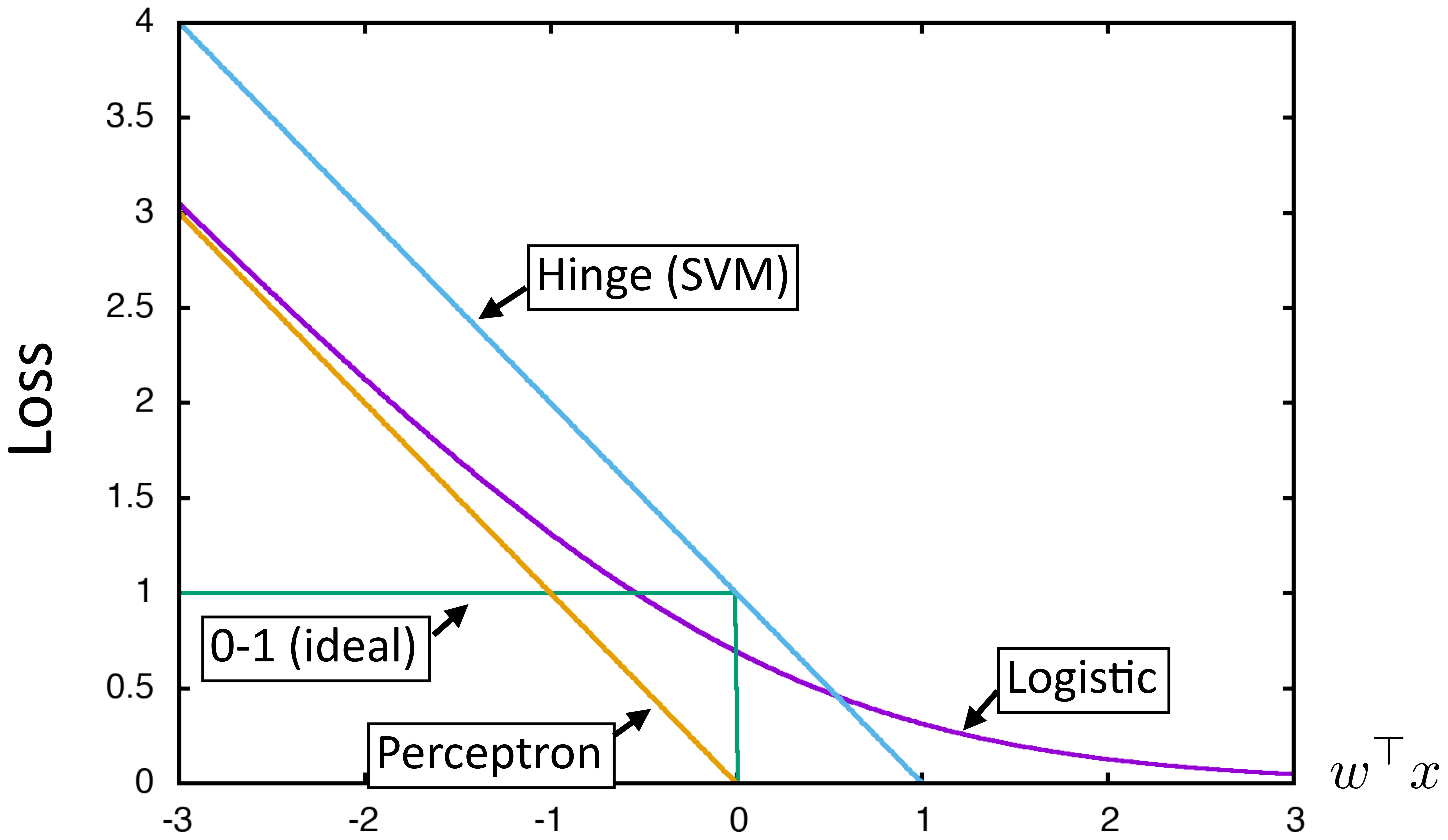
Neural Networks

Wei Xu

(many slides from Greg Durrett and Philipp Koehn)

