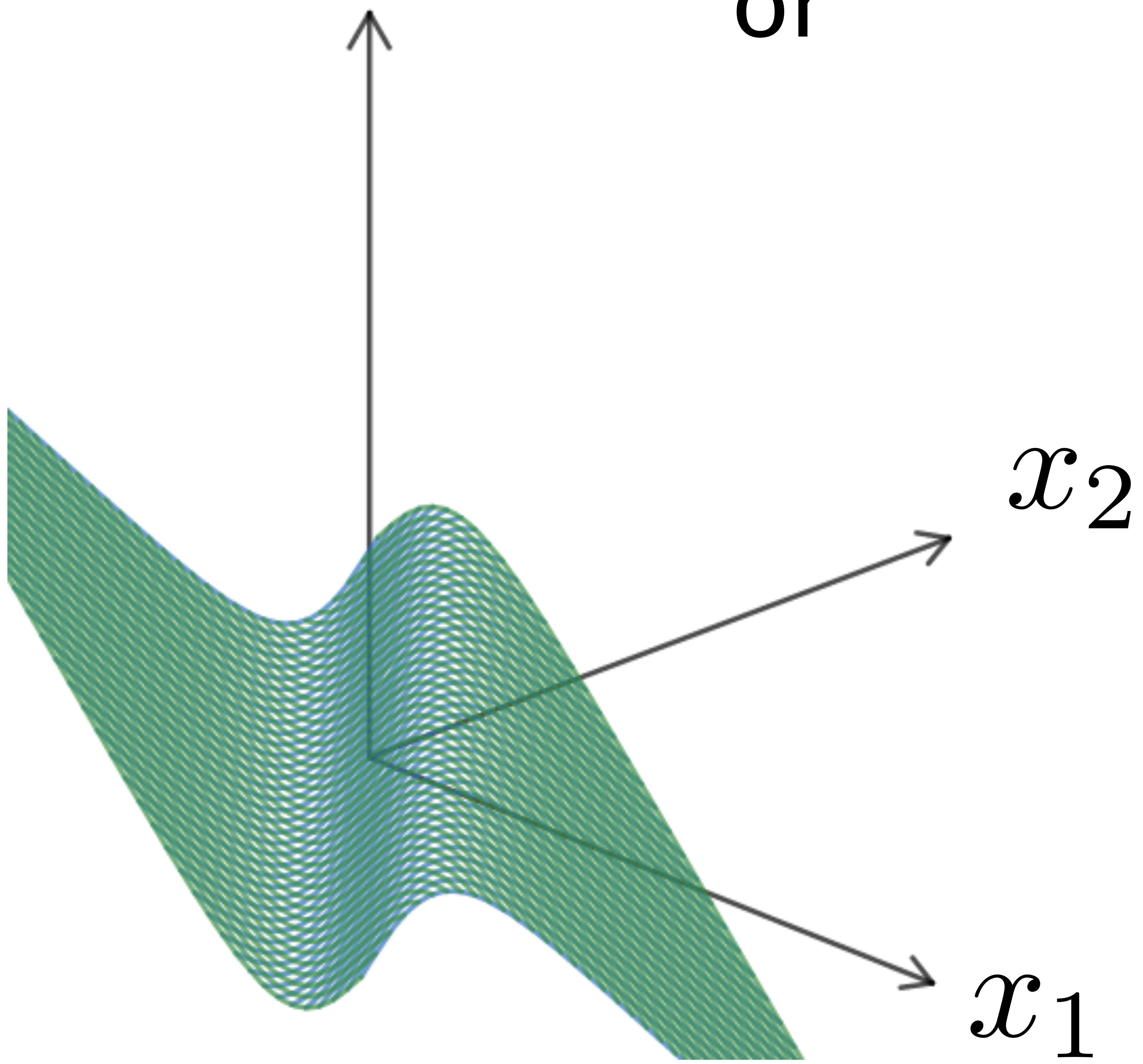
x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

$$y = a_1 x_1 + a_2 x_2$$

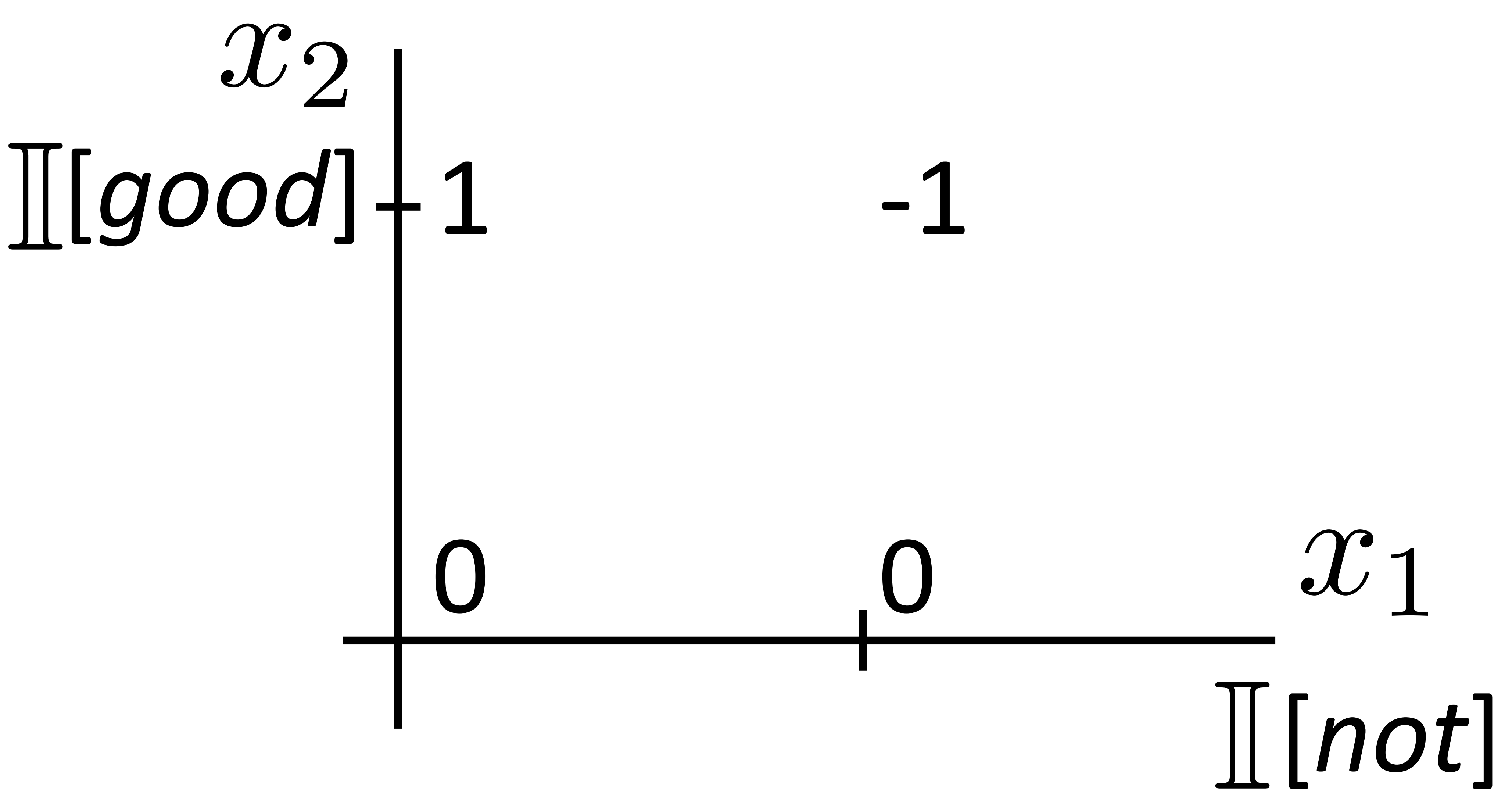
$$y = a_1 x_1 + a_2 x_2 + a_3 \tanh(x_1 + x_2)$$

$$y = -x_1 - x_2 + 2 \tanh(x_1 + x_2)$$

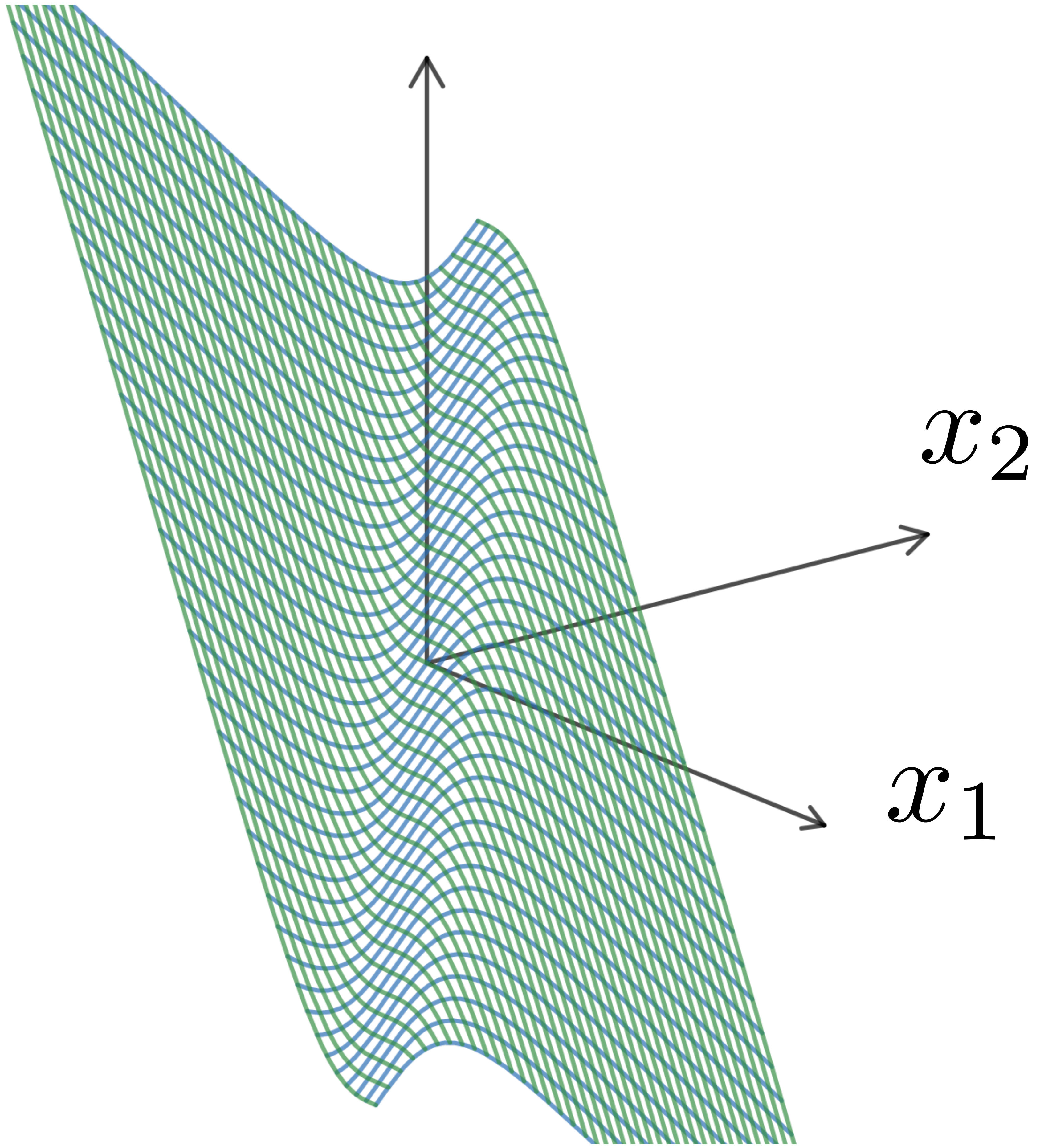
"or"



Neural Networks: XOR



$$y = -2x_1 - x_2 + 2\tanh(x_1 + x_2)$$



*the movie was **not** all that **good***

Neural Networks

Linear model: $y = \mathbf{w} \cdot \mathbf{x} + b$

$$y = g(\mathbf{w} \cdot \mathbf{x} + b)$$

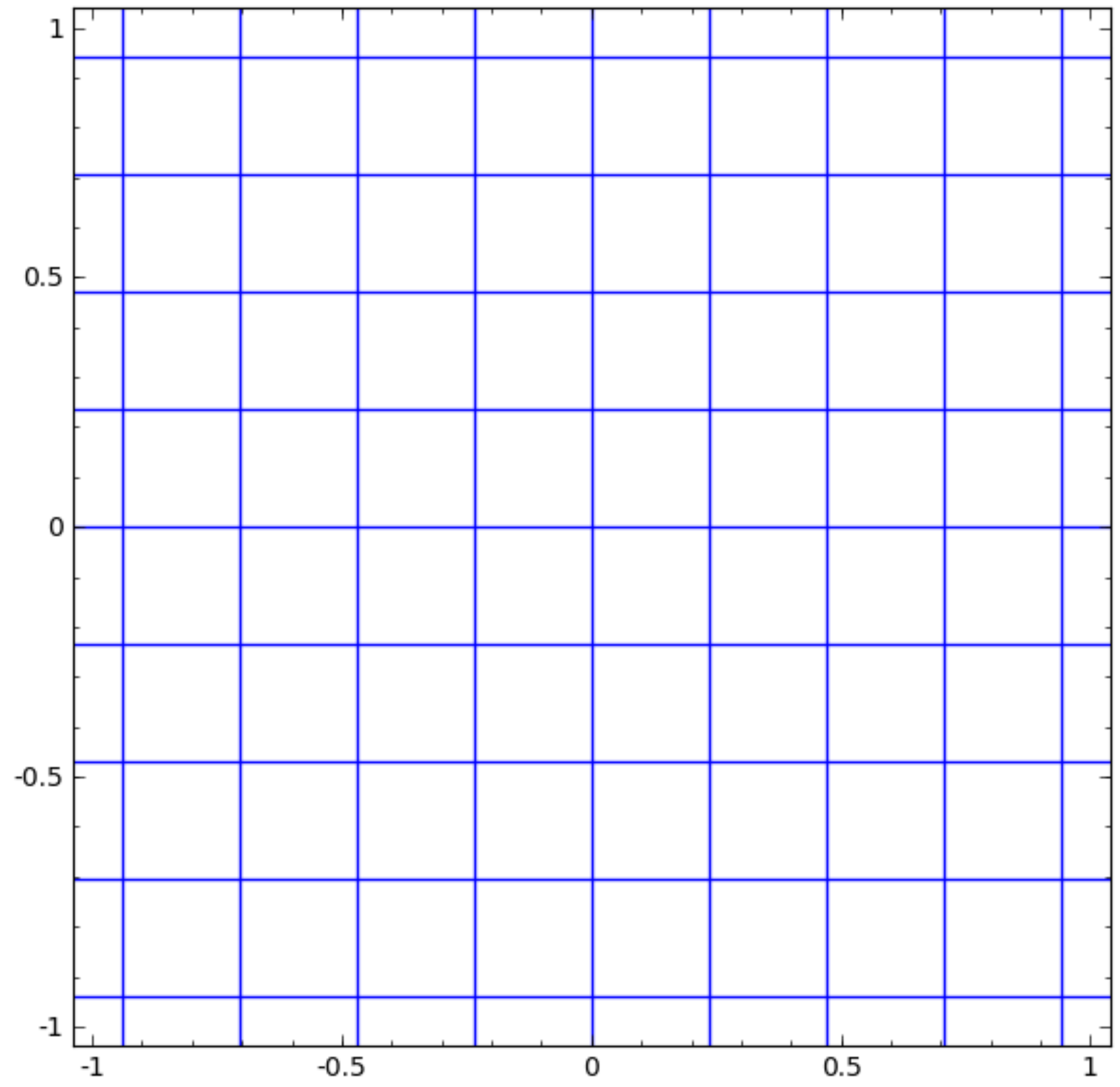
$$\mathbf{y} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Nonlinear transformation

Warp space

Shift

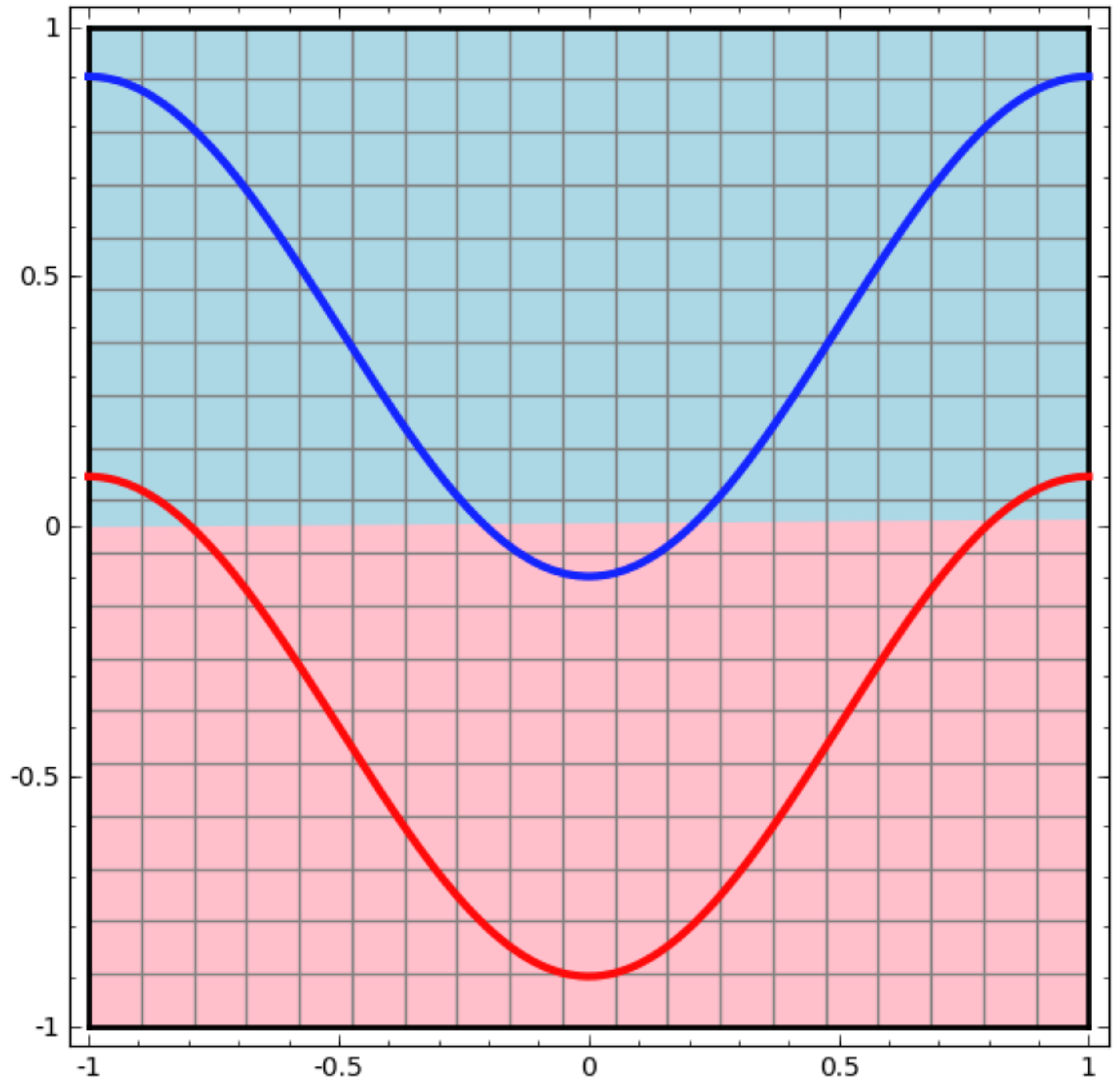
tanh



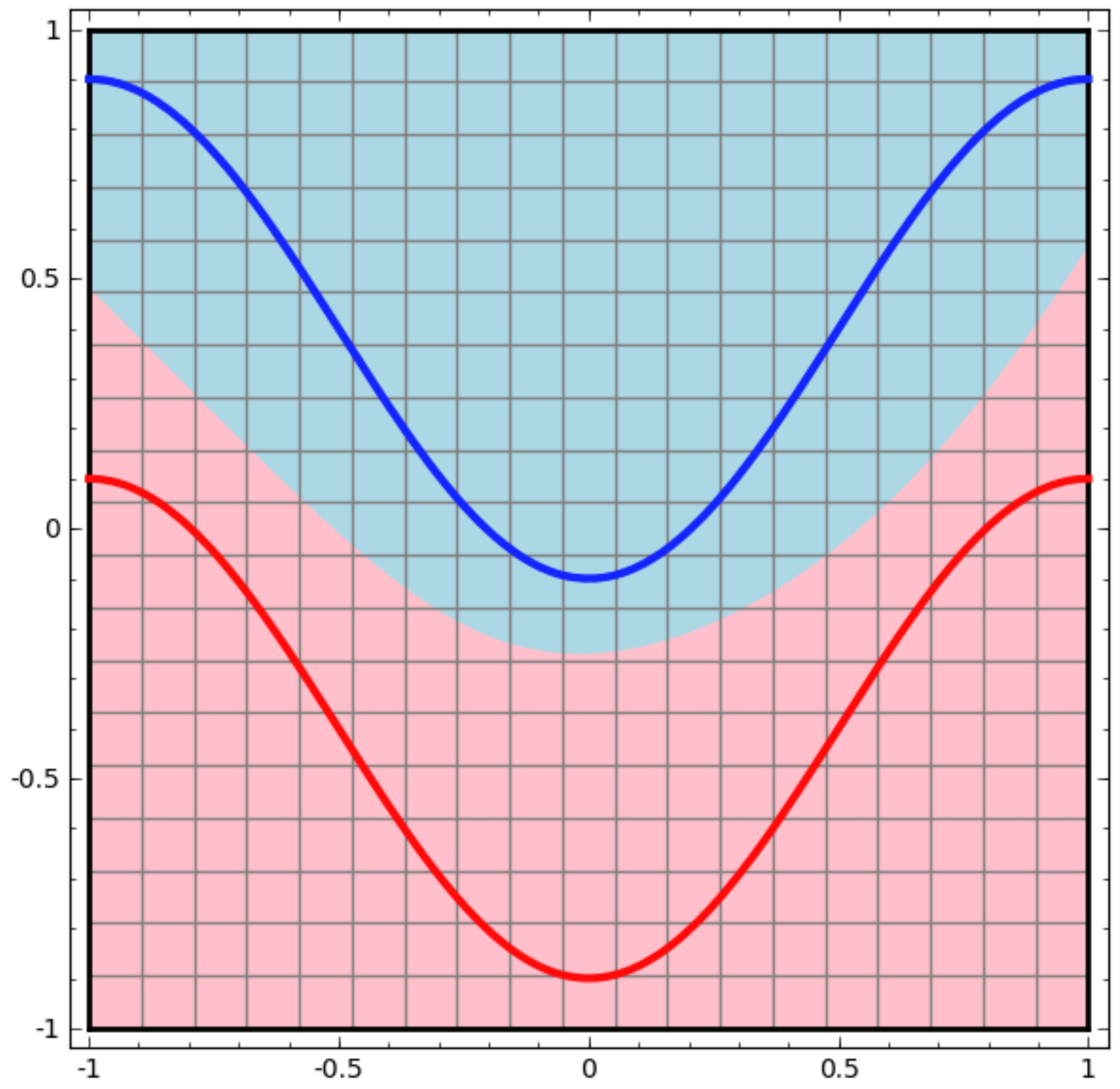
Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Neural Networks