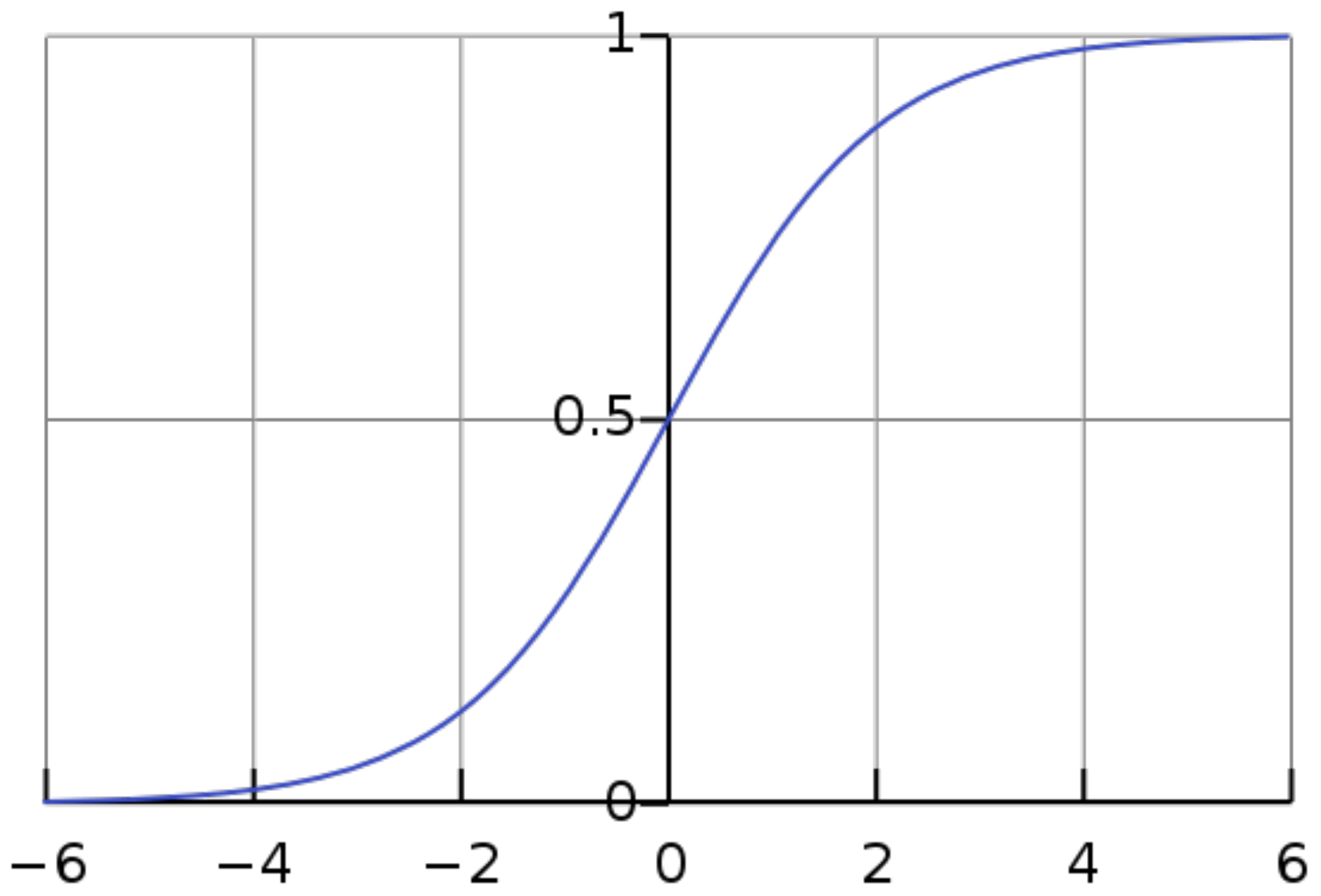
Recap: Loss Functions



Recap: Logistic Regression

$$P(y=+|x) = \text{logistic}(w^\top x)$$

$$P(y=+|x) = \frac{\exp(\sum_{i=1}^{n} w_i x_i)}{1+\exp(\sum_{i=1}^{n} w_i x_i)}$$



- To learn weights: maximize discriminative log likelihood of data P(y|x)

$$\mathcal{L}(x_j, y_j=+) = \log P(y_j=+|x_j)$$

$$= \underbrace{\sum_{i=1}^{n}}_{\text{sum over features}} w_i x_{ji} - \log\left(1+\exp\left(\sum_{i=1}^{n} w_i x_{ji}\right)\right)$$

Recap: Multiclass Logistic Regression

$$P_w(y|x) = \frac{\exp\left(w^\top f(x,y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(w^\top f(x,y')\right)}$$

sum over output space to normalize

Why? Interpret raw classifier scores as **probabilities**

too many drug trials, too few patients

$w^\top f(x,y)$		probabilities must be >= 0 → unnormalized probabilities		probabilities must sum to 1 → probabilities		correct (gold) probabilities
Health: +2.2	exp →	6.05	normalize →	0.21	log(0.21) = - 1.56	1.00
Sports: +3.1		22.2		0.77	← compare →	0.00
Science: -0.6		0.55		0.02	$\mathcal{L}(x_j, y_j^*) = \log P(y_j^*\|x_j)$	0.00

Recap: Multiclass Logistic Regression

$$P_w(y|x) = \frac{\exp\left(w^\top f(x,y)\right)}{\sum_{y' \in \mathcal{Y}} \exp\left(w^\top f(x,y')\right)}$$

sum over output space to normalize

- Training: maximize $\mathcal{L}(x,y) = \sum_{j=1}^{n} \log P(y_j^*|x_j)$

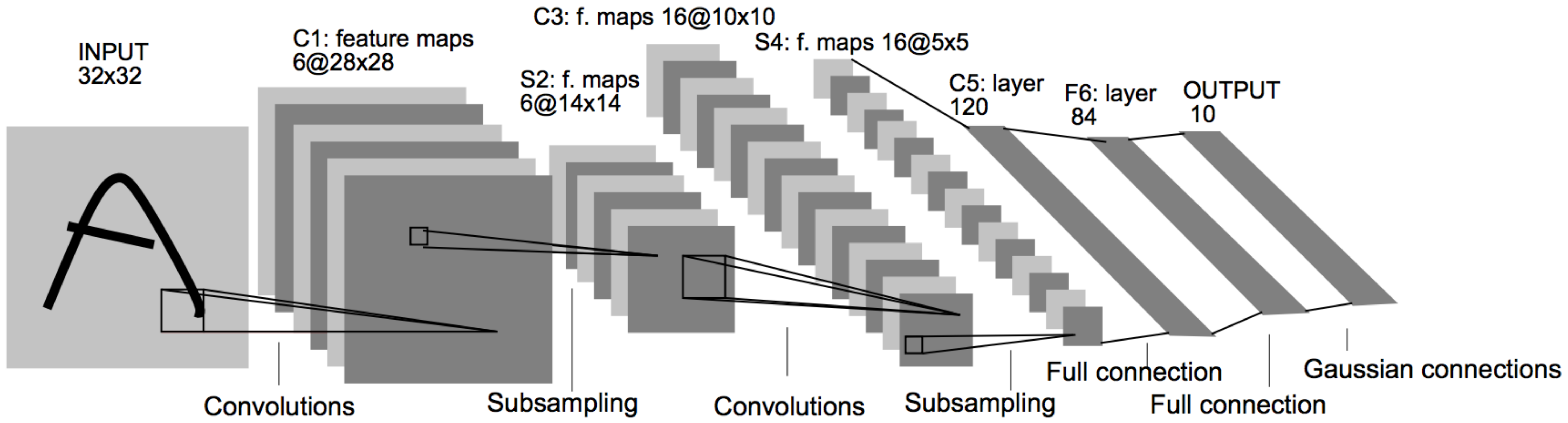
$$= \sum_{j=1}^{n} \left(w^\top f(x_j, y_j^*) - \log \sum_{y} \exp(w^\top f(x_j, y)) \right)$$

This Lecture

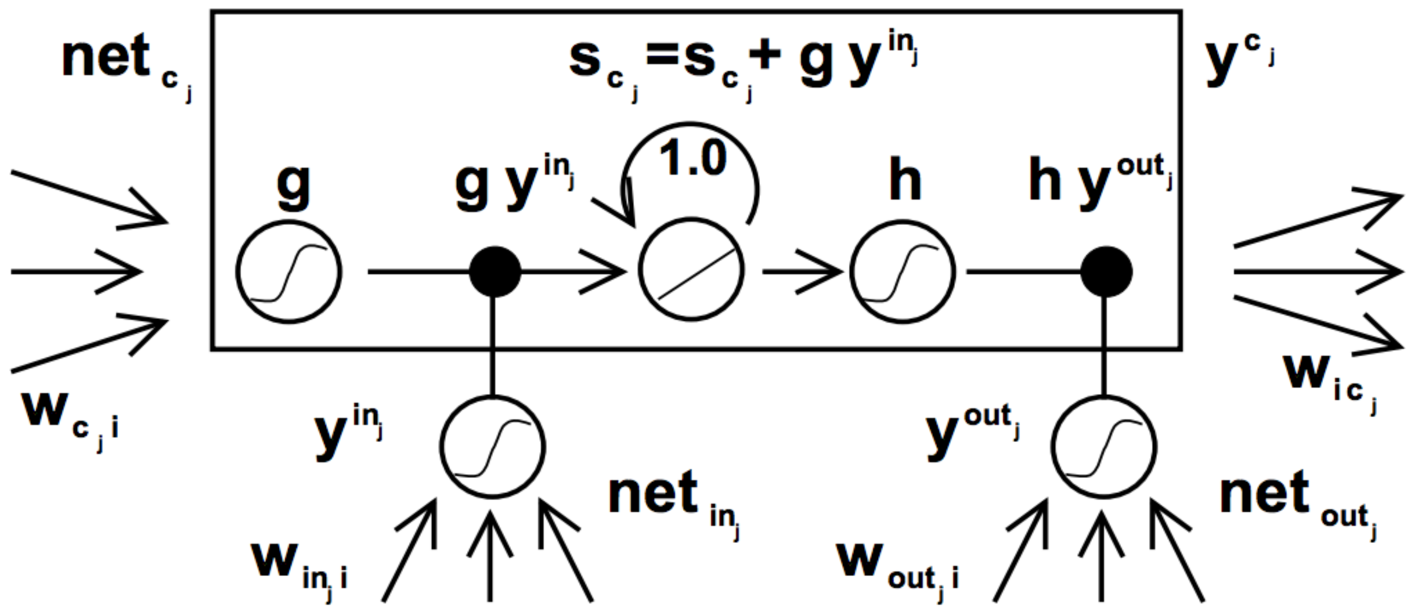
- Neural network history
- Neural network basics
- Feedforward neural networks + backpropagation
- Applications
- Implementing neural networks (if time)