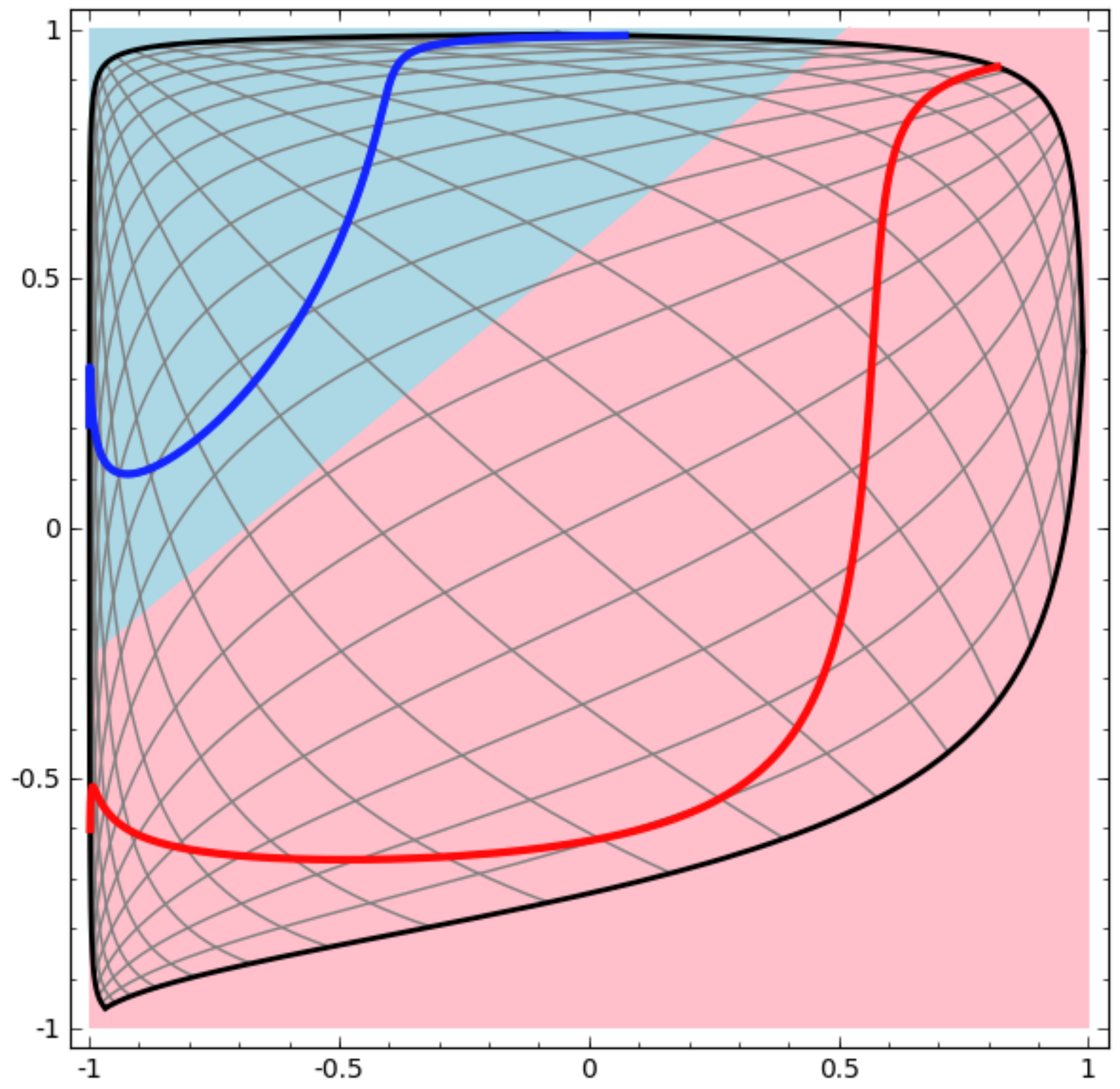
Linear classifier



Neural network

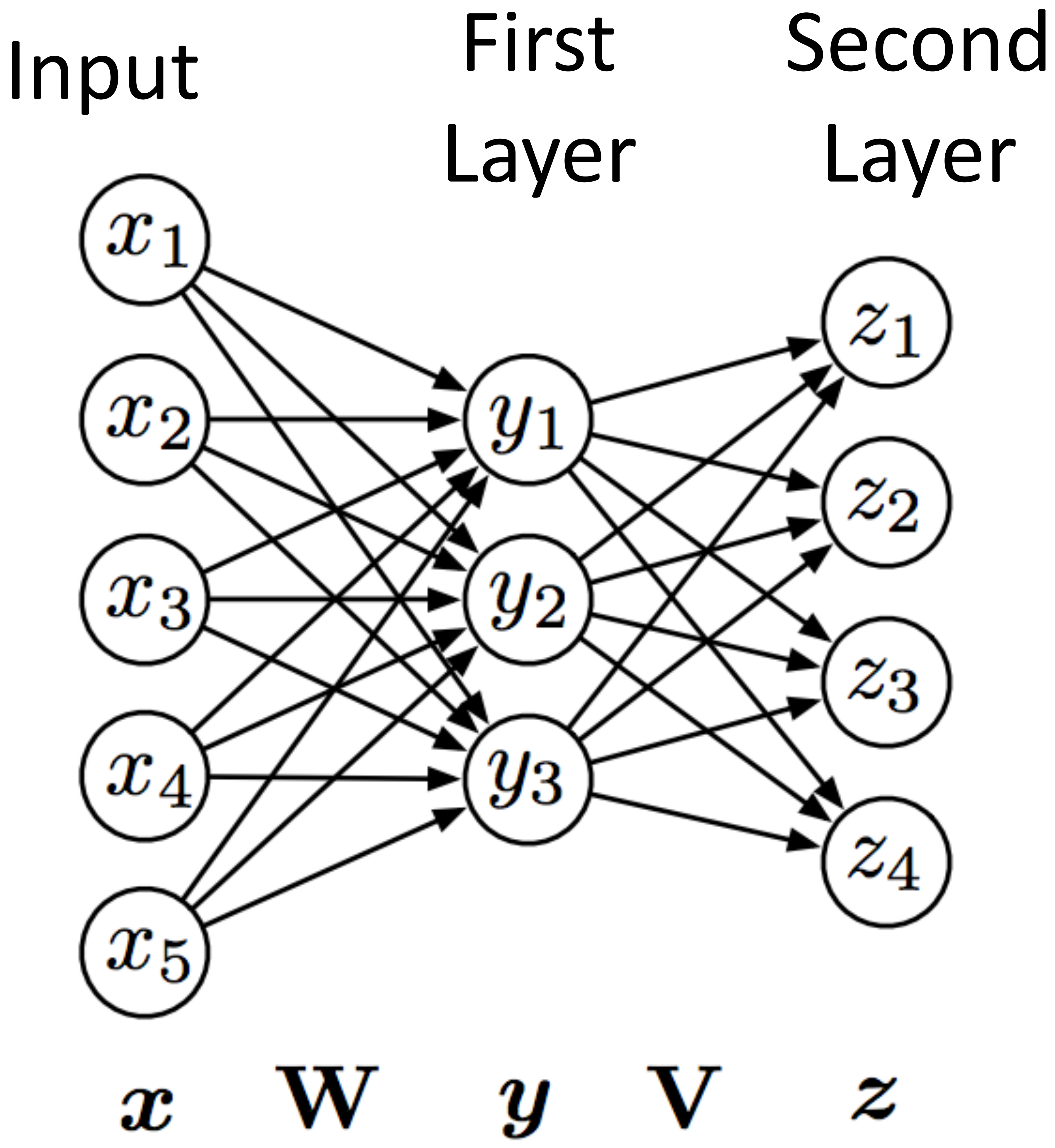


…possible because we transformed the space!



Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Deep Neural Networks



$$\boldsymbol{y} = g(\mathbf{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\mathbf{z} = g(\mathbf{V}\mathbf{y} + \mathbf{c})$$

$$\mathbf{z} = g(\mathbf{V}\underbrace{g(\mathbf{W}\mathbf{x} + \mathbf{b})}_{\text{output of first layer}} + \mathbf{c})$$

"Feedforward" computation (not recurrent)

Check: what happens if no nonlinearity?
More powerful than basic linear models?

$$\mathbf{z} = \mathbf{V}(\mathbf{W}\mathbf{x} + \mathbf{b}) + \mathbf{c}$$

Adopted from Chris Dyer

Activation Functions

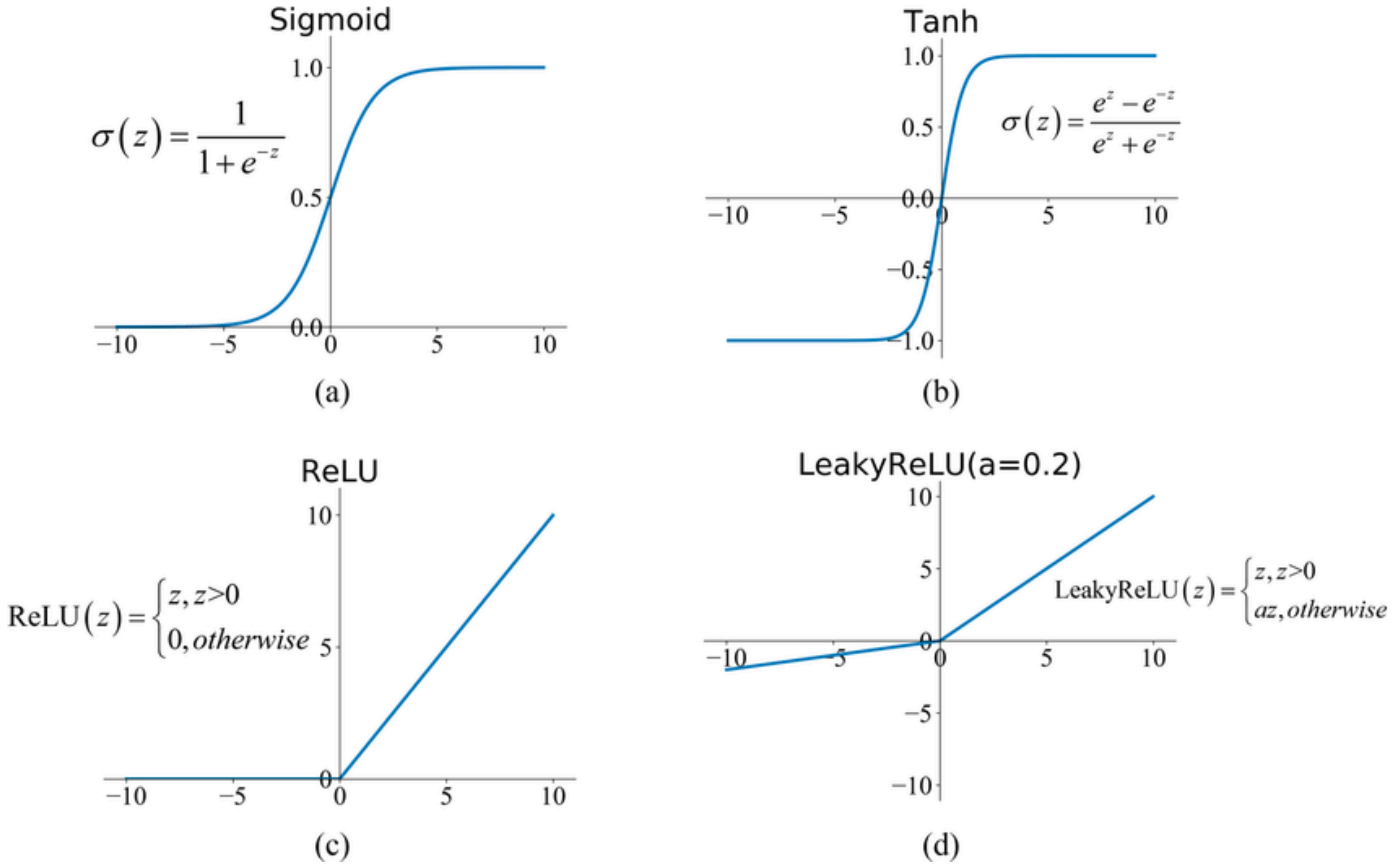
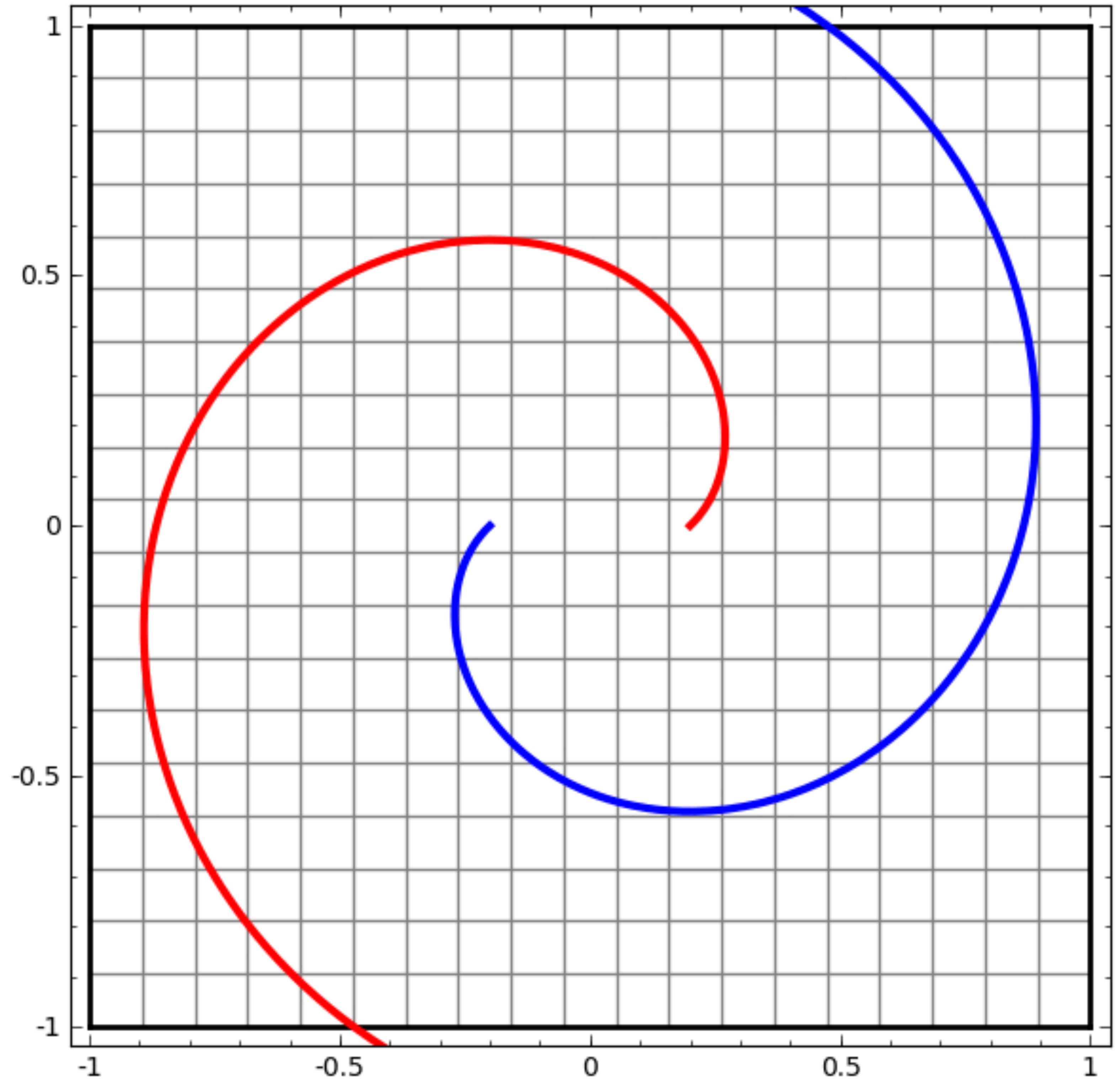


Image Credit: Junxi Feng

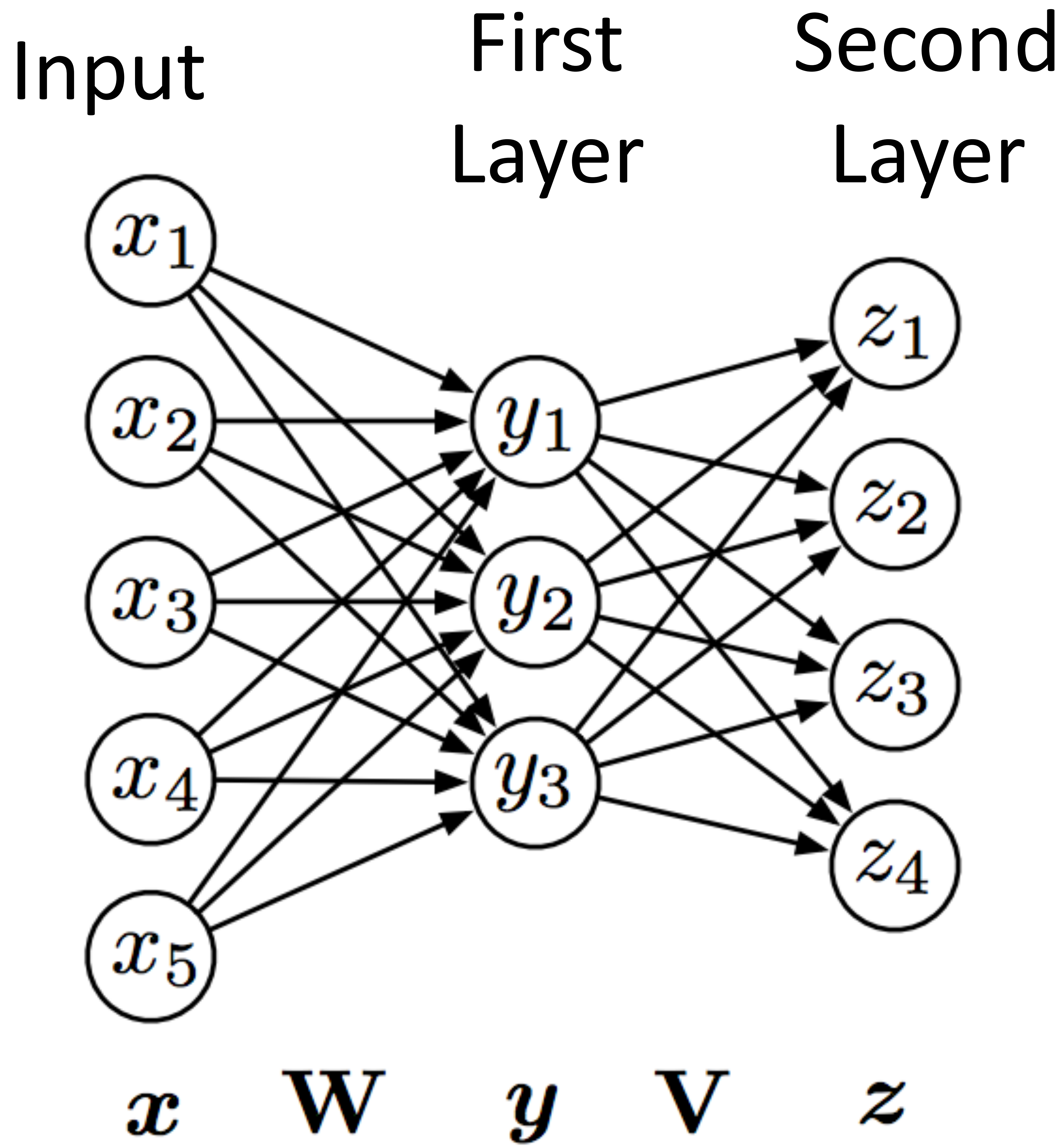
Deep Neural Networks



Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Feedforward Networks, Backpropagation