History: NN “dark ages”

- ConvNets: applied to MNIST by LeCun in 1998



- LSTMs: Hochreiter and Schmidhuber (1997)



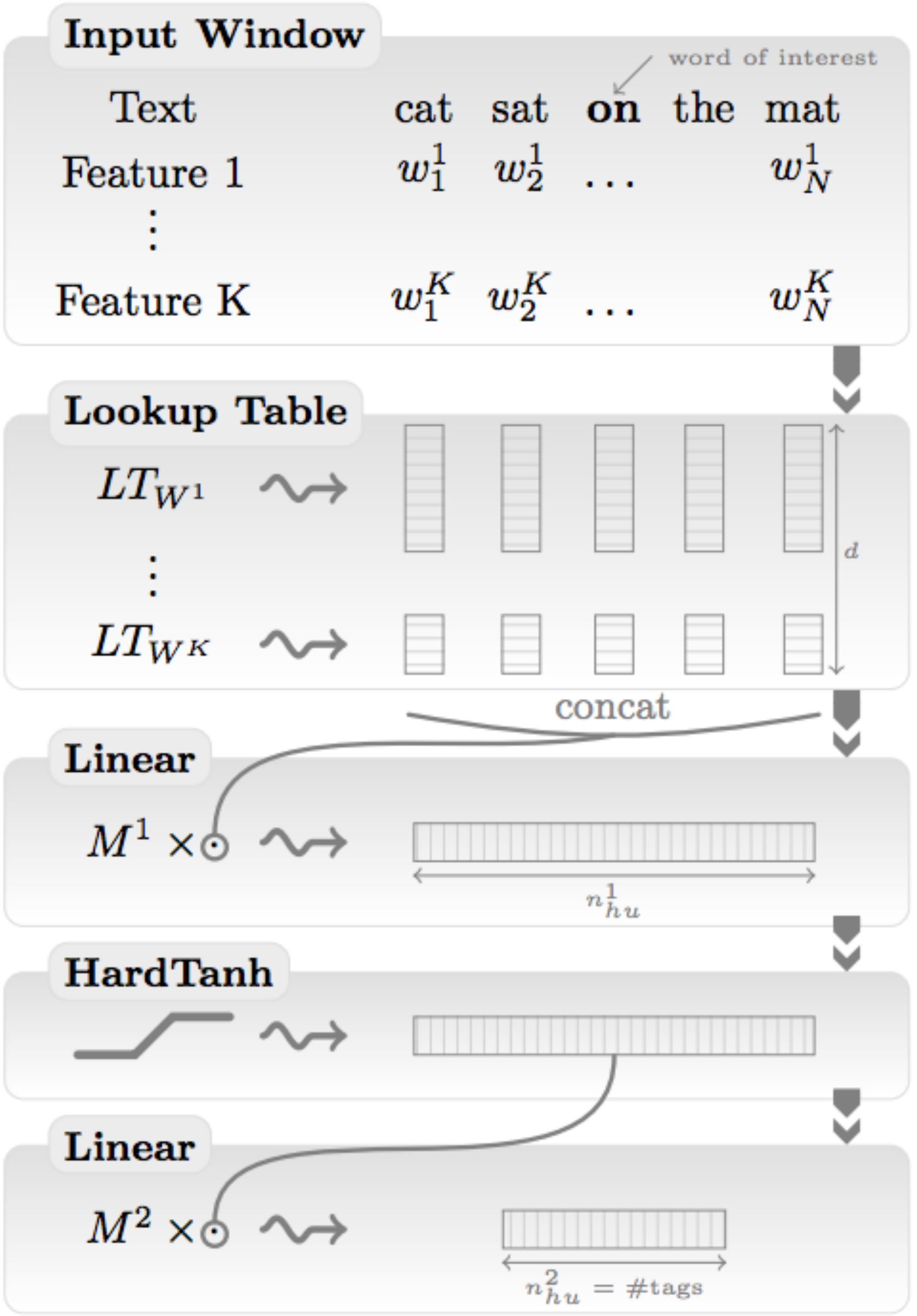
- Henderson (2003): neural shift-reduce parser, not SOTA

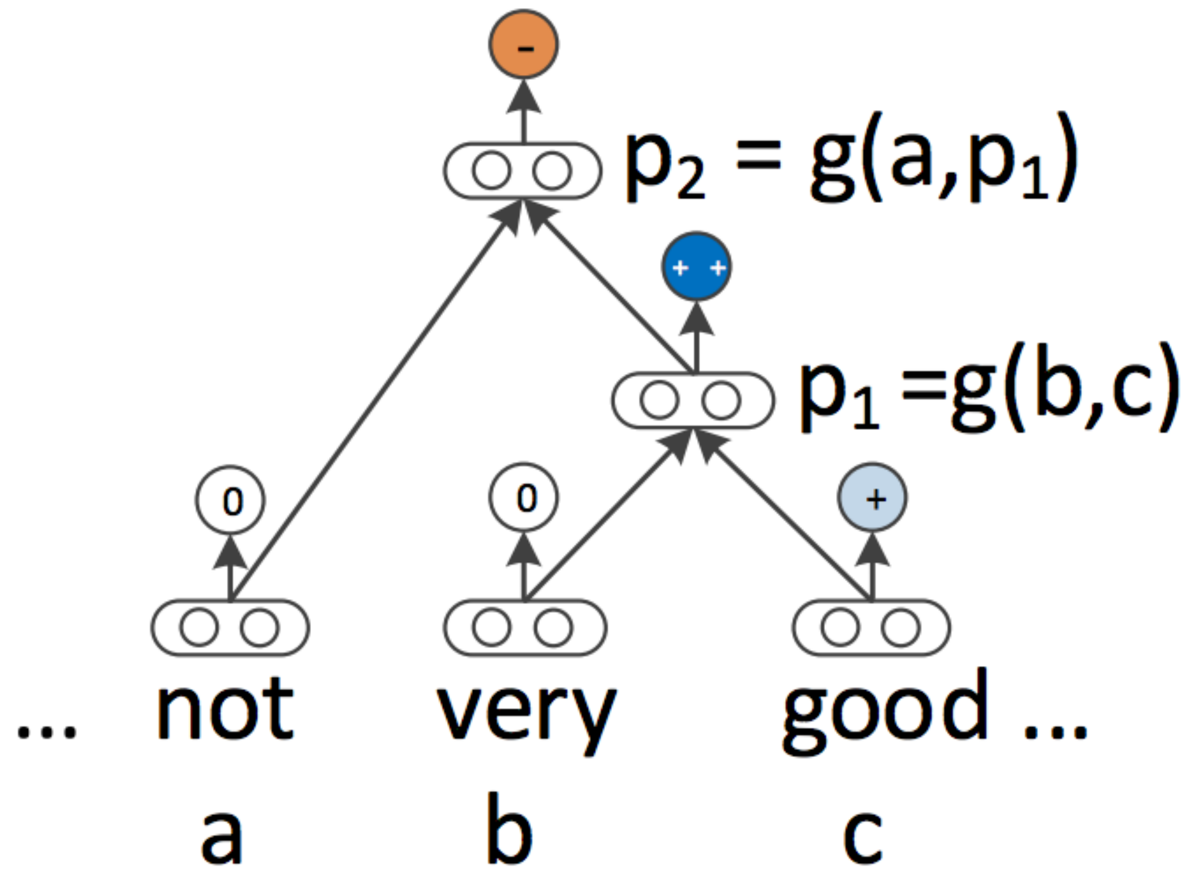
https://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-and-deep-learning/

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

Neural Net Basics

Neural Networks: motivation

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

- 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***

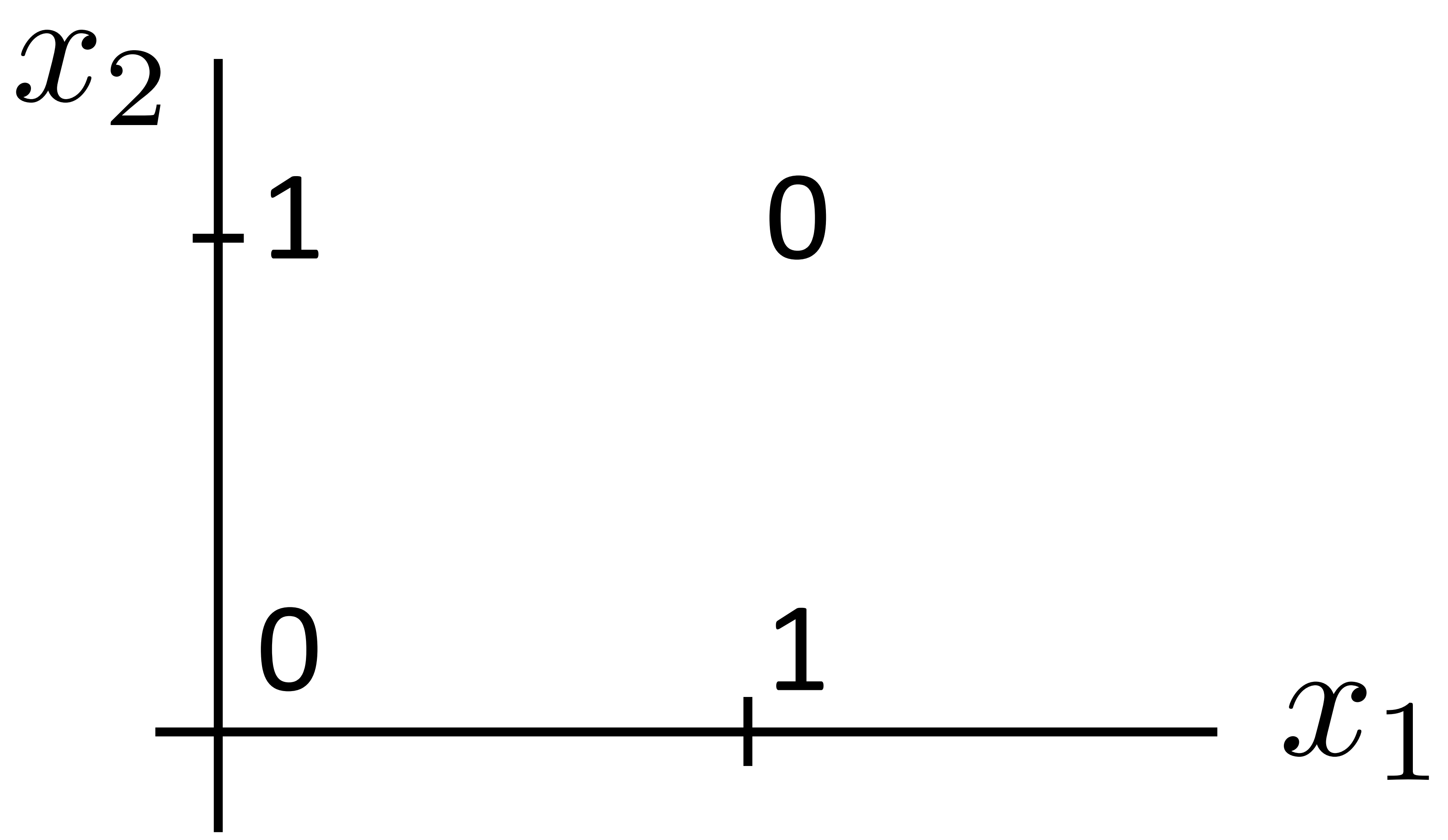
I[contains *not* & contains *good*]