Recap: Feedforward Neural Networks



$$\boldsymbol{y} = g(\mathbf{W}\boldsymbol{x} + \boldsymbol{b})$$

$$\mathbf{z} = g(\mathbf{V}\mathbf{y} + \mathbf{c})$$

$$\mathbf{z} = g(\mathbf{V}\underbrace{g(\mathbf{W}\mathbf{x} + \mathbf{b})}_{\text{output of first layer}} + \mathbf{c})$$

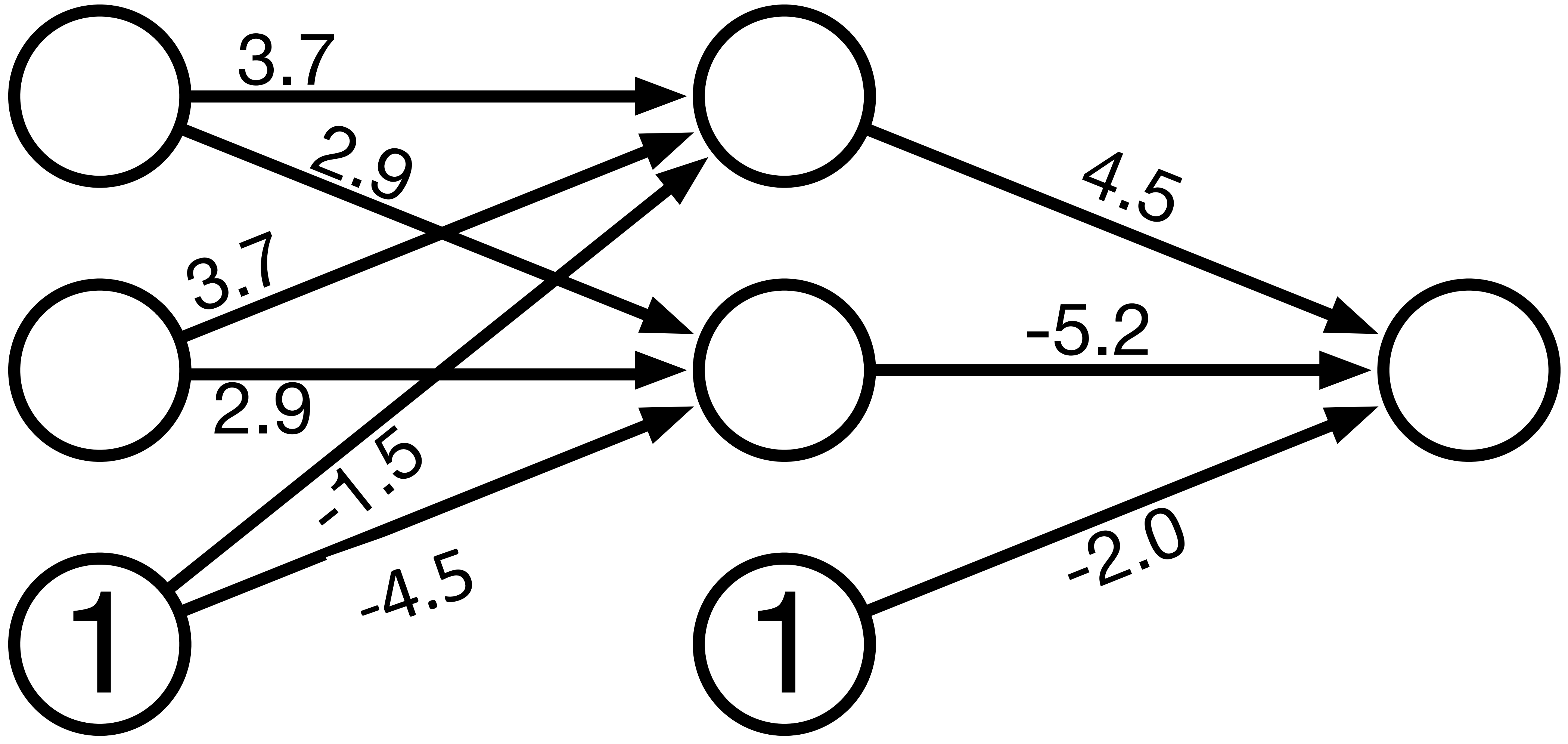
"Feedforward" computation (not recurrent)

Check: what happens if no nonlinearity?
More powerful than basic linear models?

$$\mathbf{z} = \mathbf{V}(\mathbf{W}\mathbf{x} + \mathbf{b}) + \mathbf{c}$$

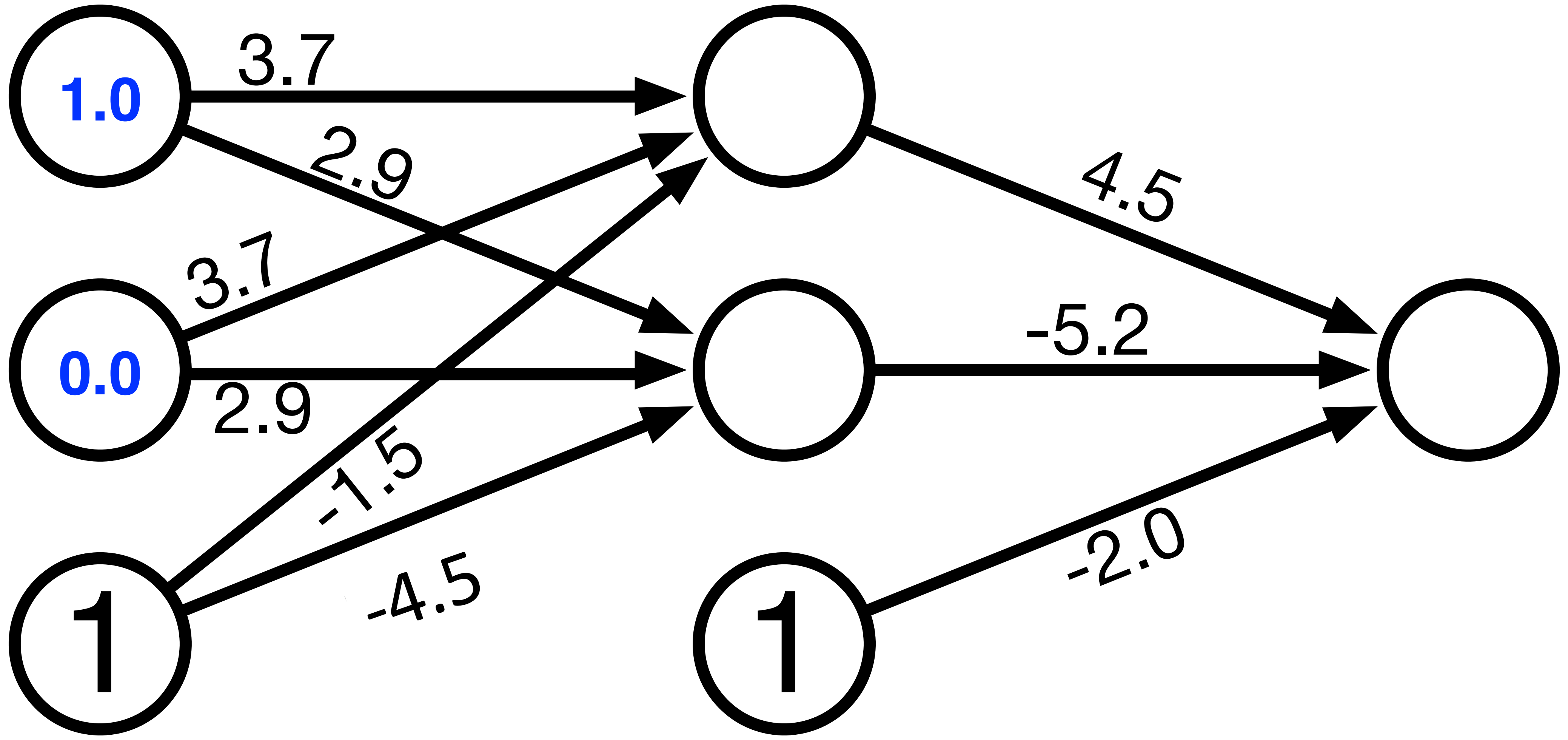
Adopted from Chris Dyer

Simple Neural Network



Slide Credit: Philipp Koehn

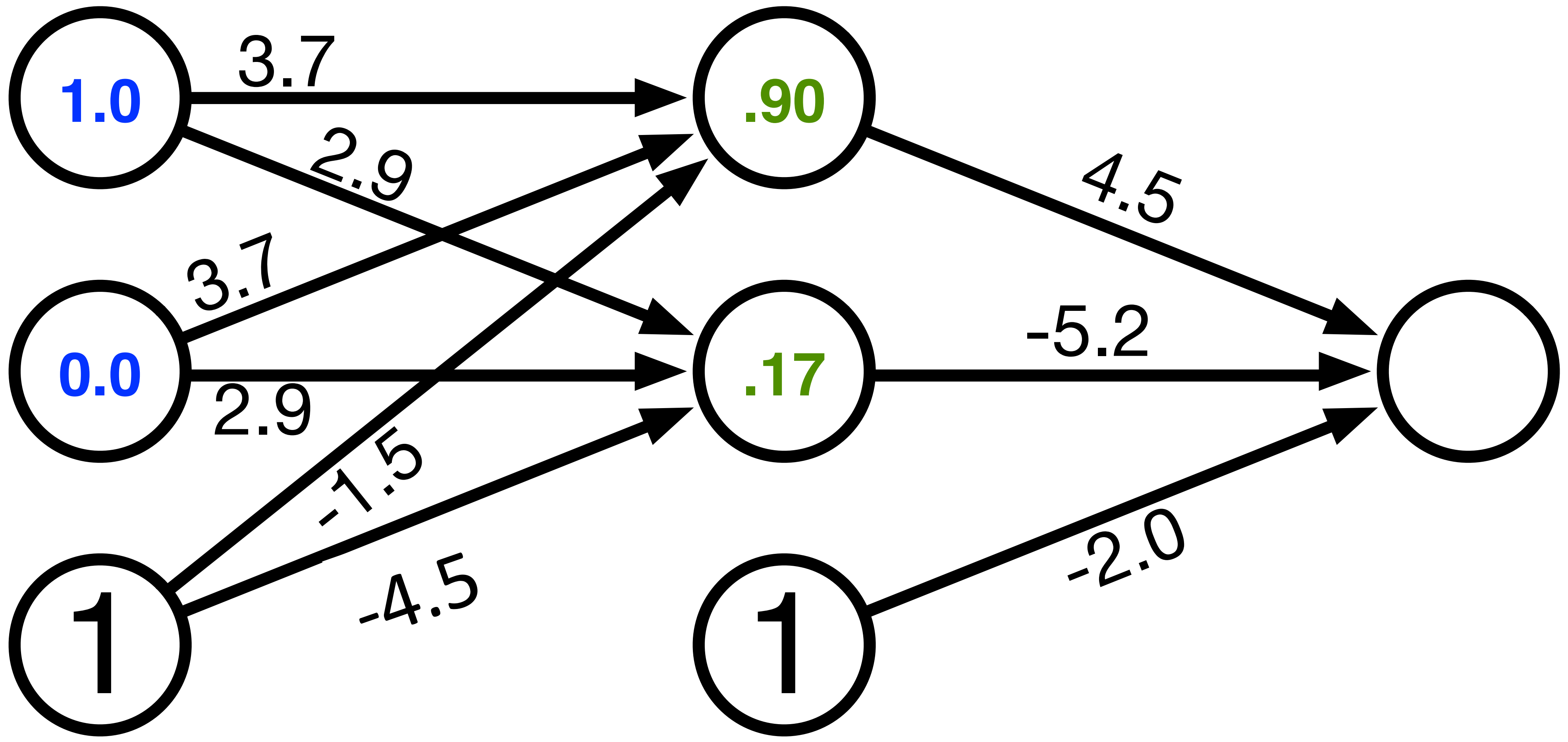
Simple Neural Network



- Try out two input values

Slide Credit: Philipp Koehn

Simple Neural Network



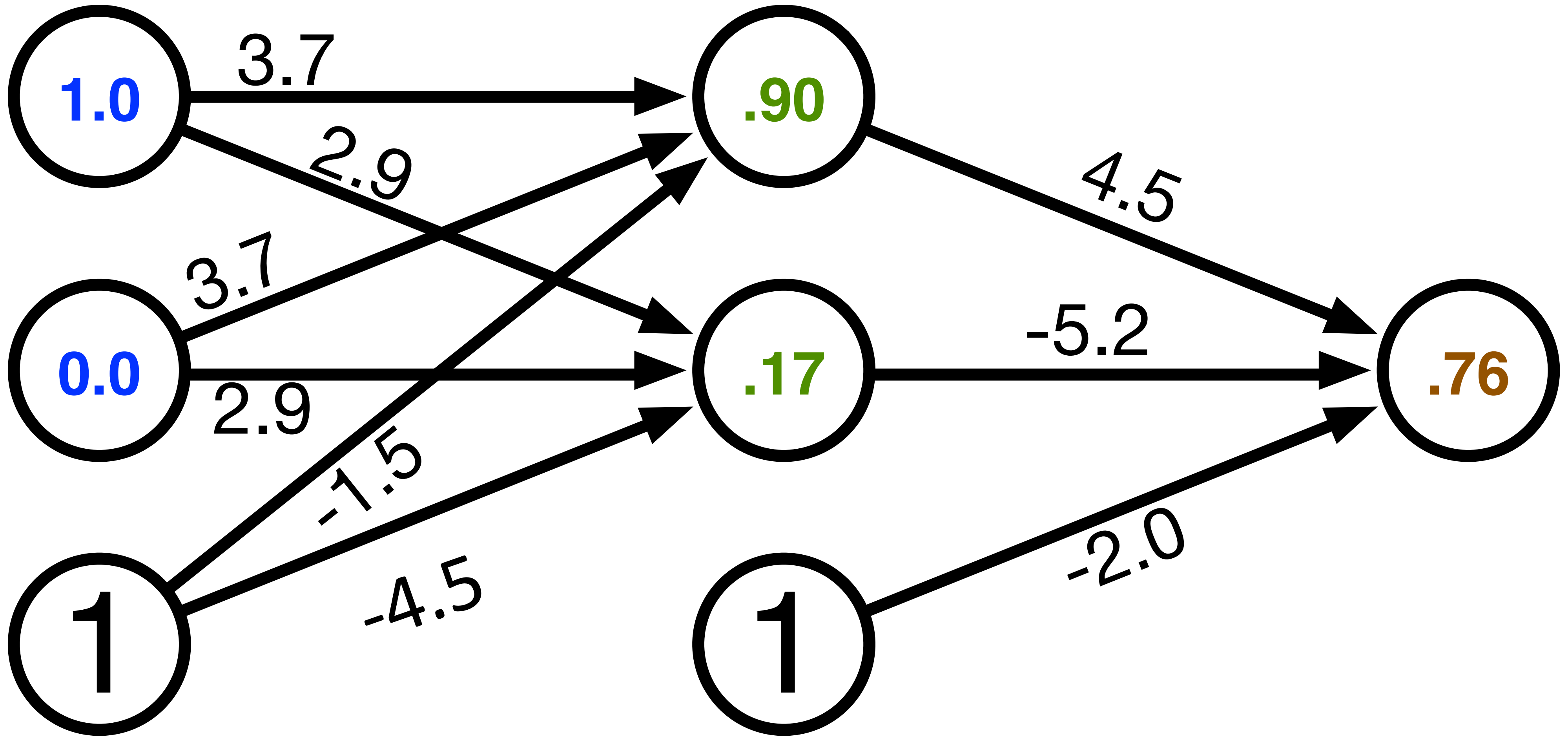
- Try out two input values
- Hidden unit computation

$$\text{sigmoid}(1.0 \times 3.7 + 0.0 \times 3.7 + 1 \times -1.5) = \text{sigmoid}(2.2) = \frac{1}{1 + e^{-2.2}} = 0.90$$

$$\text{sigmoid}(1.0 \times 2.9 + 0.0 \times 2.9 + 1 \times -4.5) = \text{sigmoid}(-1.6) = \frac{1}{1 + e^{1.6}} = 0.17$$

Slide Credit: Philipp Koehn

Simple Neural Network

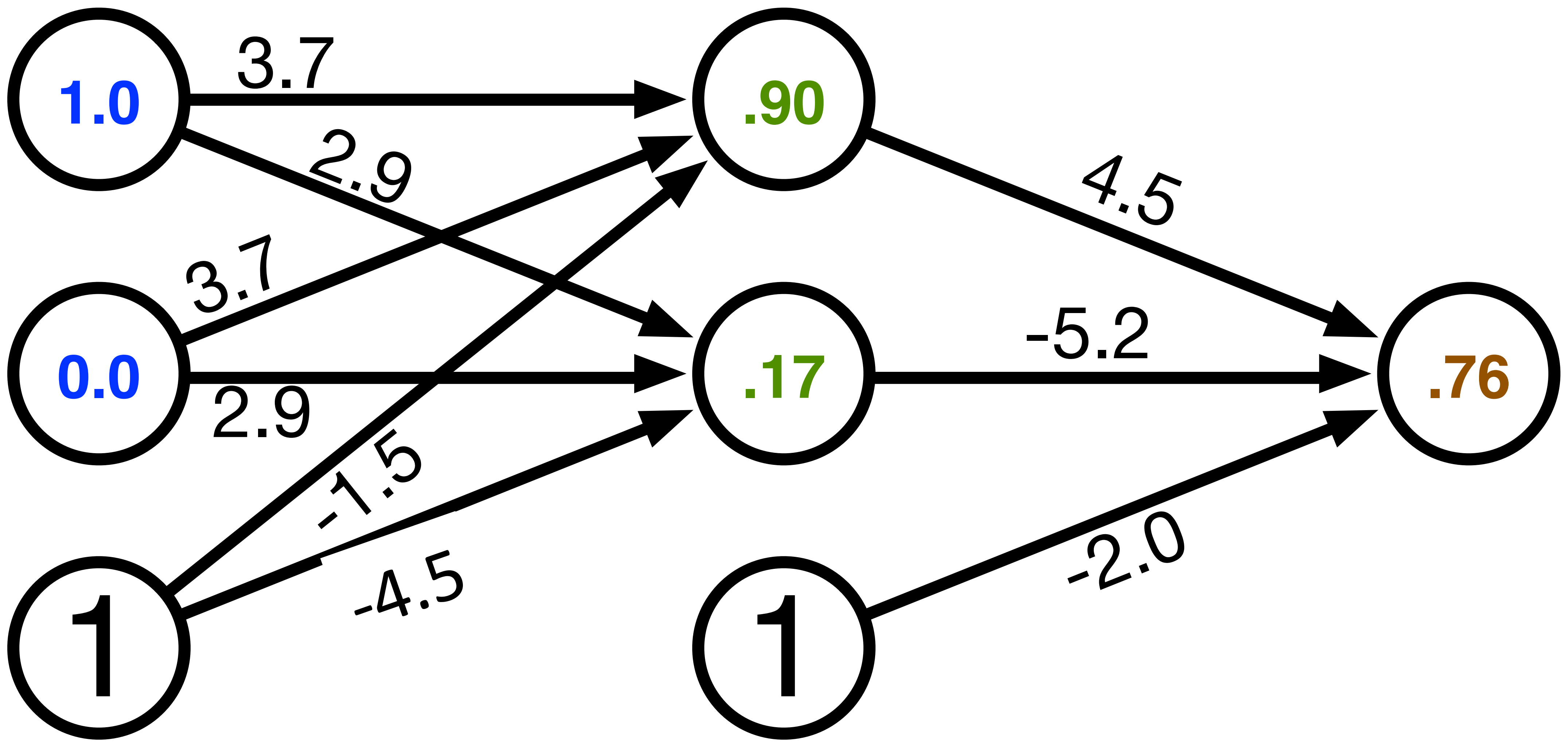


- Output unit computation

$$\text{sigmoid}(.90 \times 4.5 + .17 \times -5.2 + 1 \times -2.0) = \text{sigmoid}(1.17) = \frac{1}{1+e^{-1.17}} = 0.76$$

Slide Credit: Philipp Koehn

Error

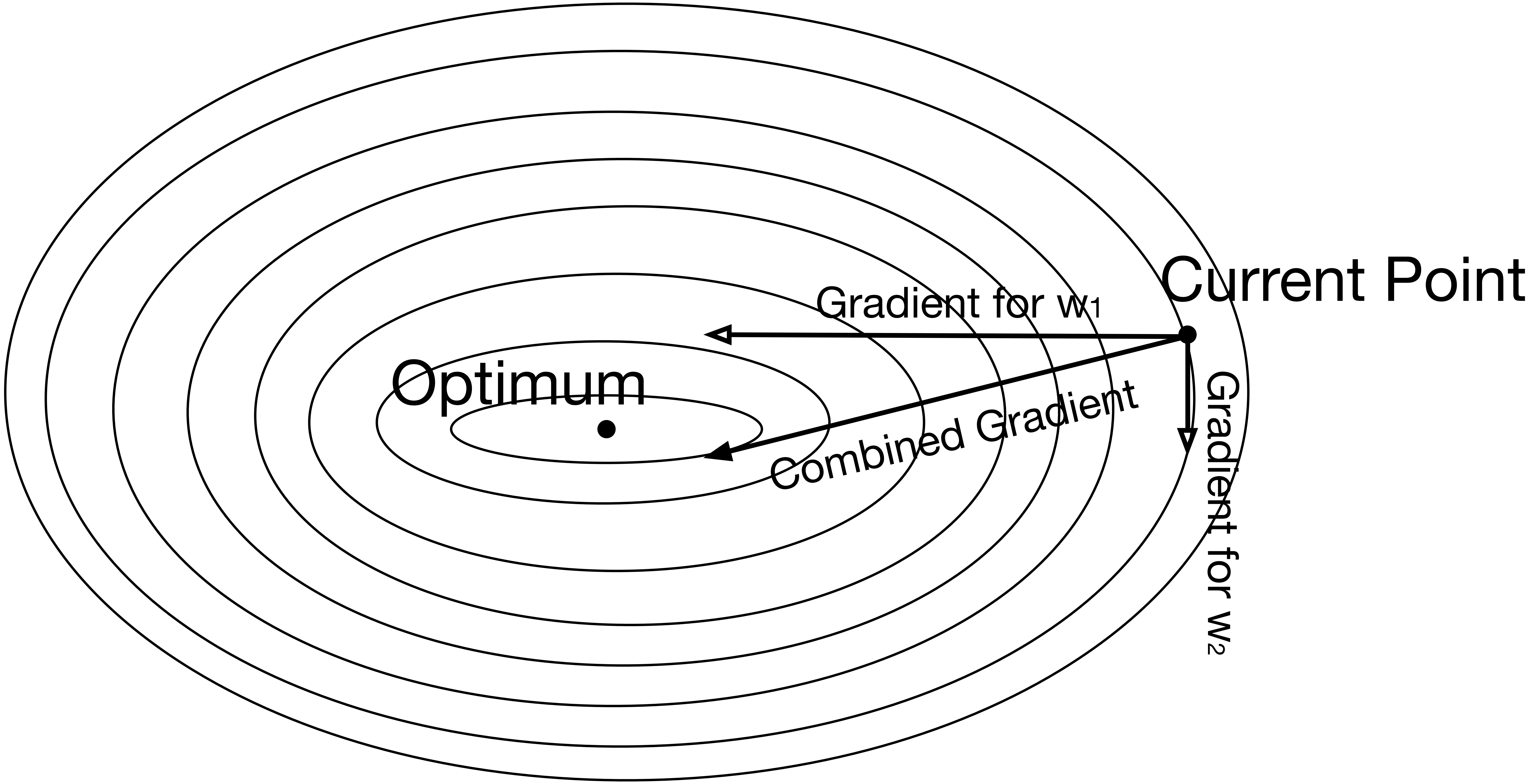


- Computed output: $y = .76$
- Correct output: $t = 1.0$

- Q: how do we adjust the weights?