Neural Networks: XOR

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

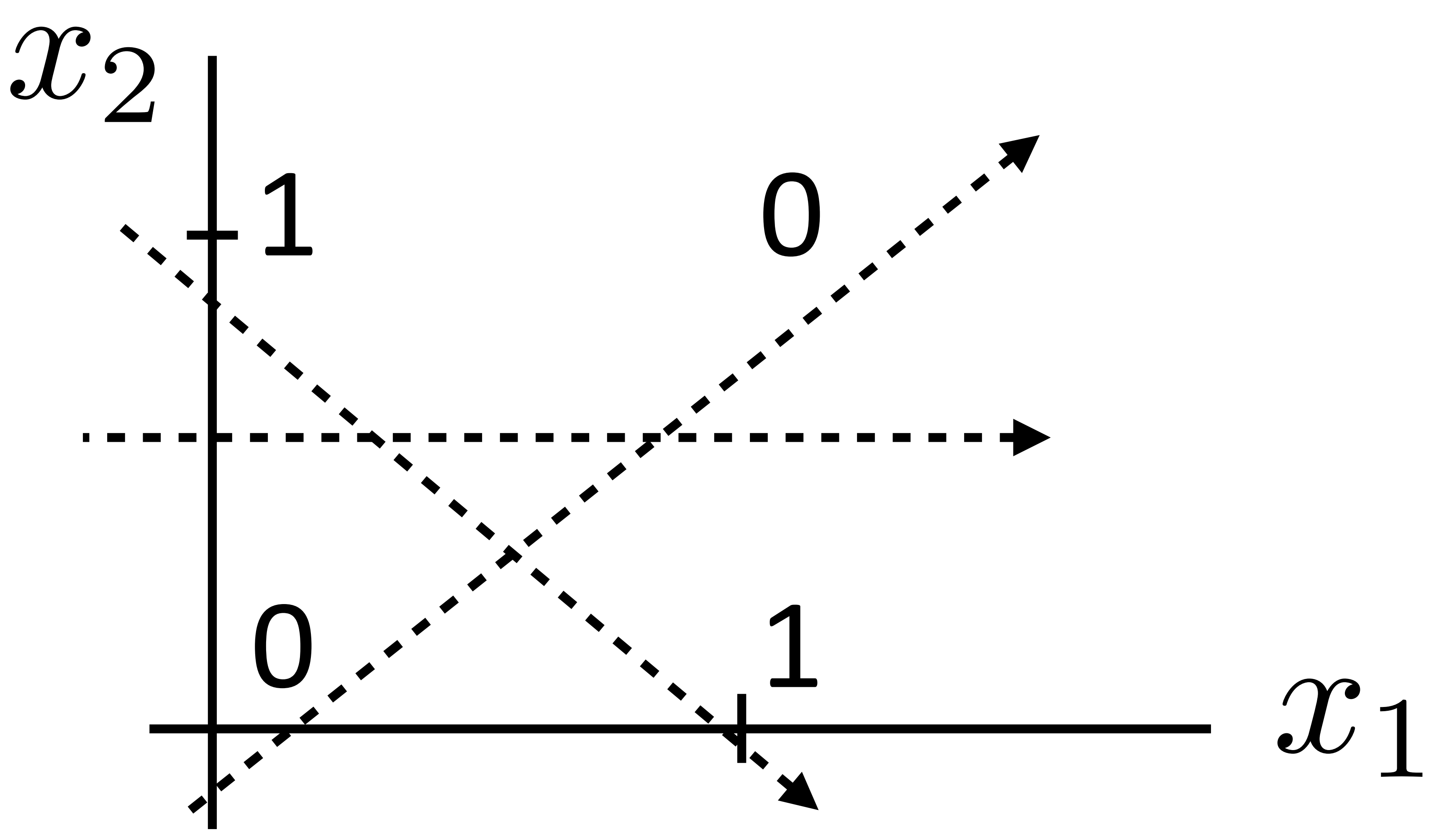
  $(\text{generally } \mathbf{x} = (x_1, \ldots, x_m))$
- Output y

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



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



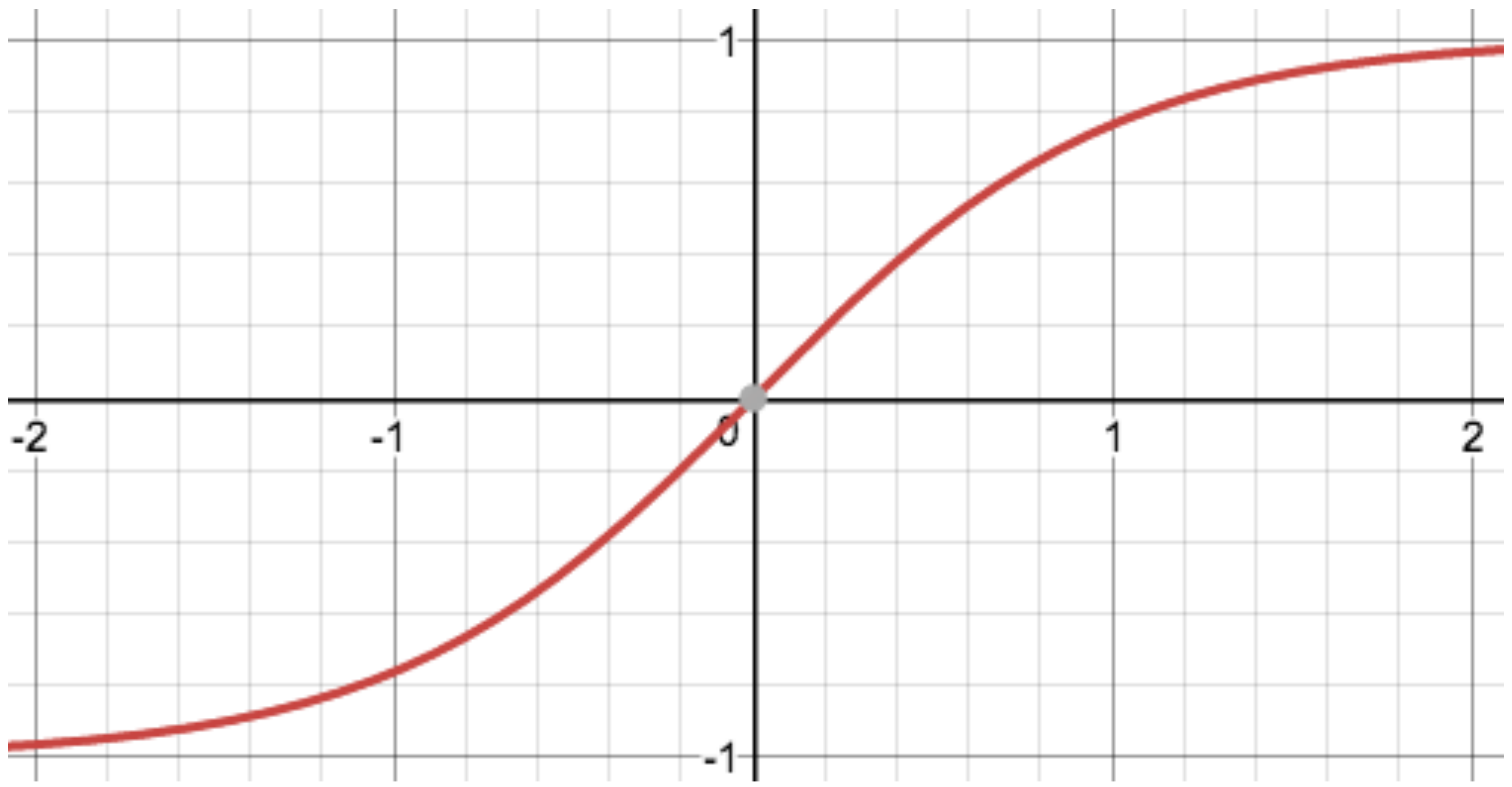
$$y = a_1 x_1 + a_2 x_2$$ X

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

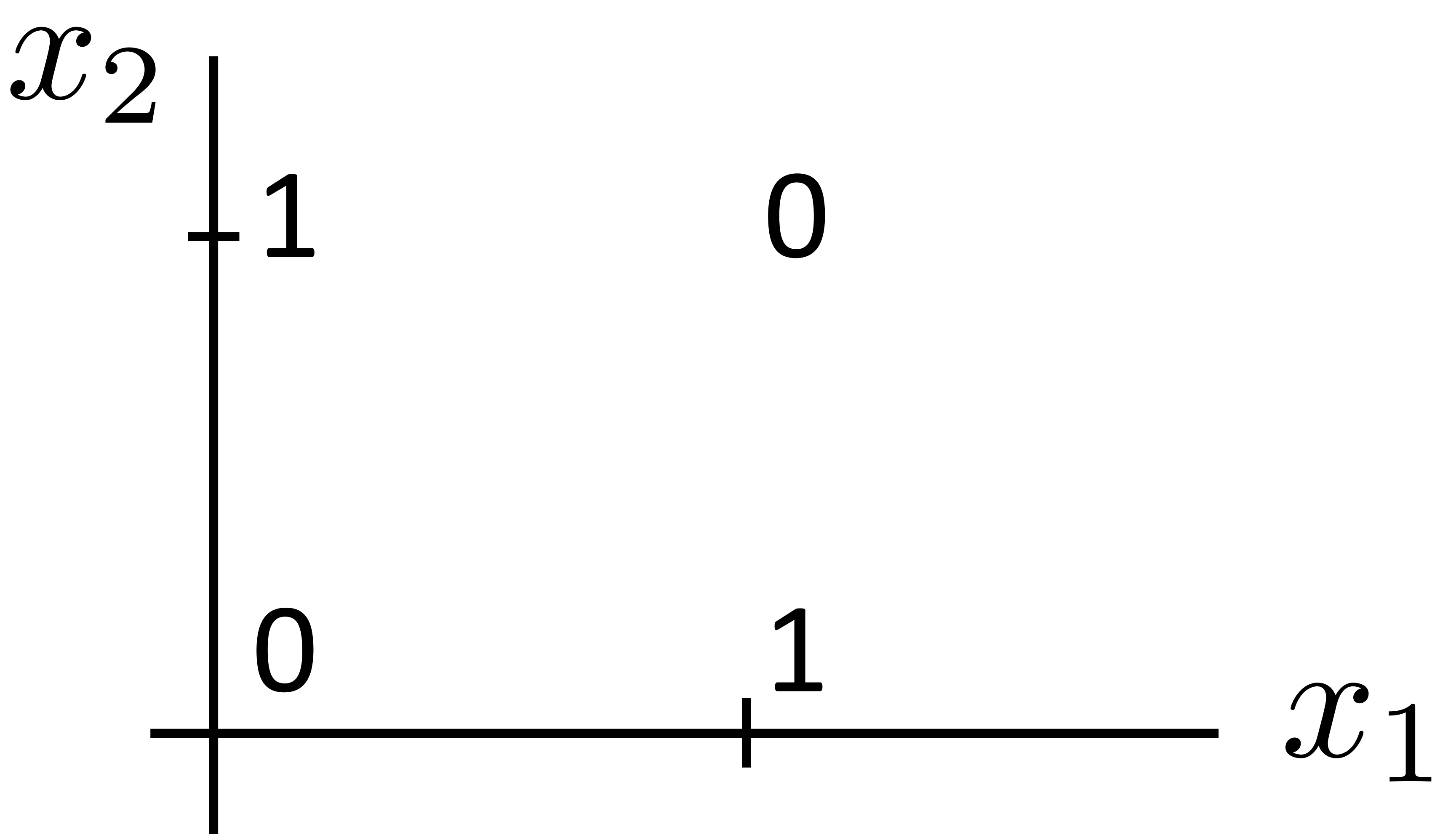
"or"

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

(looks like action potential in neuron)



Neural Networks: XOR



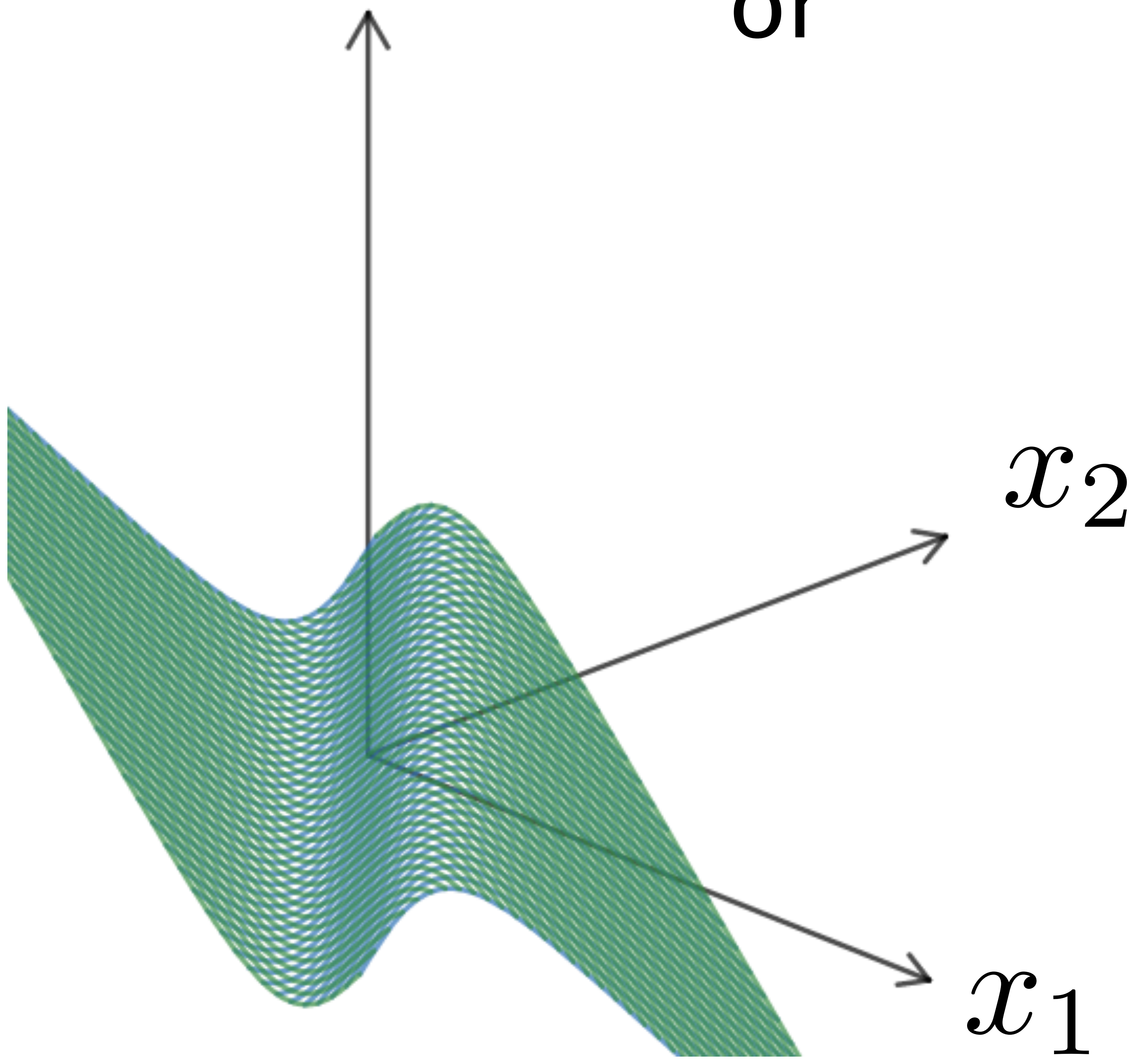
x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