Slide Credit: Philipp Koehn

Gradient Descent



Derivative of Sigmoid

- Sigmoid function:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

- Derivative:

$$\begin{aligned}
\frac{d\ \text{sigmoid}(x)}{dx} &= \frac{d}{dx}\frac{1}{1 + e^{-x}} \\
&= \frac{0 \times (1 - e^{-x}) - (-e^{-x})}{(1 + e^{-x})^2} \\
&= \frac{1}{1 + e^{-x}}\left(\frac{e^{-x}}{1 + e^{-x}}\right) \\
&= \frac{1}{1 + e^{-x}}\left(1 - \frac{1}{1 + e^{-x}}\right) \\
&= \text{sigmoid}(x)(1 - \text{sigmoid}(x))
\end{aligned}$$

Slide Credit: Philipp Koehn

Final Layer Update

- Linear combination of weights: $s = \sum_k w_k h_k$
- Activation function: $y = \text{sigmoid}(s)$
- Error (L2 norm): $E = \frac{1}{2}(t - y)^2$
- Derivative of error with regard to one weight w_k :

$$\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$$

Slide Credit: Philipp Koehn

Final Layer Update (1)

- Linear combination of weights: $s = \sum_k w_k h_k$
- Activation function: $y = \text{sigmoid}(s)$
- Error (L2 norm): $E = \frac{1}{2}(t - y)^2$
- Derivative of error with regard to one weight w_k :

$$\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$$

- Error E is defined with respect to y :

$$\frac{dE}{dy} = \frac{d}{dy}\frac{1}{2}(t - y)^2 = -(t - y)$$

Slide Credit: Philipp Koehn

Final Layer Update (2)

- Linear combination of weights: $s = \sum_k w_k h_k$
- Activation function: $y = \text{sigmoid}(s)$
- Error (L2 norm): $E = \frac{1}{2}(t - y)^2$
- Derivative of error with regard to one weight w_k :

$$\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$$

- y with respect to s is $\text{sigmoid}(s)$:

$$\frac{dy}{ds} = \frac{d\,\text{sigmoid}(s)}{ds} = \text{sigmoid}(s)(1 - \text{sigmoid}(s)) = y(1 - y)$$

Slide Credit: Philipp Koehn

Final Layer Update (3)

- Linear combination of weights: $s = \sum_k w_k h_k$
- Activation function: $y = \text{sigmoid}(s)$
- Error (L2 norm): $E = \frac{1}{2}(t - y)^2$
- Derivative of error with regard to one weight w_k :

$$\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$$

- s is weighted linear combination of hidden node values h_k :

$$\frac{ds}{dw_k} = \frac{d}{dw_k}\sum_k w_k h_k = h_k$$

Slide Credit: Philipp Koehn

Putting it All Together

- Derivative of error with regard to one weight w_k :

$$\frac{dE}{dw_k} = \frac{dE}{dy}\frac{dy}{ds}\frac{ds}{dw_k}$$

$$= -(t-y) \quad y(1-y) \quad h_k$$

error   derivative of sigmoid: y'

- Weighted adjustment will be scaled by a fixe learning rate μ :

$$\Delta w_k = \mu \; (t-y) \; y' \; h_k$$

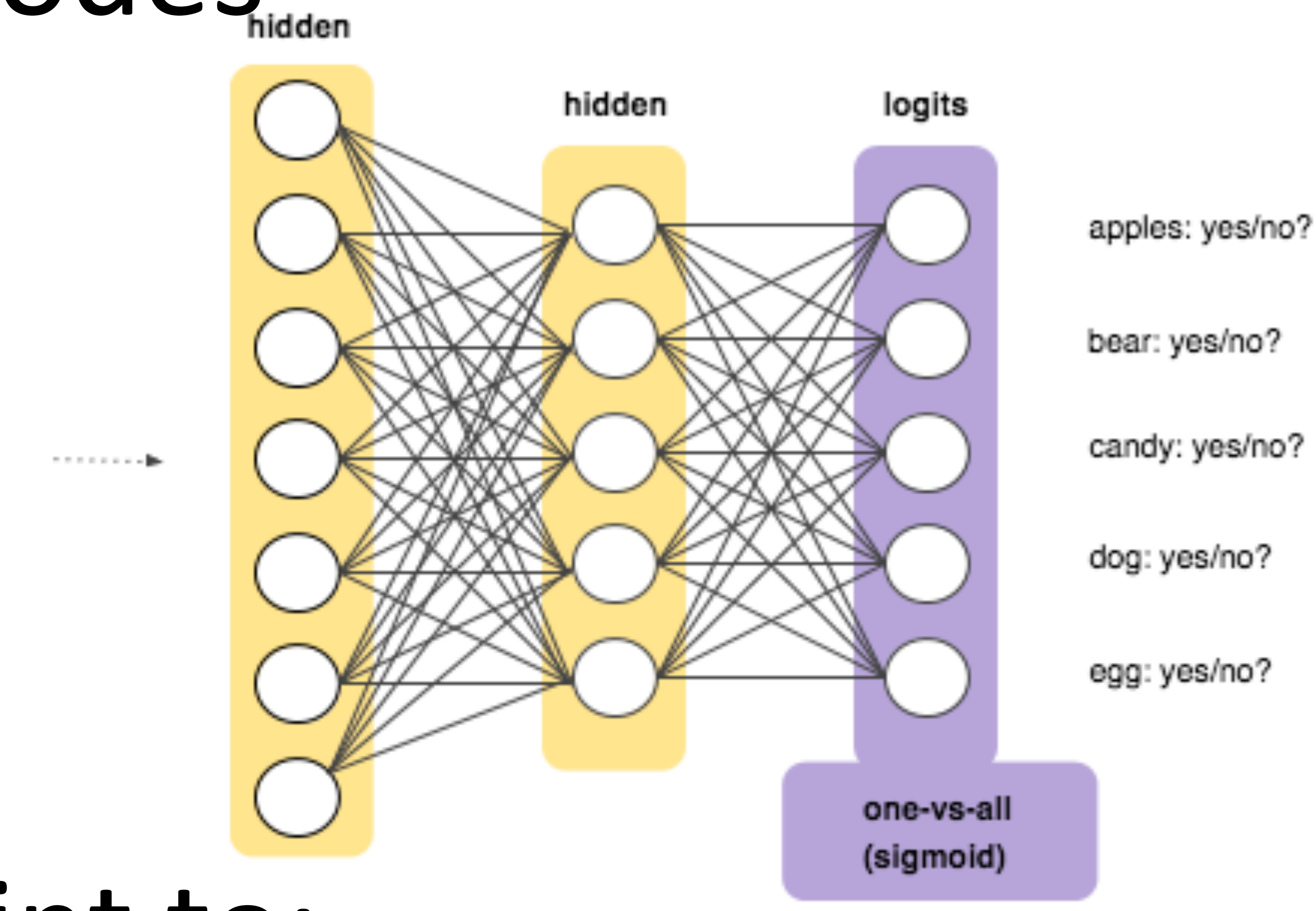
Slide Credit: Philipp Koehn

Multiple Output Nodes

- Previous slides discussed the situation with only one output node:

$$E = \frac{1}{2}(t - y)^2 \qquad \Delta w_k = \mu\ (t - y)\ y'\ h_k$$

- Sometimes, neural networks have multiple output nodes



- Error is computed over all j output nodes:

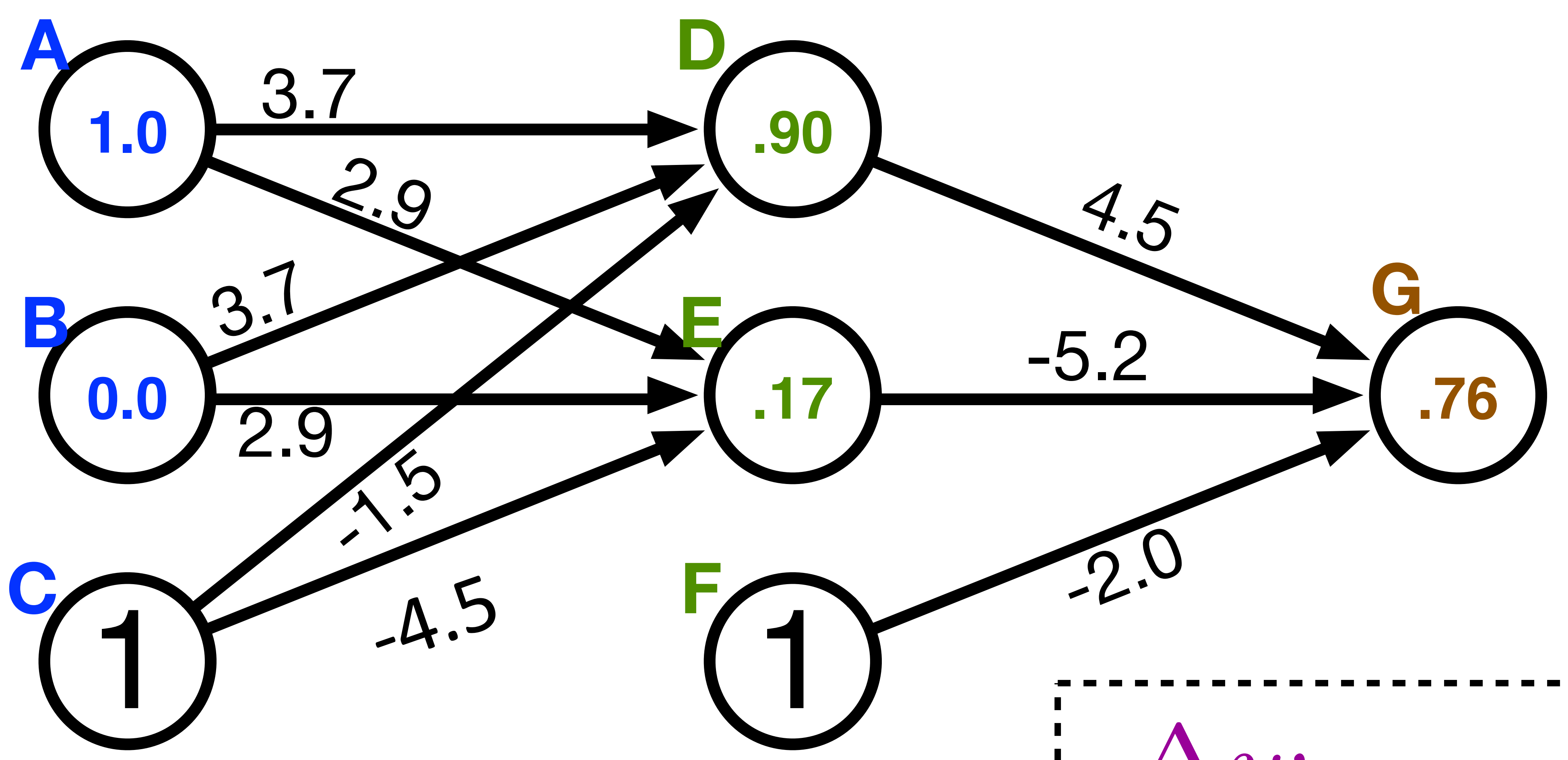
$$E = \sum_j \frac{1}{2}(t_j - y_j)^2$$

- Weights are adjusted according to the node they point to:

$$\Delta w_{j \leftarrow k} = \mu(t_j - y_j)\ y_j'\ h_k$$

Slide Credit: Philipp Koehn

Our Example



$$\Delta w_{j \leftarrow k} = \mu (t_j - y_j)\, y'_j\, h_k$$

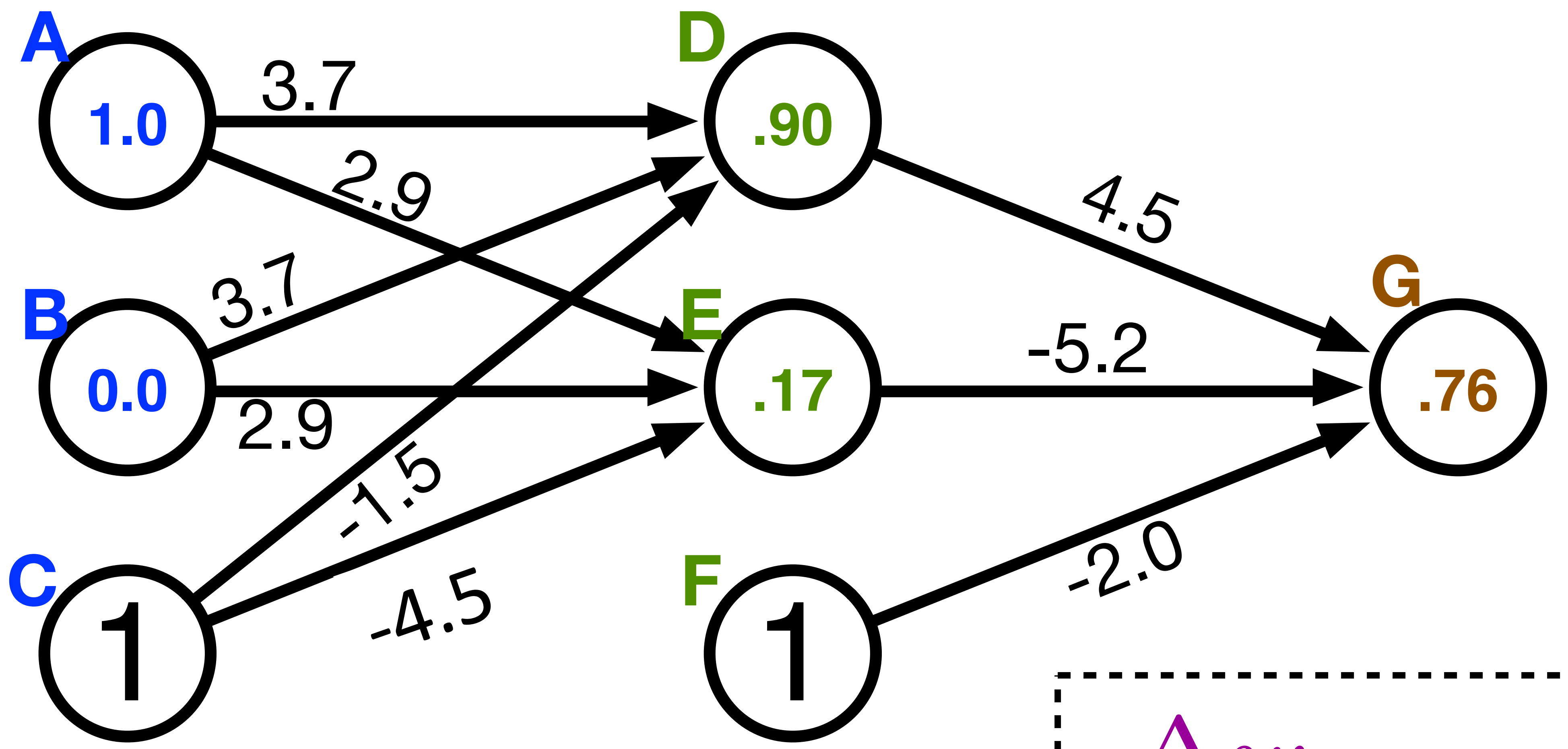
learning rate — error term — hidden node value

- Computed output: $y = .76$
- Correct output: $t = 1.0$

- Q: how do we adjust the weights?

Slide Credit: Philipp Koehn

Our Example



$$\Delta w_{j \leftarrow k} = \mu \underbrace{(t_j - y_j)\; y'_j}_{\text{error term}}\; h_k$$

learning rate

hidden node value

- Computed output: $y = .76$
- Correct output: $t = 1.0$
- Final layer weight updates (learning rate $\mu = 10$):

error term $\rightarrow$ $\delta_{\mathsf{G}} = (t - y)\, y' = (1 - .76)\; 0.181 = .0434$

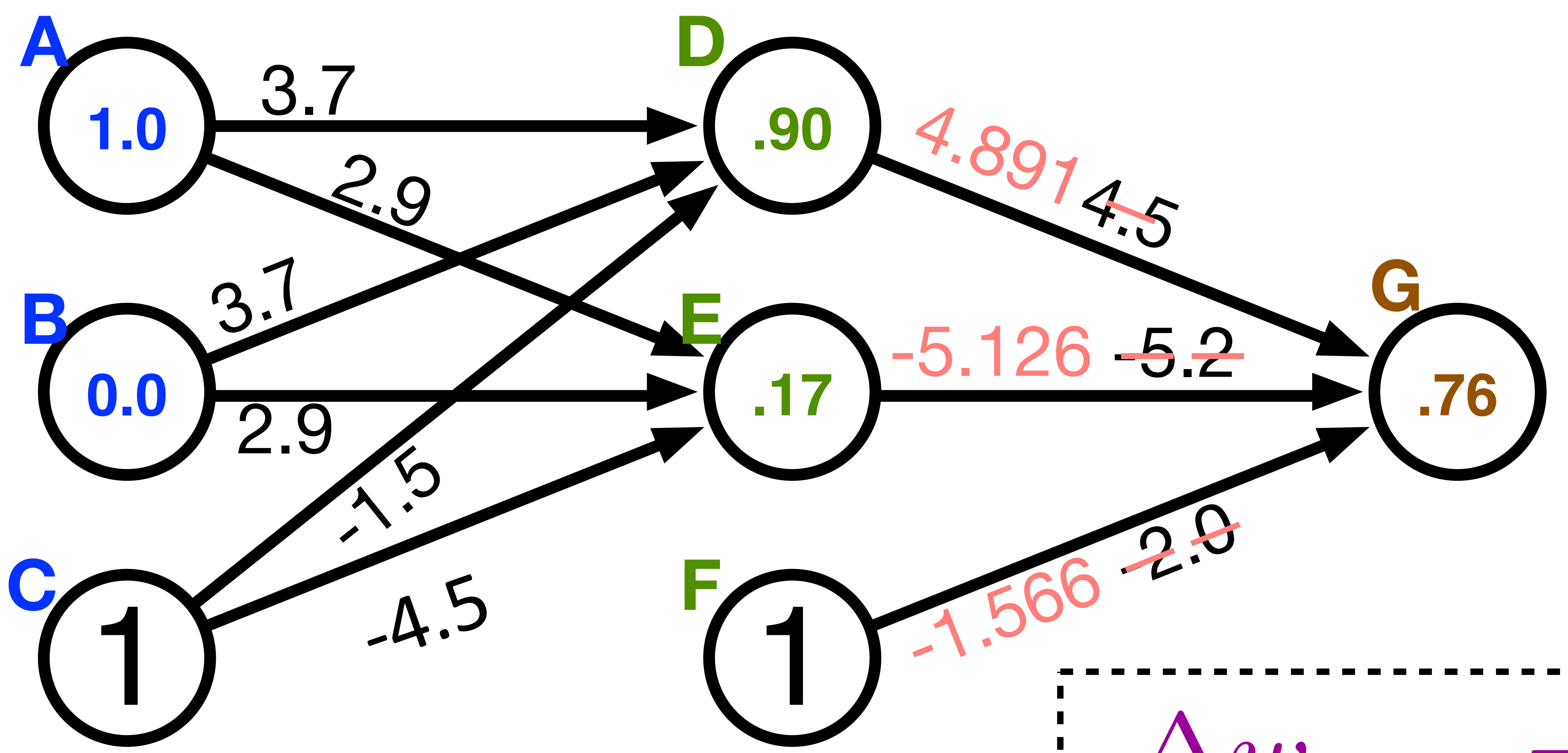
$\Delta w_{\mathsf{GD}} = \mu\; \delta_{\mathsf{G}}\; h_{\mathsf{D}} = 10 \times .0434 \times .90 = .391$

$\Delta w_{\mathsf{GE}} = \mu\; \delta_{\mathsf{G}}\; h_{\mathsf{E}} = 10 \times .0434 \times .17 = .074$

$\Delta w_{\mathsf{GF}} = \mu\; \delta_{\mathsf{G}}\; h_{\mathsf{F}} = 10 \times .0434 \times 1 = .434$

Slide Credit: Philipp Koehn

Our Example



$$\Delta w_{j \leftarrow k} = \underset{\text{learning rate}}{\mu} \underbrace{(t_j - y_j)\, y'_j}_{\text{error term}} \underset{\text{hidden node value}}{h_k}$$

- Computed output: $y = .76$
- Correct output: $t = 1.0$
- Final layer weight updates (learning rate $\mu = 10$):

error term $\rightarrow$ $\delta_{\mathsf{G}} = (t - y)\, y' = (1 - .76)\ 0.181 = .0434$

$\Delta w_{\mathsf{GD}} = \mu\ \delta_{\mathsf{G}}\ h_{\mathsf{D}} = 10 \times .0434 \times .90 = .391$

$\Delta w_{\mathsf{GE}} = \mu\ \delta_{\mathsf{G}}\ h_{\mathsf{E}} = 10 \times .0434 \times .17 = .074$

$\Delta w_{\mathsf{GF}} = \mu\ \delta_{\mathsf{G}}\ h_{\mathsf{F}} = 10 \times .0434 \times 1 = .434$

* Due to the floating-point rounding up, y' get somewhere between 1.80 and 1.824.