$$y = a_1 x_1 + a_2 x_2$$ X

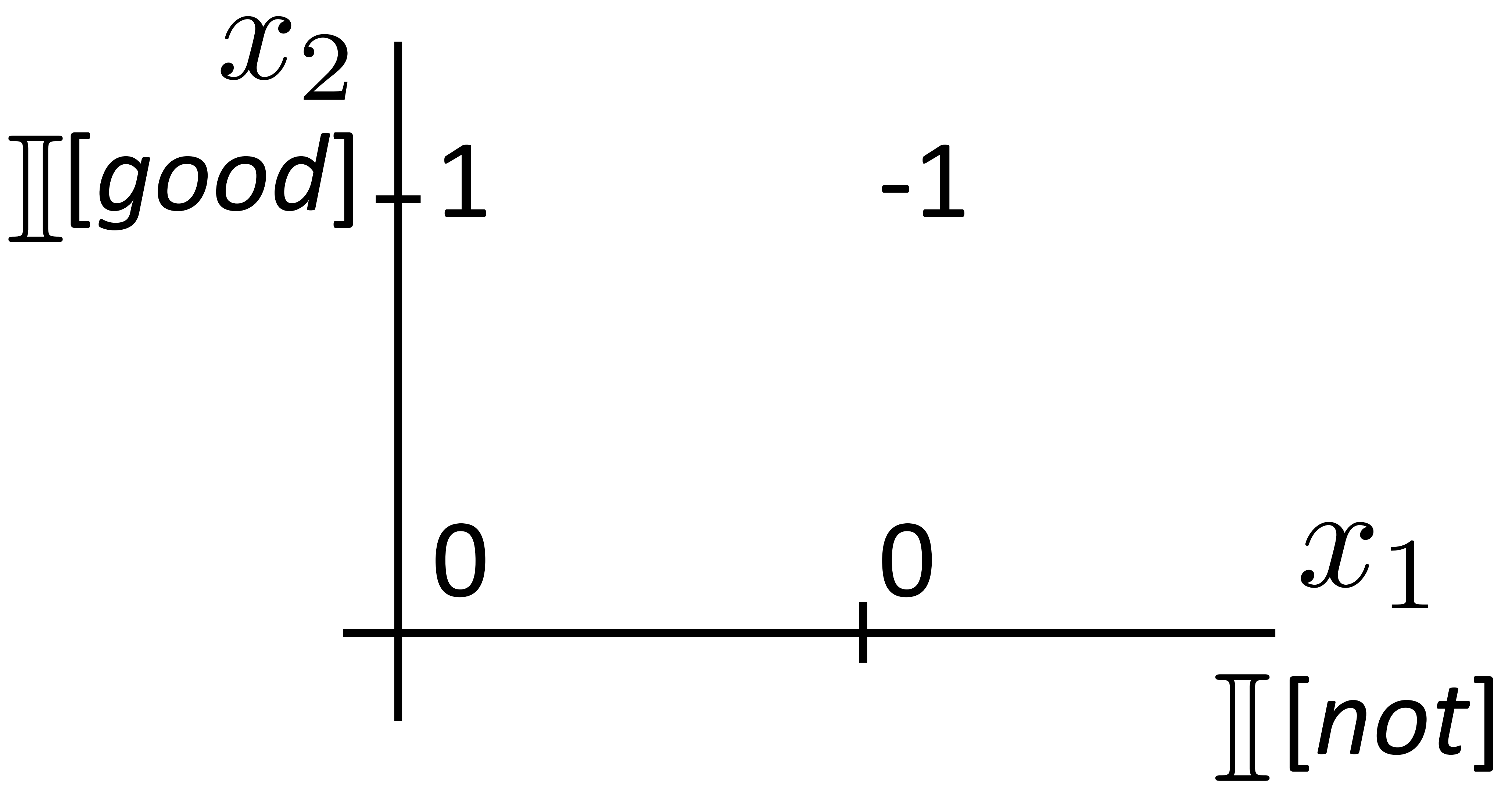
$$y = a_1 x_1 + a_2 x_2 + a_3 \tanh(x_1 + x_2)$$ ✓

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

“or”