D: 1/(1 + e^(-2.2))= 0.90024951088
E: 1/(1 + e^(1.6))= 0.16798161486
G: sigmoid(0.90024951088 * 4.5 + 0.16798161486 * -5.2 -2.0) = sigmoid (1.17761840169) = 0.76451931587

y' = y(1-y) = 0.76451931587 * (1-0.76451931587) = 0.18002953153
y' = y(1-y) = 0.76 * (1-0.76) = 0.1824

Slide Credit: Philipp Koehn

Hidden Layer Updates

- In a hidden layer, we do not have a target output value
- But, we can compute how much each node contributed to downstream error
- Definition of error term of each node:

$$\delta_j = (t_j - y_j)\ y'_j$$

- Back-propagate the error term:

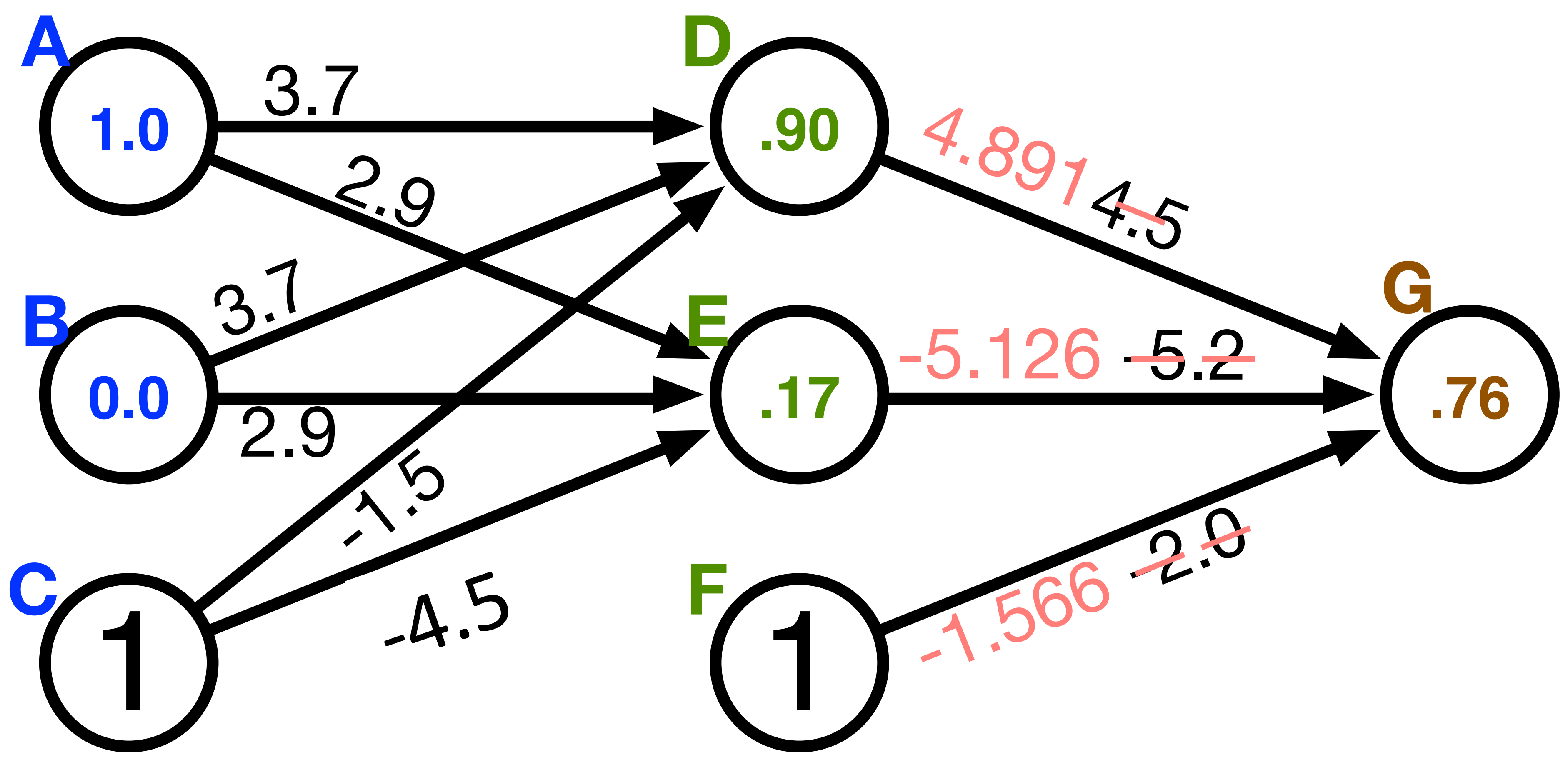
$$\delta_i = \left(\sum_j w_{j \leftarrow i} \delta_j \right) y'_i$$

- Universal update formula:

$$\Delta w_{j \leftarrow k} = \mu\ \delta_j\ h_k$$

Slide Credit: Philipp Koehn

Hidden Layer Updates



- Hidden node **D**:

$\delta_{\text{D}} = \left(\sum_j w_{j\leftarrow i}\delta_j\right) y'_{\text{D}} = w_{\text{GD}}\ \delta_{\text{G}}\ y'_{\text{D}} = 4.5 \times .0434 \times .0898 = .0175$
$\Delta w_{\text{DA}} = \mu\ \delta_{\text{D}}\ h_{\text{A}} = 10 \times .0175 \times 1.0 = .175$
$\Delta w_{\text{DB}} = \mu\ \delta_{\text{D}}\ h_{\text{B}} = 10 \times .0175 \times 0.0 = 0$
$\Delta w_{\text{DC}} = \mu\ \delta_{\text{D}}\ h_{\text{C}} = 10 \times .0175 \times 1 = .175$

- Hidden node **E**:

$\delta_{\text{E}} = \left(\sum_j w_{j\leftarrow i}\delta_j\right) y'_{\text{E}} = w_{\text{GE}}\ \delta_{\text{G}}\ y'_{\text{E}} = -5.2 \times .0434 \times 0.2055 = -.0464$
$\Delta w_{\text{EA}} = \mu\ \delta_{\text{E}}\ h_{\text{A}} = 10 \times -.0464 \times 1.0 = -.464$
etc.

Slide Credit: Philipp Koehn

Feedforward Networks, Backpropagation (more formally)

Logistic Regression with NNs

$$P(y|\mathbf{x}) = \frac{\exp(w^\top f(\mathbf{x}, y))}{\sum_{y'} \exp(w^\top f(\mathbf{x}, y'))}$$

- Single scalar probability

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}\left([w^\top f(\mathbf{x}, y)]_{y \in \mathcal{Y}}\right)$$

- Compute scores for all possible labels at once (returns vector)

$$\text{softmax}(p)_i = \frac{\exp(p_i)}{\sum_{i'} \exp(p_{i'})}$$

- softmax: exps and normalizes a given vector

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wf(\mathbf{x}))$$

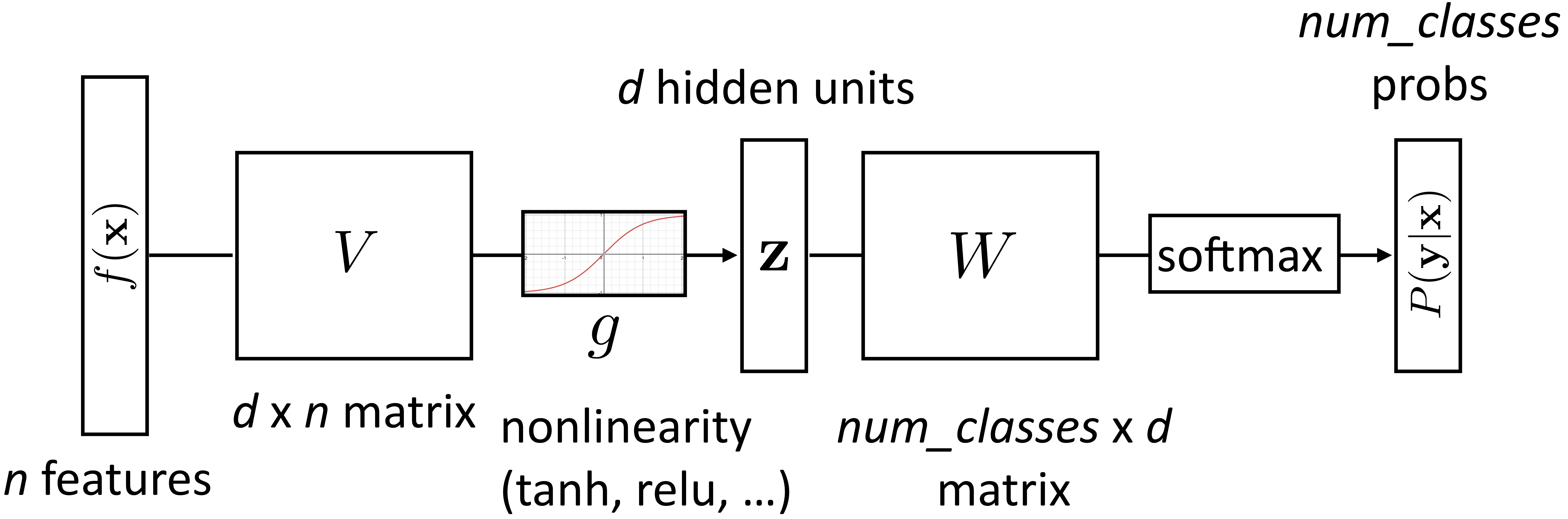
- Weight vector per class;
  W is [num classes x num feats]

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$

- Now one hidden layer

Neural Networks for Classification

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



We can think of a neural network classifier with one hidden layer as building a vector z which is a hidden layer representation (i.e. latent features) of the input, and then running standard logistic regression on the features that the network develops in z.

Training Neural Networks

$$P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(W\mathbf{z}) \qquad \mathbf{z} = g(Vf(\mathbf{x}))$$

- Maximize log likelihood of training data

$$\mathcal{L}(\mathbf{x}, i^*) = \log P(y = i^*|\mathbf{x}) = \log\left(\mathrm{softmax}(W\mathbf{z}) \cdot e_{i^*}\right)$$

- *i**: index of the gold label

- e_i: 1 in the *i*th row, zero elsewhere. Dot by this = select *i*th index

$$\mathcal{L}(\mathbf{x}, i^*) = W\mathbf{z} \cdot e_{i^*} - \log \sum_j \exp(W\mathbf{z}) \cdot e_j$$

Computing Gradients

$$\mathcal{L}(\mathbf{x}, i^*) = W\mathbf{z} \cdot e_{i^*} - \log \sum_j \exp(W\mathbf{z}) \cdot e_j$$

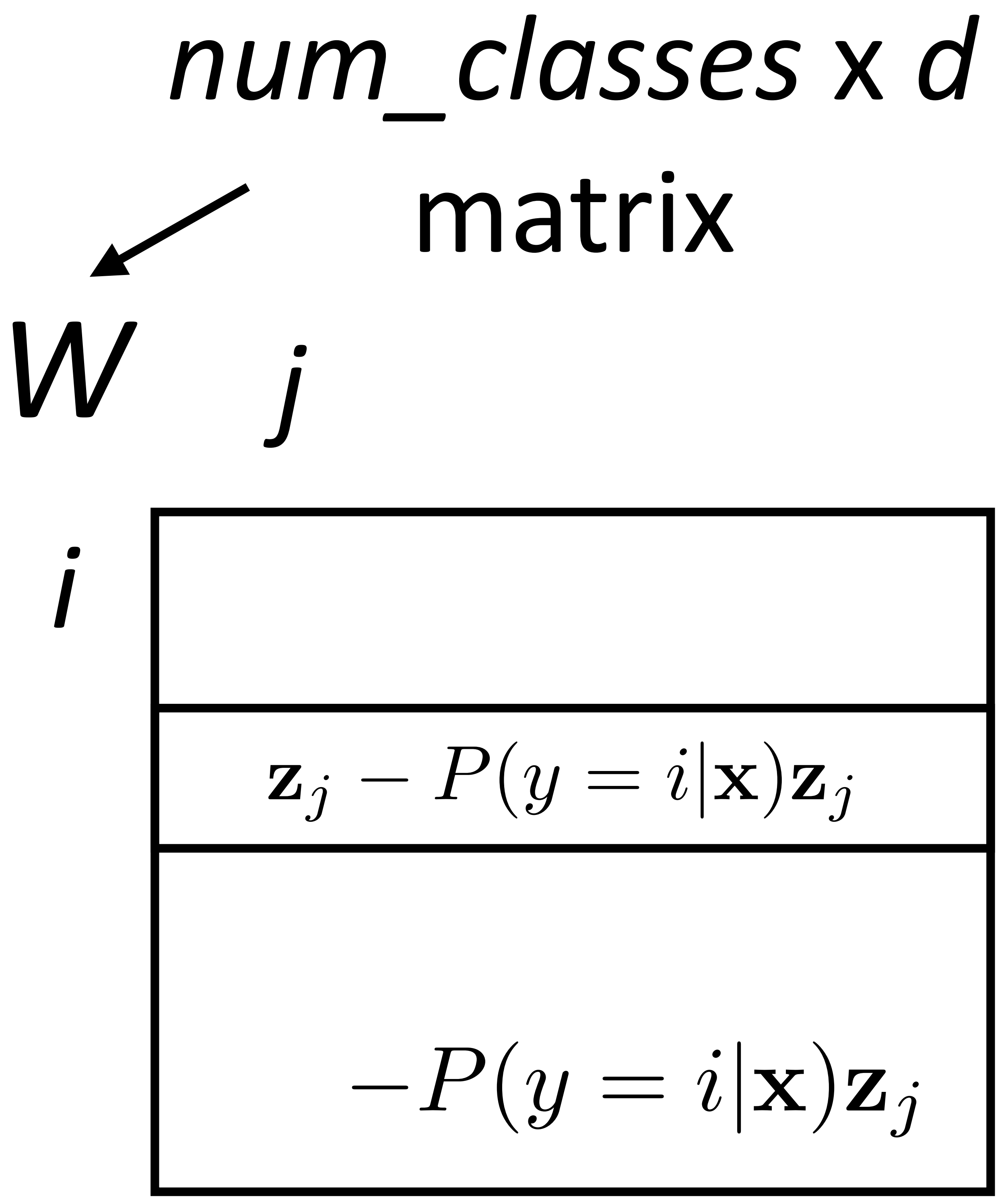
- Gradient with respect to *W*

index of output space $\mathcal{Y}$

$$\frac{\partial}{\partial W_{ij}} \mathcal{L}(\mathbf{x}, i^*) = \begin{cases} \mathbf{z}_j - P(y = i|\mathbf{x})\mathbf{z}_j & \text{if } i = i^* \\ -P(y = i|\mathbf{x})\mathbf{z}_j & \text{otherwise} \end{cases}$$

index of gold label

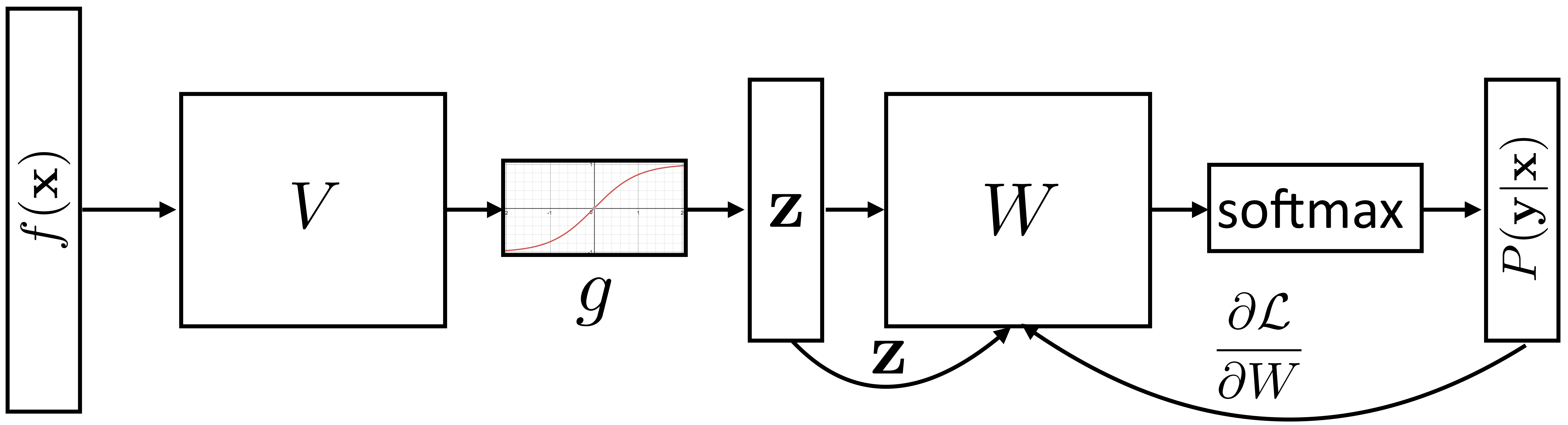
index of vector **z**



- Looks like logistic regression with ***z*** as the features!

Neural Networks for Classification

$$P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(Wg(Vf(\mathbf{x})))$$



$$\boxed{\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{W}} = \mathbf{z}(e_{i^*} - P(\mathbf{y}|\mathbf{x})) = \mathbf{z} \cdot err(\mathrm{root})}$$

Computing Gradients: Backpropagation

$$\mathcal{L}(\mathbf{x}, i^*) = W\mathbf{z} \cdot e_{i^*} - \log \sum_j \exp(W\mathbf{z}) \cdot e_j$$

$$\mathbf{z} = g(Vf(\mathbf{x}))$$

Activations at hidden layer

- Gradient with respect to *V*: apply the chain rule

$$\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial V_{ij}} = \boxed{\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}}} \frac{\partial \mathbf{z}}{\partial V_{ij}}$$

[some math…]

$$err(\text{root}) = e_{i^*} - P(\mathbf{y}|\mathbf{x})$$

dim = num_classes

$$\boxed{\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} = err(\mathbf{z}) = W^\top err(\text{root})}$$

dim = d

Computing Gradients: Backpropagation

$$\mathcal{L}(\mathbf{x}, i^*) = W\mathbf{z} \cdot e_{i^*} - \log \sum_{j=1}^{m} \exp(W\mathbf{z} \cdot e_j)$$

$\mathbf{z} = g(Vf(\mathbf{x}))$

Activations at hidden layer

- Gradient with respect to *V*: apply the chain rule

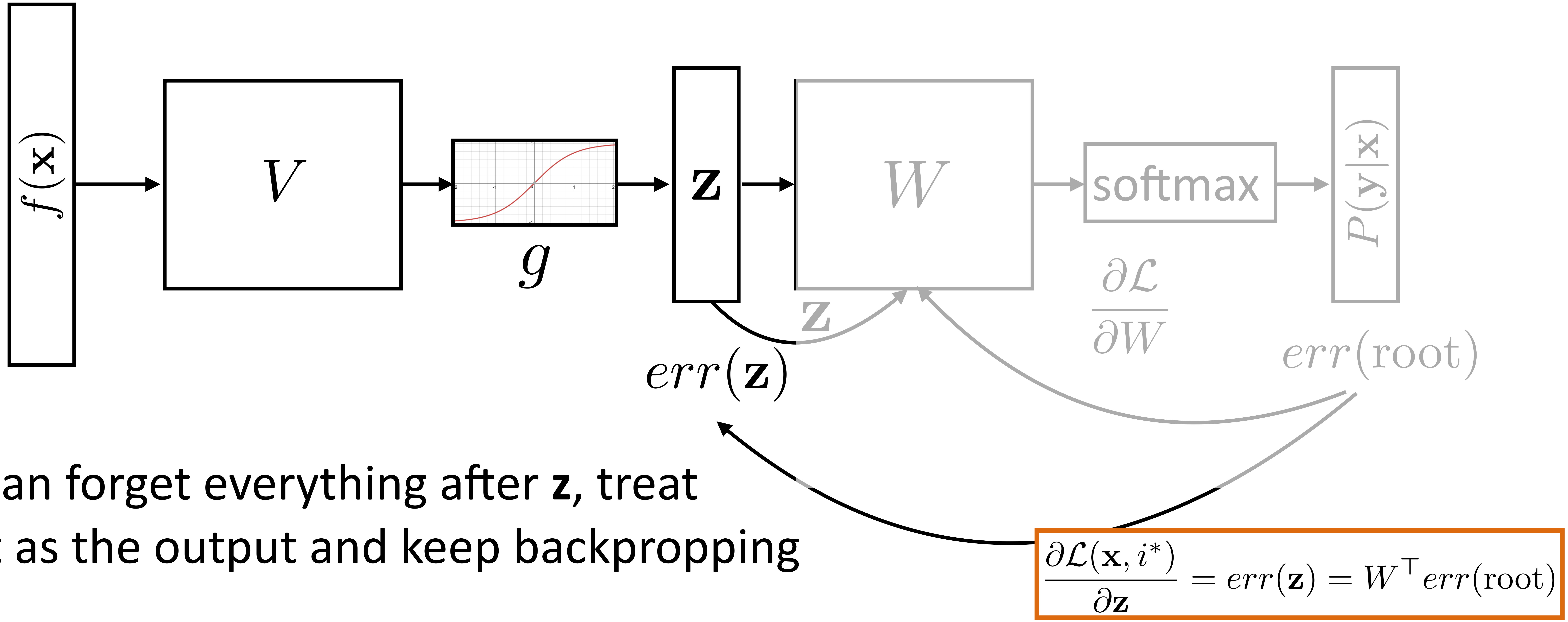
$$\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial V_{ij}} = \frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} \boxed{\frac{\partial \mathbf{z}}{V_{ij}}} \qquad \frac{\partial \mathbf{z}}{V_{ij}} = \boxed{\frac{\partial g(\mathbf{a})}{\partial \mathbf{a}}} \boxed{\frac{\partial \mathbf{a}}{\partial V_{ij}}}$$

$\mathbf{a} = Vf(\mathbf{x})$

- First term: gradient of nonlinear activation function at ***a*** (depends on current value)
- Second term: gradient of linear function
- Straightforward computation once we have *err*(**z**)

Backpropagation: Picture

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$

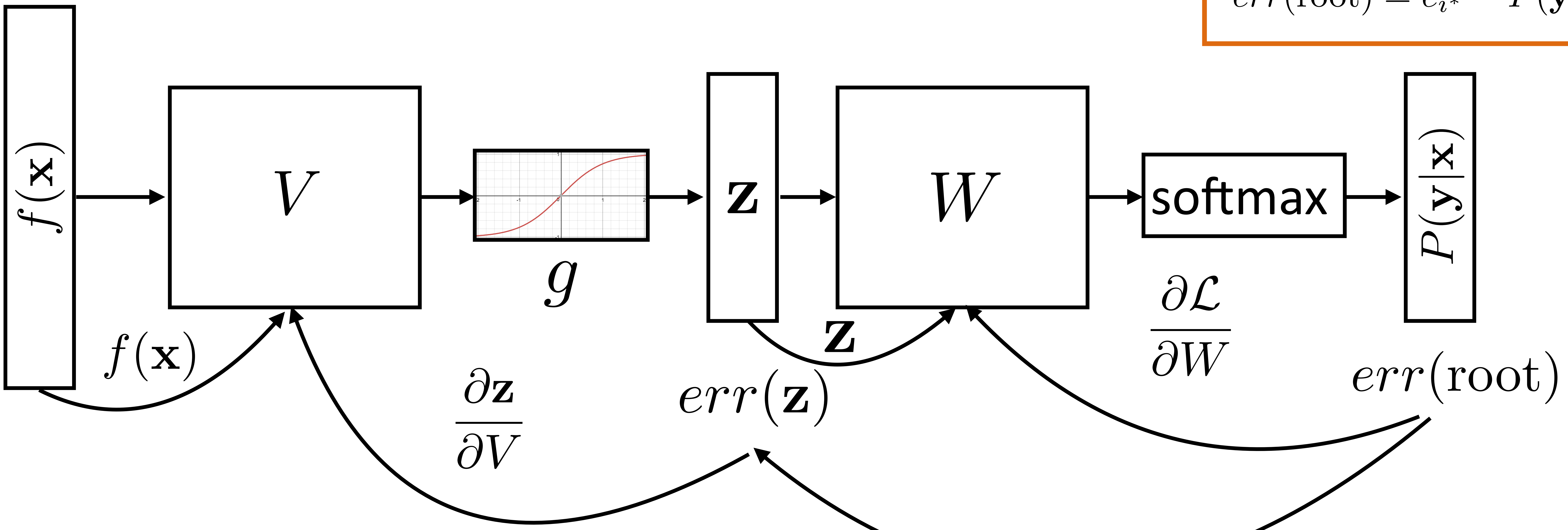


- Can forget everything after **z**, treat it as the output and keep backpropping

Backpropagation: Picture

$$P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(W g(V f(\mathbf{x})))$$

$$\boxed{err(\mathrm{root}) = e_{i^*} - P(\mathbf{y}|\mathbf{x})}$$



$$\boxed{\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial V_{ij}} = \frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial V_{ij}} = err(\mathbf{z}) \frac{\partial \mathbf{z}}{\partial V_{ij}}}$$

$$\boxed{\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} = err(\mathbf{z}) = W^\top err(\mathrm{root})}$$

Backpropagation

$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$

- Step 1: compute $err(\text{root}) = e_{i^*} - P(\mathbf{y}|\mathbf{x})$ (vector)
- Step 2: compute derivatives of *W* using *err*(root) (matrix)
- Step 3: compute $\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} = err(\mathbf{z}) = W^\top err(\text{root})$ (vector)
- Step 4: compute derivatives of *V* using *err*(**z**) (matrix)
- Step 5+: continue backpropagation (compute err(*f*(**x**)) if necessary…)

Backpropagation: Takeaways

- Gradients of output weights *W* are easy to compute — looks like logistic regression with hidden layer ***z*** as feature vector
- Can compute derivative of loss with respect to ***z*** to form an "error signal" for backpropagation
- Easy to update parameters based on "error signal" from next layer, keep pushing error signal back as backpropagation
- Need to remember the values from the forward computation

https://inst.eecs.berkeley.edu/~cs182/sp06/notes/backprop.pdf

Applications

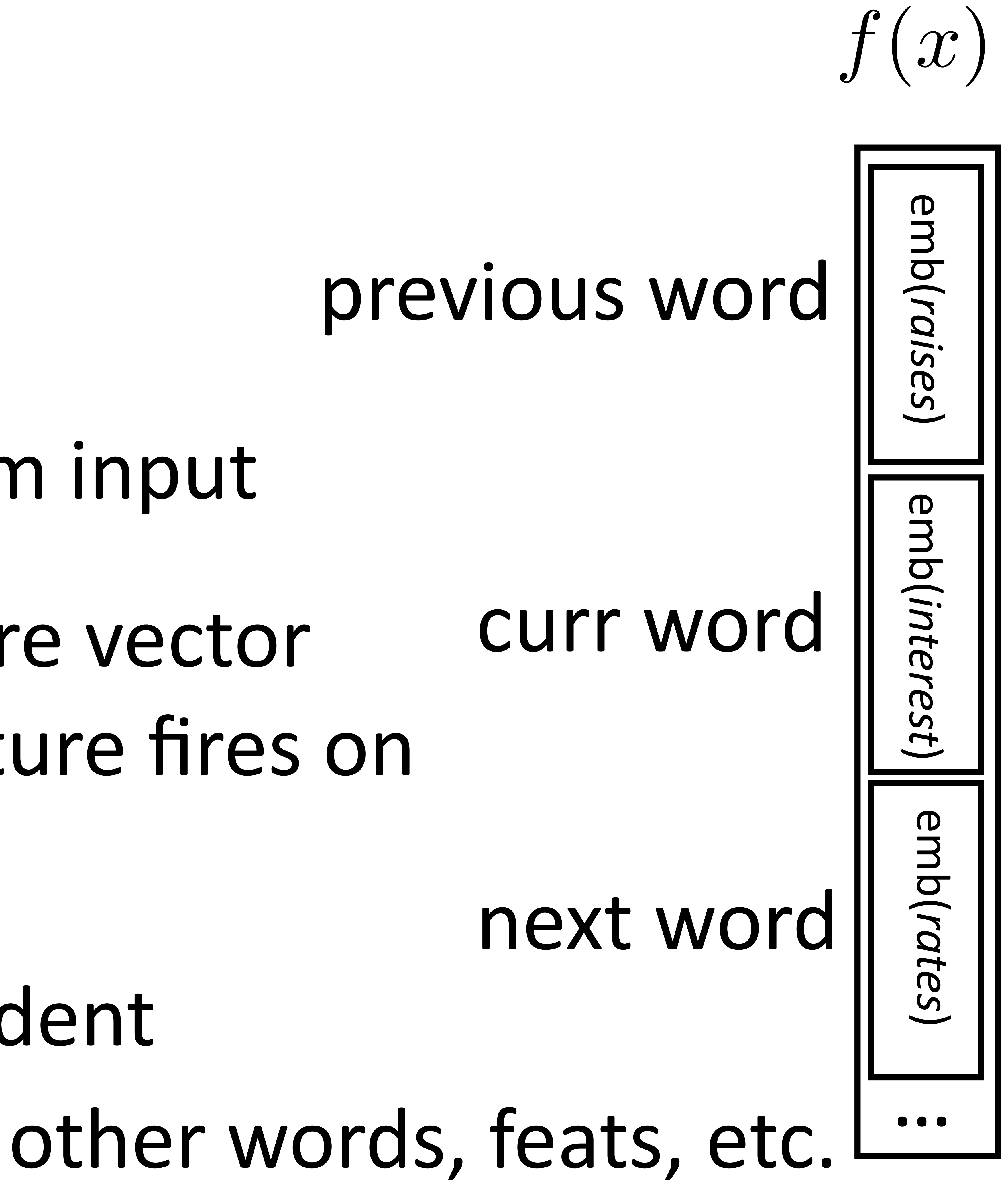
NLP with Feedforward Networks

- Part-of-speech tagging with FFNNs

??

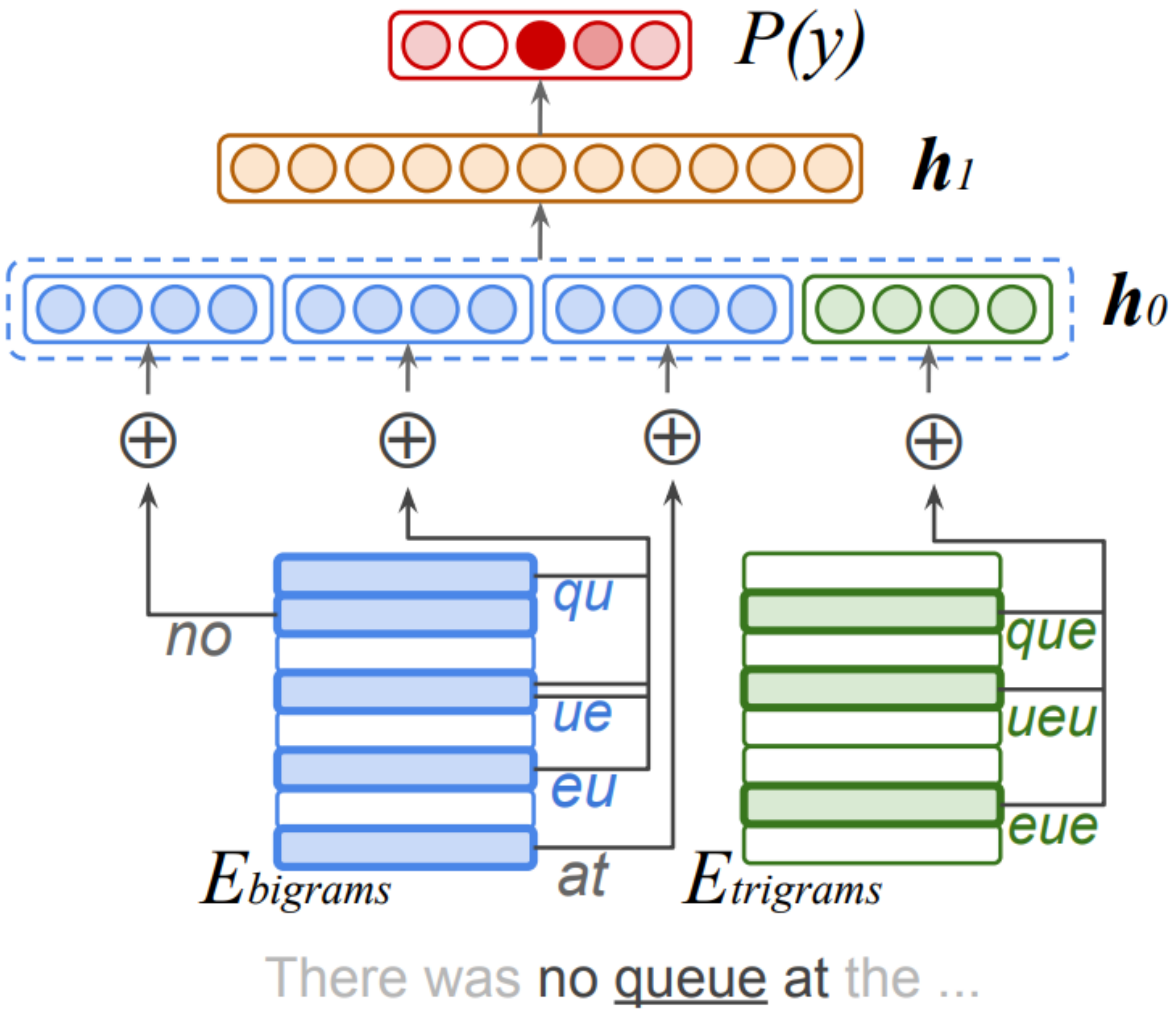
*Fed raises **interest** rates in order to …*

- Word embeddings for each word form input
- ~1000 features here — smaller feature vector than in sparse models, but every feature fires on every example
- Weight matrix learns position-dependent processing of the words



Botha et al. (2017)

NLP with Feedforward Networks



- Hidden layer mixes these different signals and learns feature conjunctions

Botha et al. (2017)

NLP with Feedforward Networks

- Multilingual tagging results:

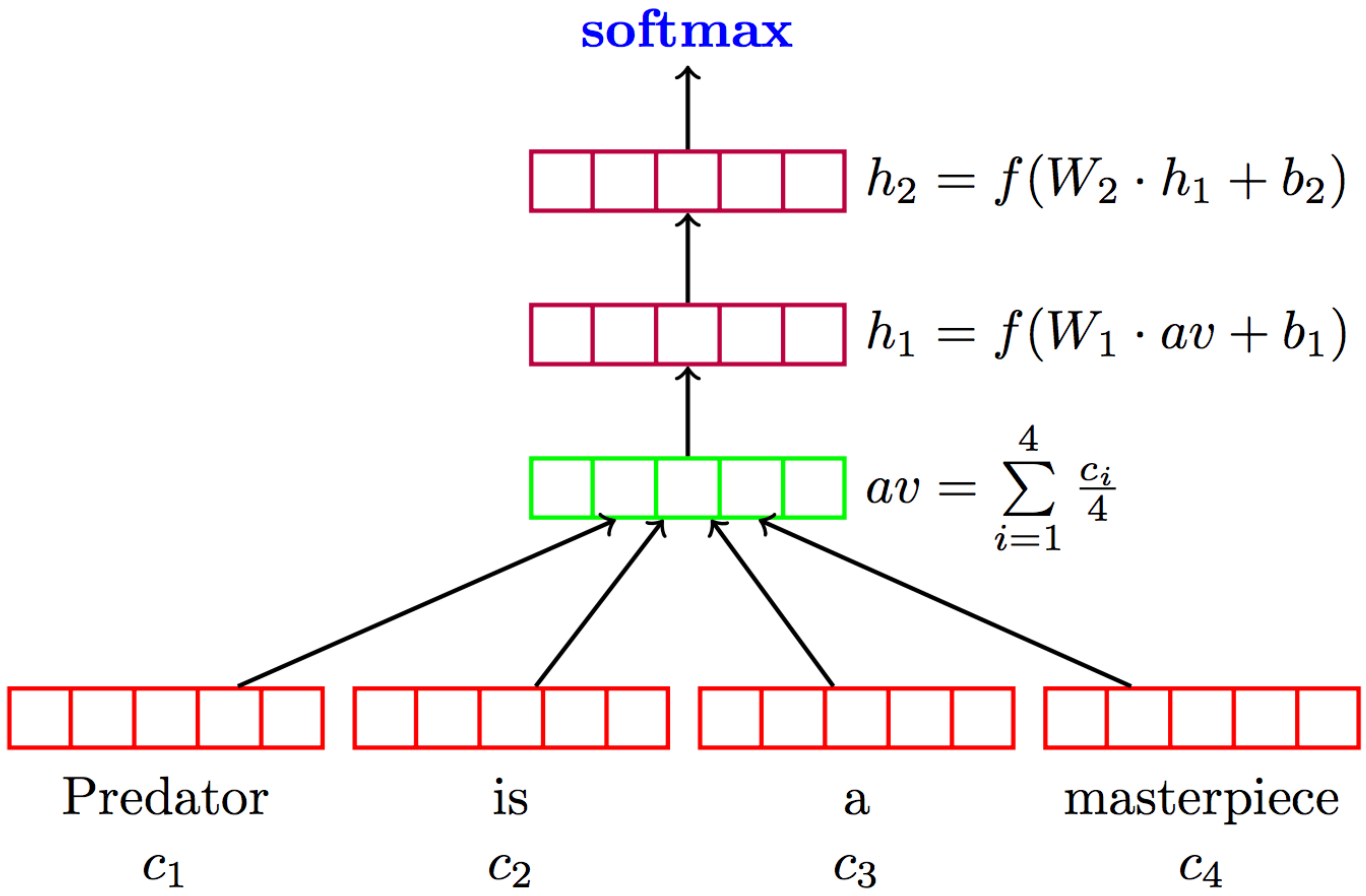
Model	Acc.	Wts.	MB	Ops.
Gillick et al. (2016)	95.06	900k	-	6.63m
Small FF	94.76	241k	0.6	0.27m
+Clusters	95.56	261k	1.0	0.31m
$\frac{1}{2}$ Dim.	95.39	143k	0.7	0.18m

- Gillick used LSTMs; this is smaller, faster, and better

Botha et al. (2017)

Sentiment Analysis

- Deep Averaging Networks: feedforward neural network on average of word embeddings from input



Iyyer et al. (2015)

Sentiment Analysis

Model	RT	SST fine	SST bin	IMDB	Time (s)
DAN-ROOT	—	46.9	85.7	—	**31**
DAN-RAND	77.3	45.4	83.2	88.8	136
$\boxed{\text{DAN}}$	80.3	47.7	86.3	89.4	136
NBOW-RAND	76.2	42.3	81.4	88.9	91
NBOW	79.0	43.6	83.6	89.0	91
BiNB	—	41.9	83.1	—	—
$\boxed{\text{NBSVM-bi}}$	79.4	—	—	91.2	—
RecNN*	77.7	43.2	82.4	—	—
RecNTN*	—	45.7	85.4	—	—
DRecNN	—	49.8	86.6	—	431
TreeLSTM	—	**50.6**	86.9	—	—
DCNN*	—	48.5	86.9	89.4	—
PVEC*	—	48.7	87.8	**92.6**	—
$\boxed{\text{CNN-MC}}$	**81.1**	47.4	**88.1**	—	2,452
WRRBM*	—	—	—	89.2	—

Iyyer et al. (2015)

Bag-of-words

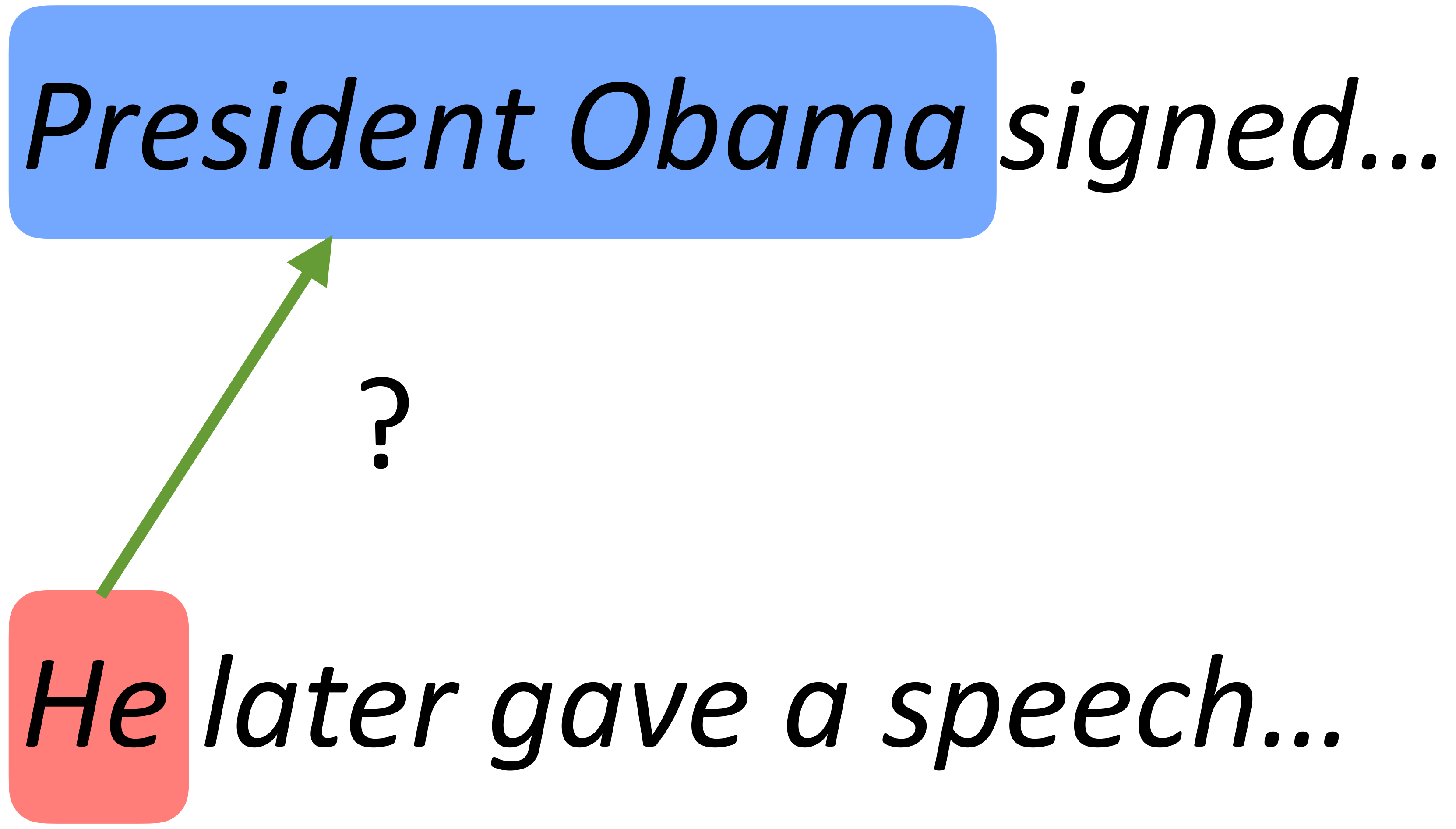
Wang and Manning (2012)

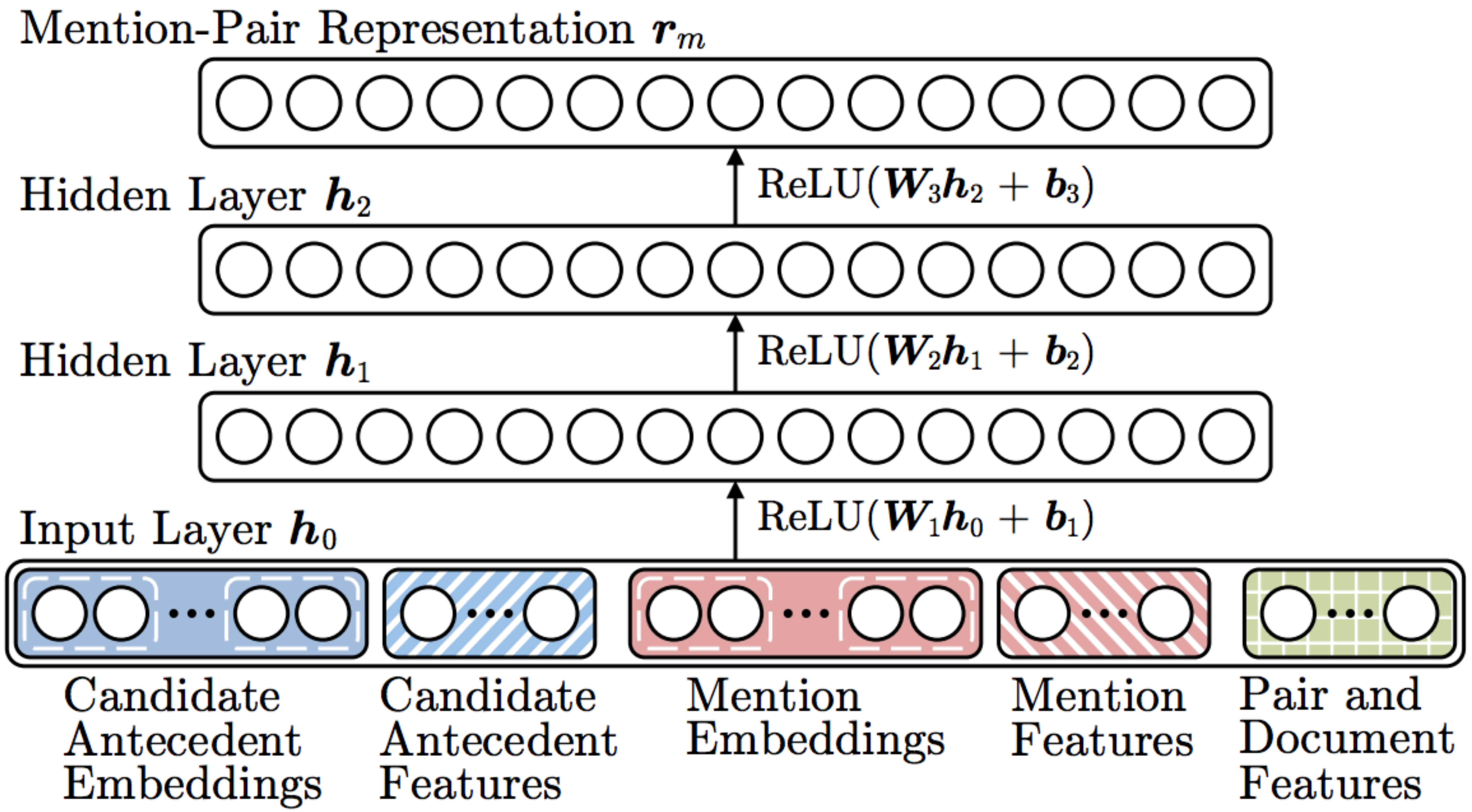
Tree RNNs / CNNS / LSTMS

Kim (2014)

Coreference Resolution

- Feedforward networks identify coreference arcs





Clark and Manning (2015), Wiseman et al. (2015)

Training Tips

Computation Graphs

- Computing gradients is hard!
- Automatic differentiation: instrument code to keep track of derivatives

$$\texttt{y = x * x} \xrightarrow[\text{codegen}]{} \texttt{(y,dy) = (x * x, 2 * x * dx)}$$

- Computation is now something we need to reason about symbolically
- Use a library like PyTorch or TensorFlow. This class: PyTorch

Computation Graphs in Pytorch

- Define forward pass for $P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(Wg(Vf(\mathbf{x})))$

```
class FFNN(nn.Module):
    def __init__(self, inp, hid, out):
        super(FFNN, self).__init__()
        self.V = nn.Linear(inp, hid)
        self.g = nn.Tanh()
        self.W = nn.Linear(hid, out)
        self.softmax = nn.Softmax(dim=0)

    def forward(self, x):
        return self.softmax(self.W(self.g(self.V(x))))
```

Computation Graphs in Pytorch

$$P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(Wg(Vf(\mathbf{x})))$$

```
ei*: one-hot vector
of the label
(e.g., [0, 1, 0])
```

```
ffnn = FFNN()
def make_update(input, gold_label):
   ffnn.zero_grad() # clear gradient variables
   probs = ffnn.forward(input)
   loss = torch.neg(torch.log(probs)).dot(gold_label)
   loss.backward()
   optimizer.step()
```

Training a Model

Define a computation graph

For each epoch:

  For each batch of data:

    Compute loss on batch

    Autograd to compute gradients and take step

Decode test set

Batching

- Batching data gives speedups due to more efficient matrix operations

- Need to make the computation graph process a batch at the same time

```
# input is [batch_size, num_feats]
# gold_label is [batch_size, num_classes]
def make_update(input, gold_label)
    ...
    probs = ffnn.forward(input) # [batch_size, num_classes]
    loss = torch.sum(torch.neg(torch.log(probs)).dot(gold_label))
    ...
```

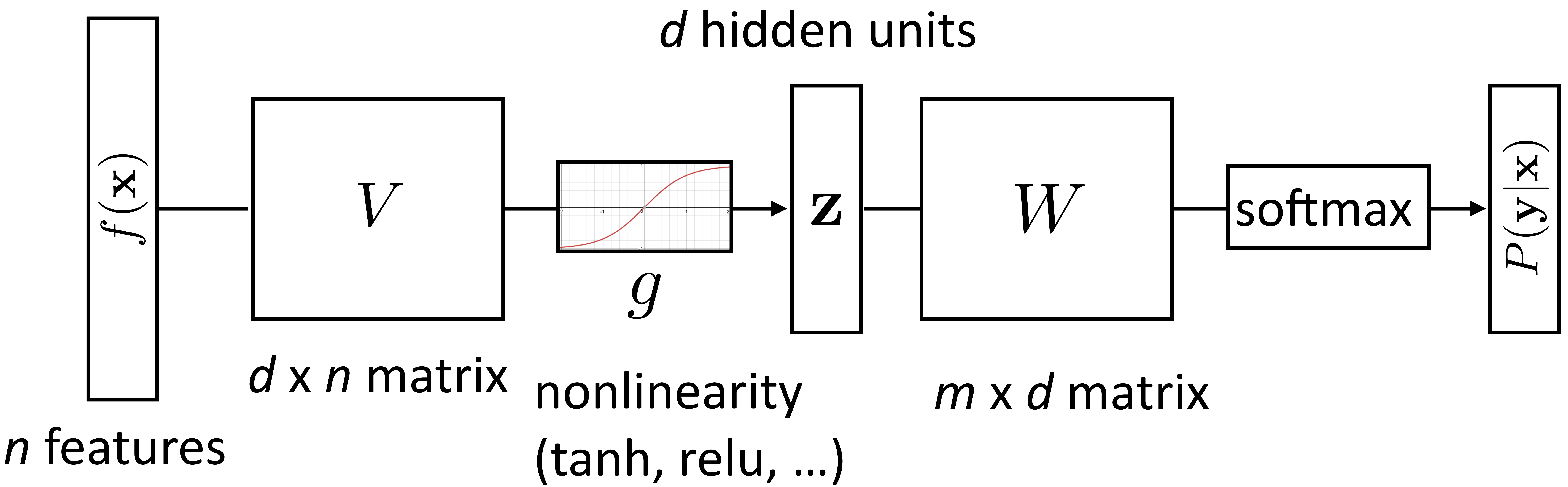
- Batch sizes from 1-100 often work well

Training Basics

- Basic formula: compute gradients on batch, use first-order optimization method (SGD, Adagrad, etc.)
- How to initialize? How to regularize? What optimizer to use?
- This lecture: some practical tricks. Take deep learning or optimization courses to understand this further

How does initialization affect learning?

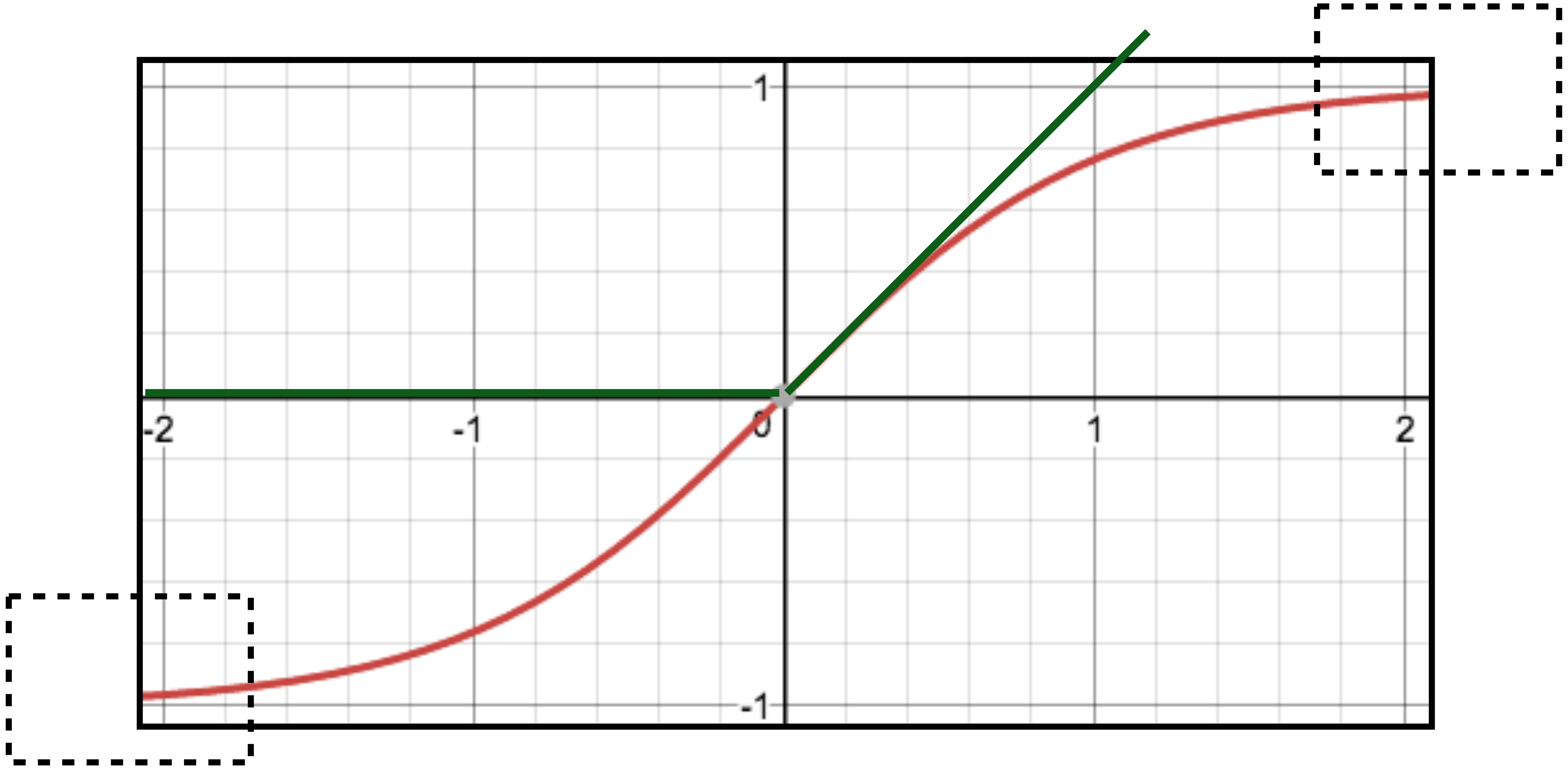
$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



- How do we initialize V and W? What consequences does this have?
- *Nonconvex* problem, so initialization matters!

How does initialization affect learning?

- Nonlinear model…how does this affect things?



- Tanh: If cell activations are too large in absolute value, gradients are small
- ReLU: larger dynamic range (all positive numbers), but can produce big values, and can break down if everything is too negative ("dead" ReLU)

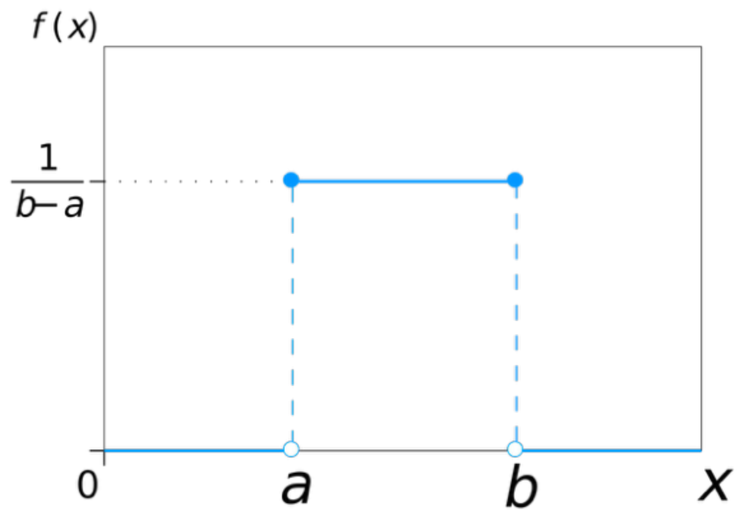
Krizhevsky et al. (2012)

Initialization

1) Can't use zeroes for parameters to produce hidden layers: all values in that hidden layer are always 0 and have gradients of 0, never change

2) Initialize too large and cells are saturated

- Can do random uniform / normal initialization with appropriate scale
- Xavier initializer: $U\left[-\sqrt{\frac{6}{\text{fan-in} + \text{fan-out}}}, +\sqrt{\frac{6}{\text{fan-in} + \text{fan-out}}}\right]$
  - Want variance of inputs and gradients for each layer to be the same



Mean & Standard Deviation

$$\mu = \frac{a+b}{2} \quad \text{and} \quad \sigma = \frac{b-a}{\sqrt{12}}$$

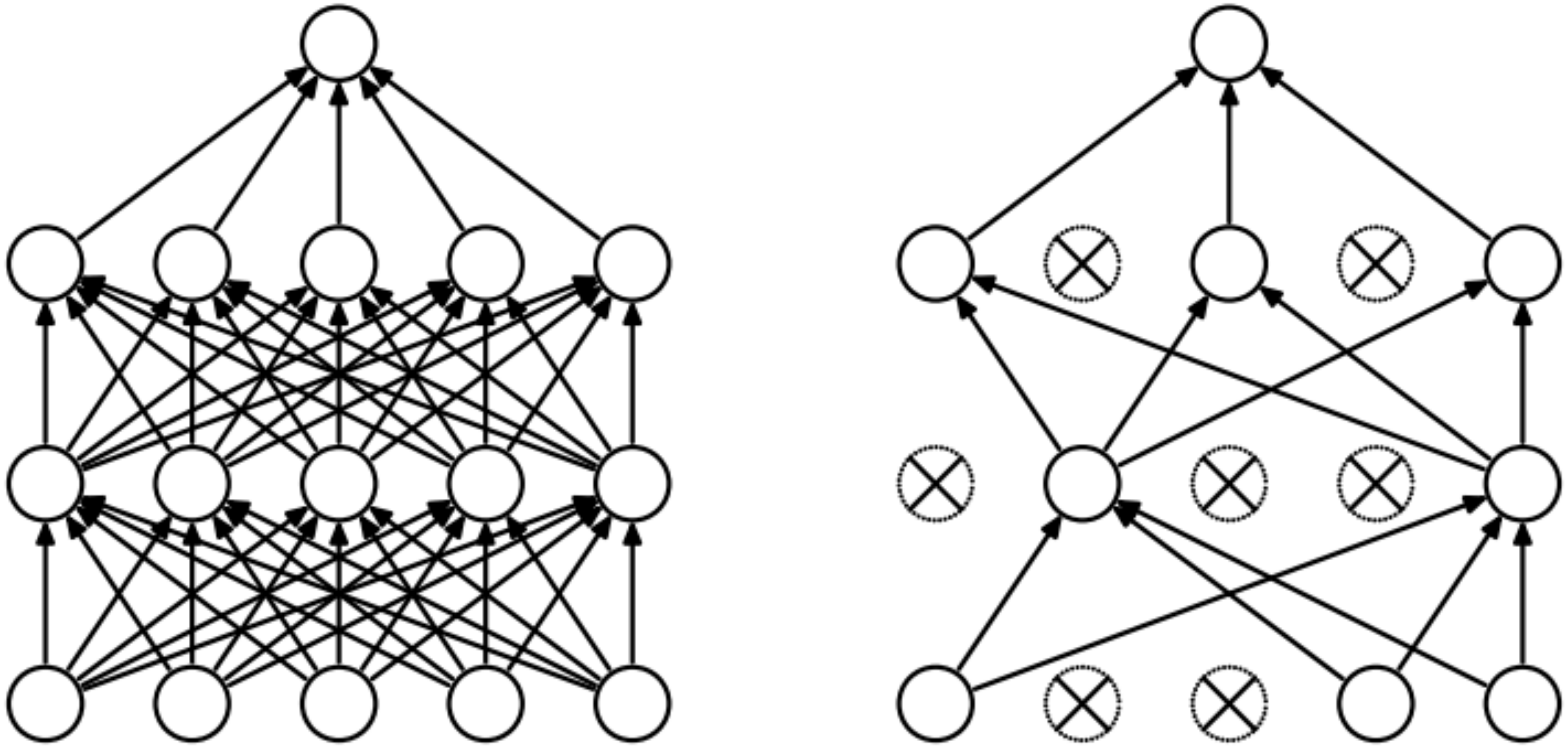
https://mmuratarat.github.io/2019-02-25/xavier-glorot-he-weight-init
https://arxiv.org/pdf/1502.01852v1.pdf

Regularization: Dropout

- Probabilistically zero out parts of the network during training to prevent overfitting, use whole network at test time

- Form of stochastic regularization

- Similar to benefits of ensembling: network needs to be robust to missing signals, so it has redundancy



(a) Standard Neural Net

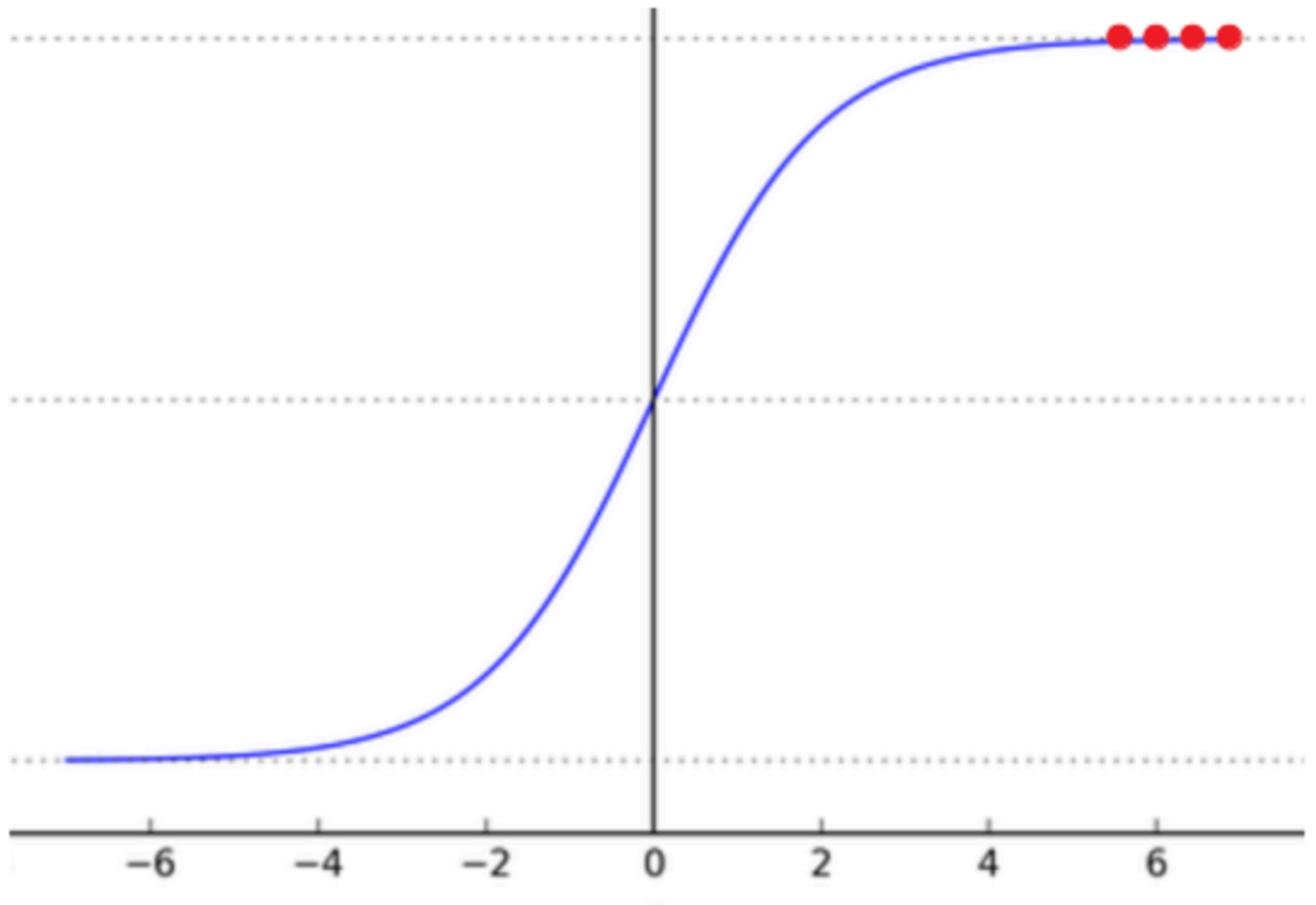
(b) After applying dropout.

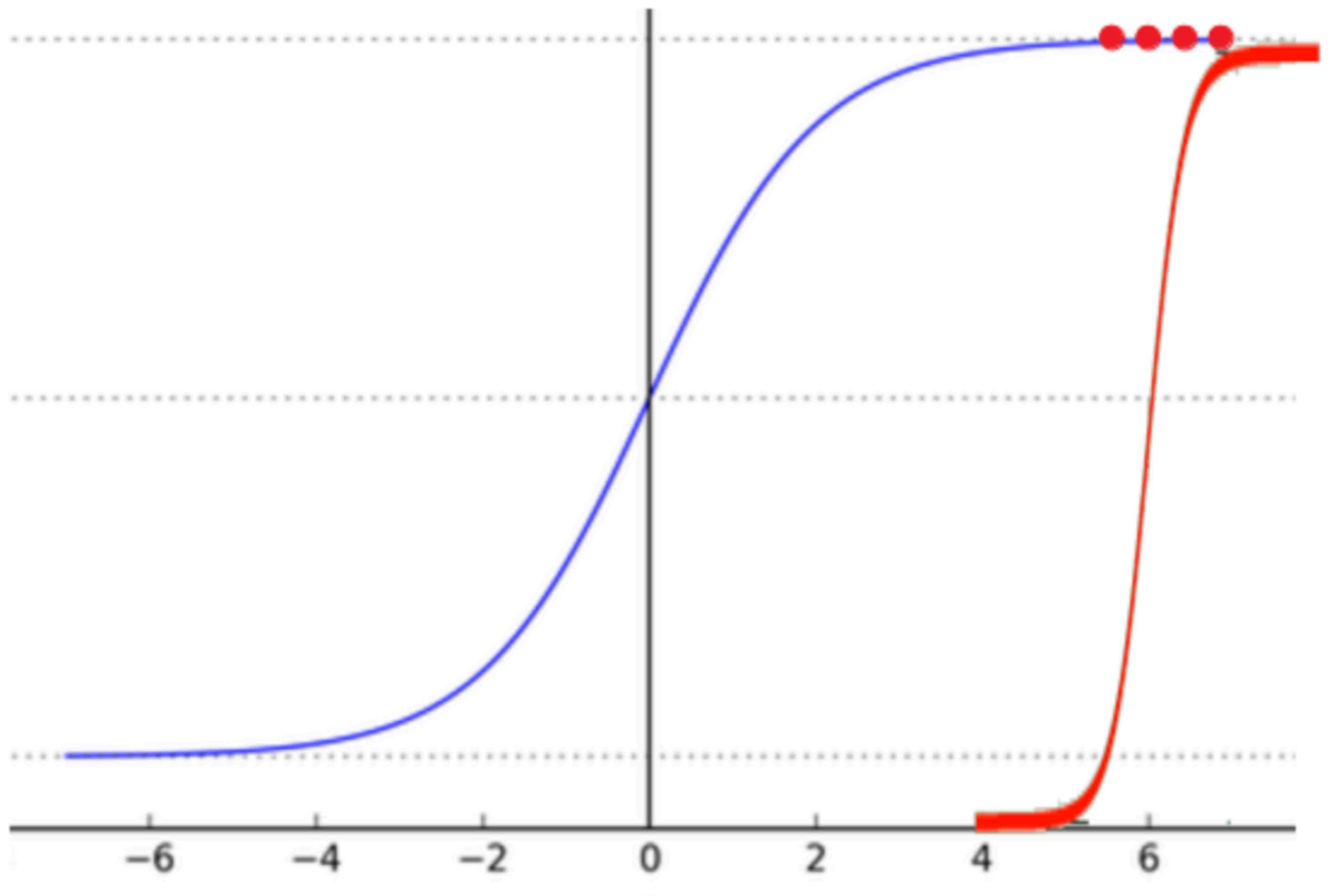
- One line in Pytorch/Tensorflow

Srivastava et al. (2014)

Batch Normalization

- Batch normalization (Ioffe and Szegedy, 2015): periodically shift+rescale each layer to have mean 0 and variance 1 over a batch (useful if net is deep)

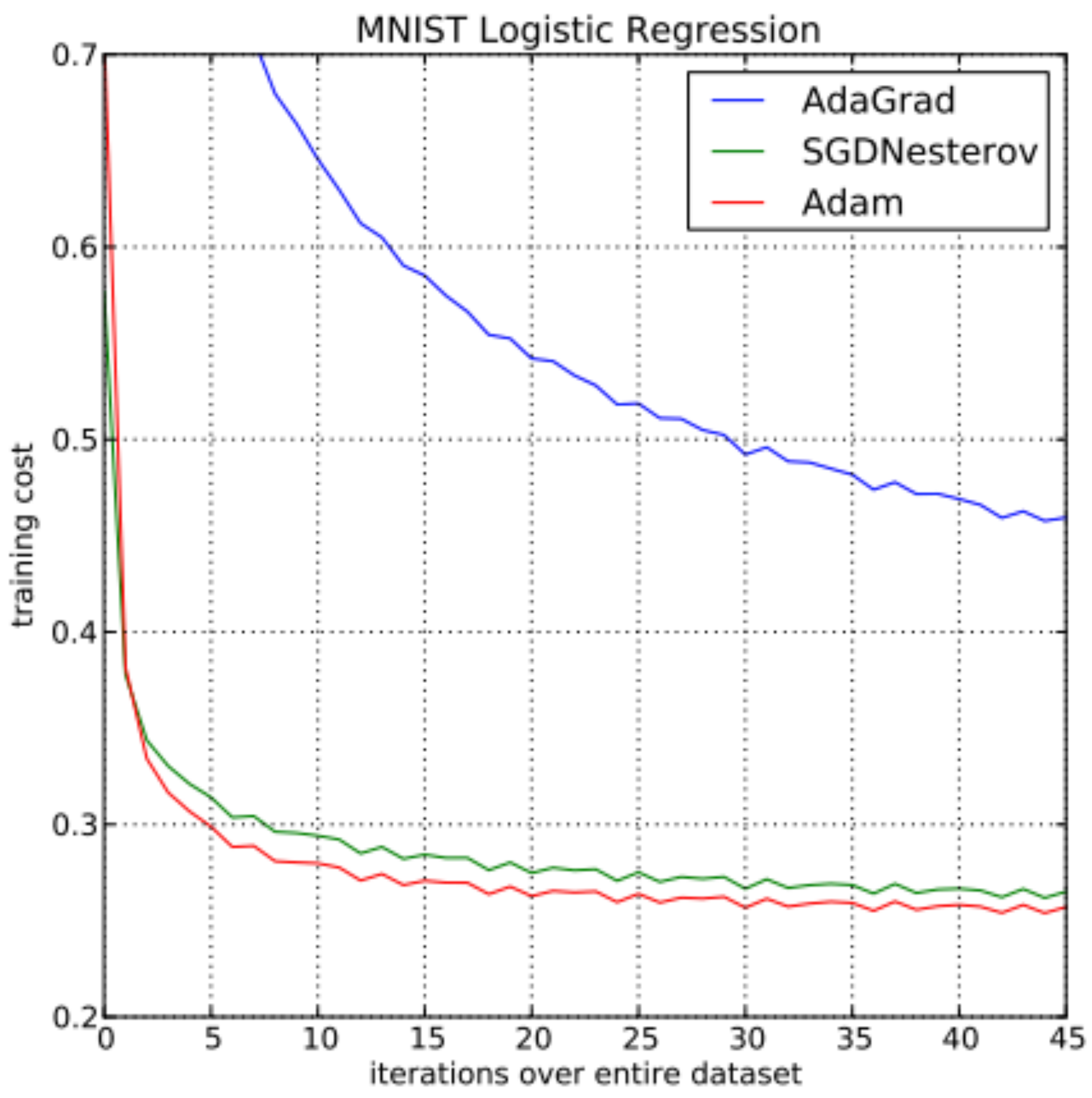


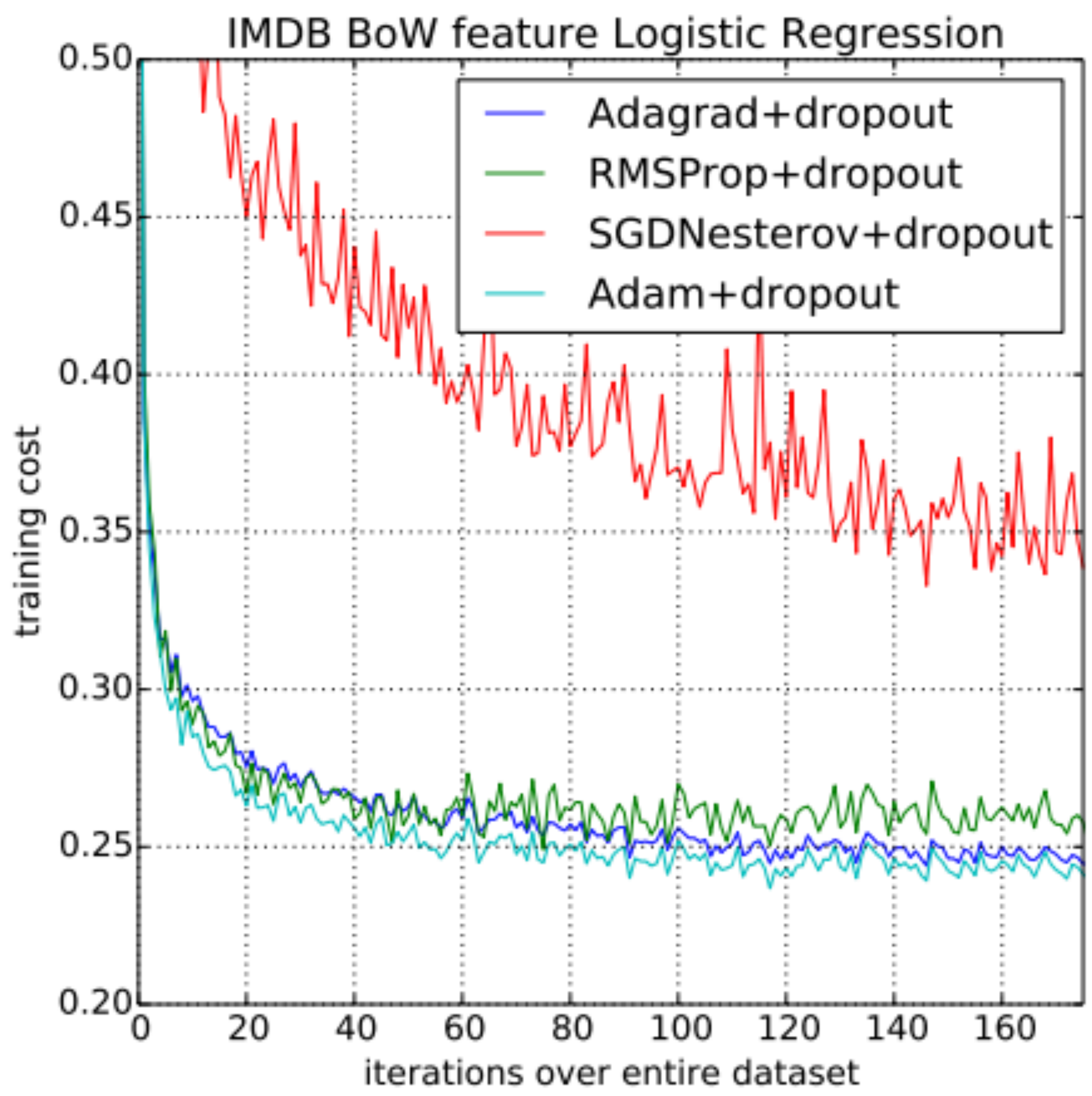


https://medium.com/@shiyan/xavier-initialization-and-batch-normalization-my-understanding-b5b91268c25c

Optimizer

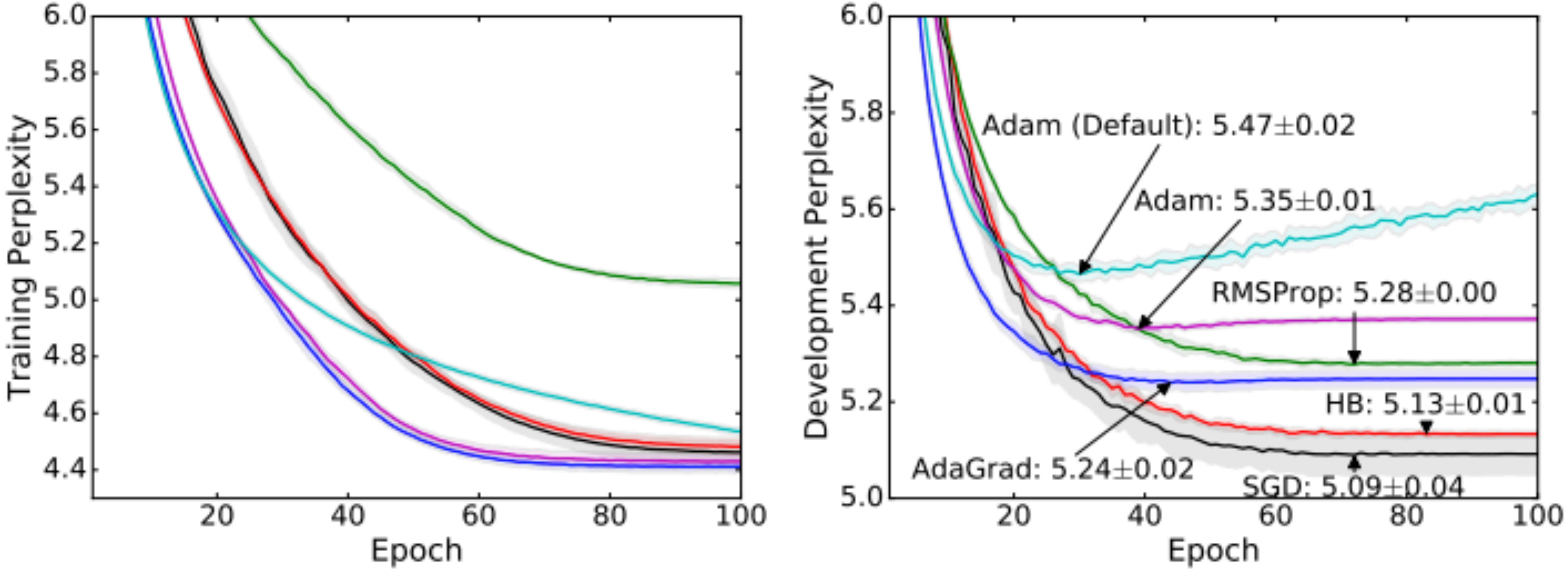
- Adam (Kingma and Ba, ICLR 2015) is very widely used
- Adaptive step size like Adagrad, incorporates momentum





Optimizer

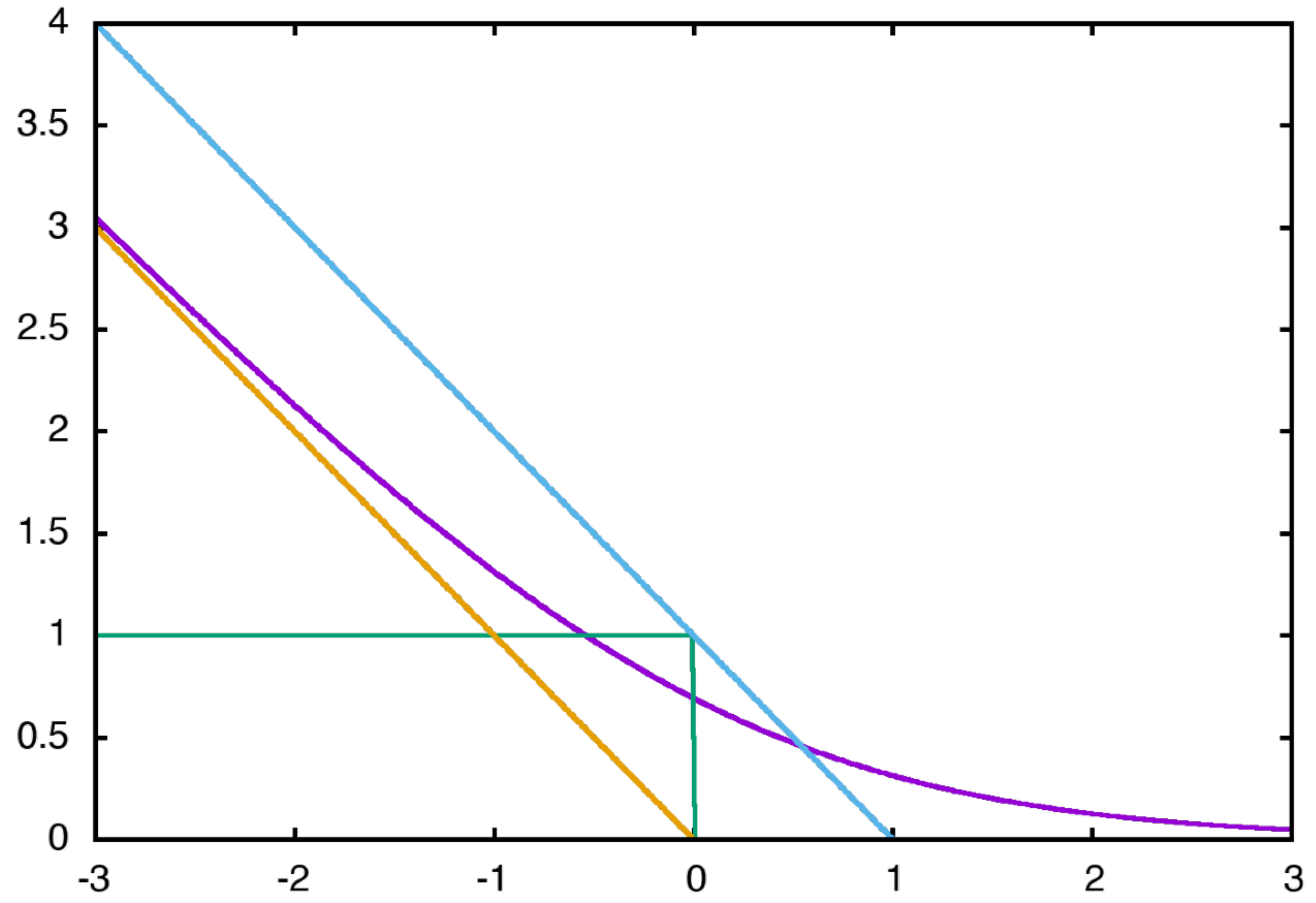
- Wilson et al. NIPS 2017: adaptive methods can actually perform badly at test time (Adam is in pink, SGD in black)
- Check dev set periodically, decrease learning rate if not making progress



(e) Generative Parsing (Training Set) (f) Generative Parsing (Development Set)

Four Elements of NNs

- Model: feedforward, RNNs, CNNs can be defined in a uniform framework
- Objective: many loss functions look similar, just changes the last layer of the neural network



- Inference: define the network, your library of choice takes care of it (mostly…)
- Training: lots of choices for optimization/hyperparameters

Next Class

- Word representations
- word2vec/GloVe
- Evaluating word embeddings