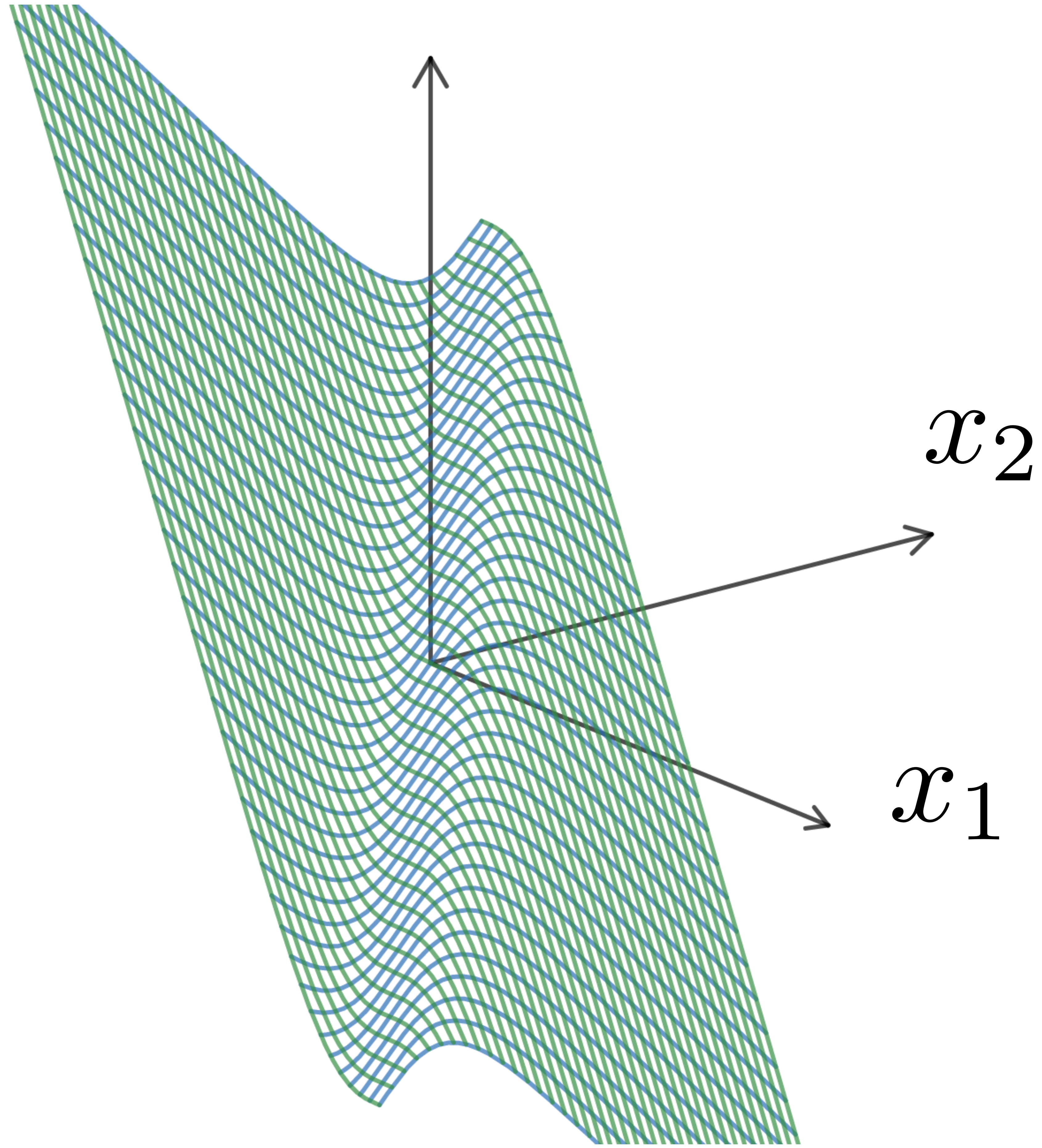


Neural Networks: XOR



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

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



Neural Networks

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

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

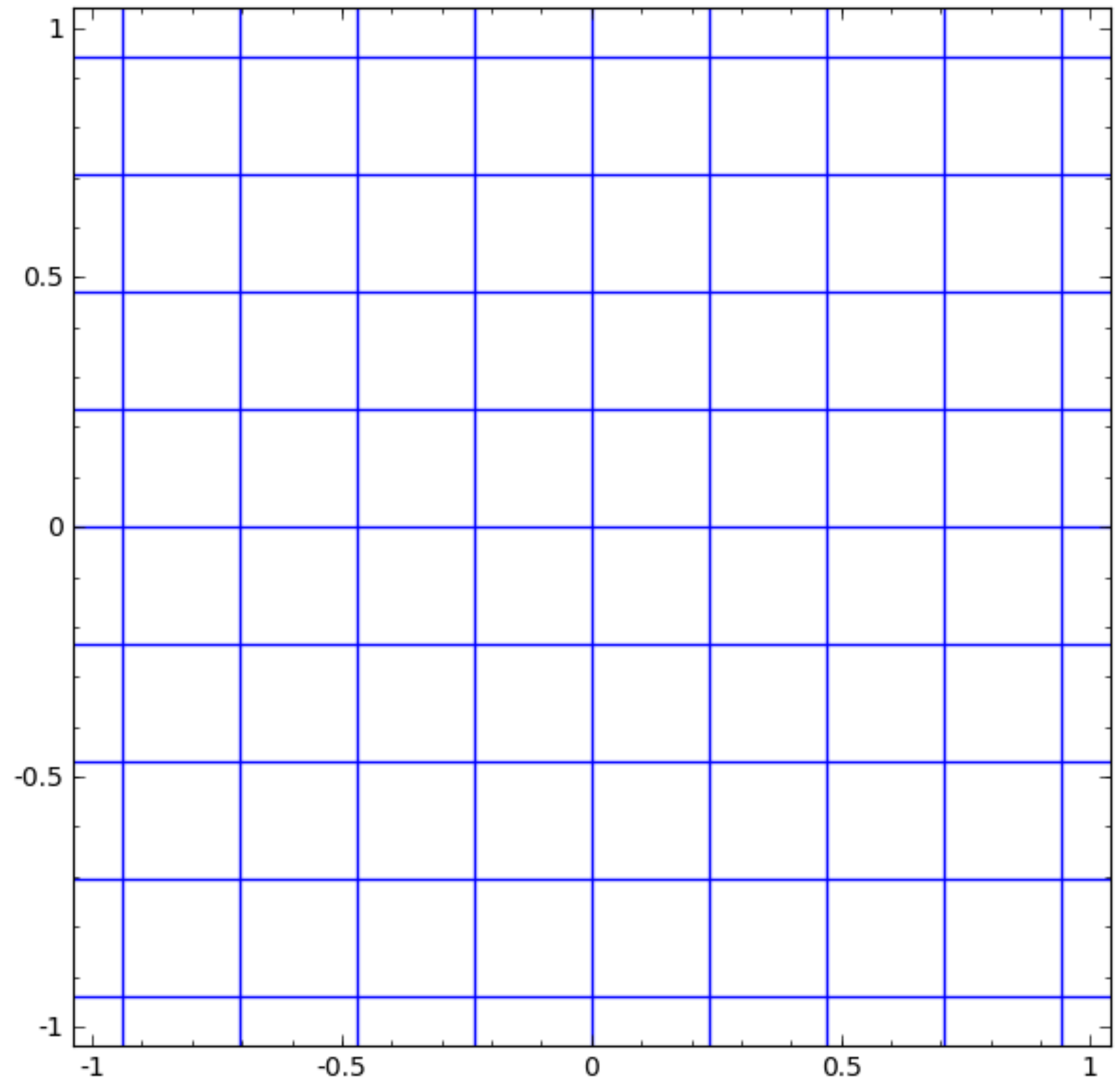
$$\mathbf{y} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Nonlinear transformation

Warp space

Shift

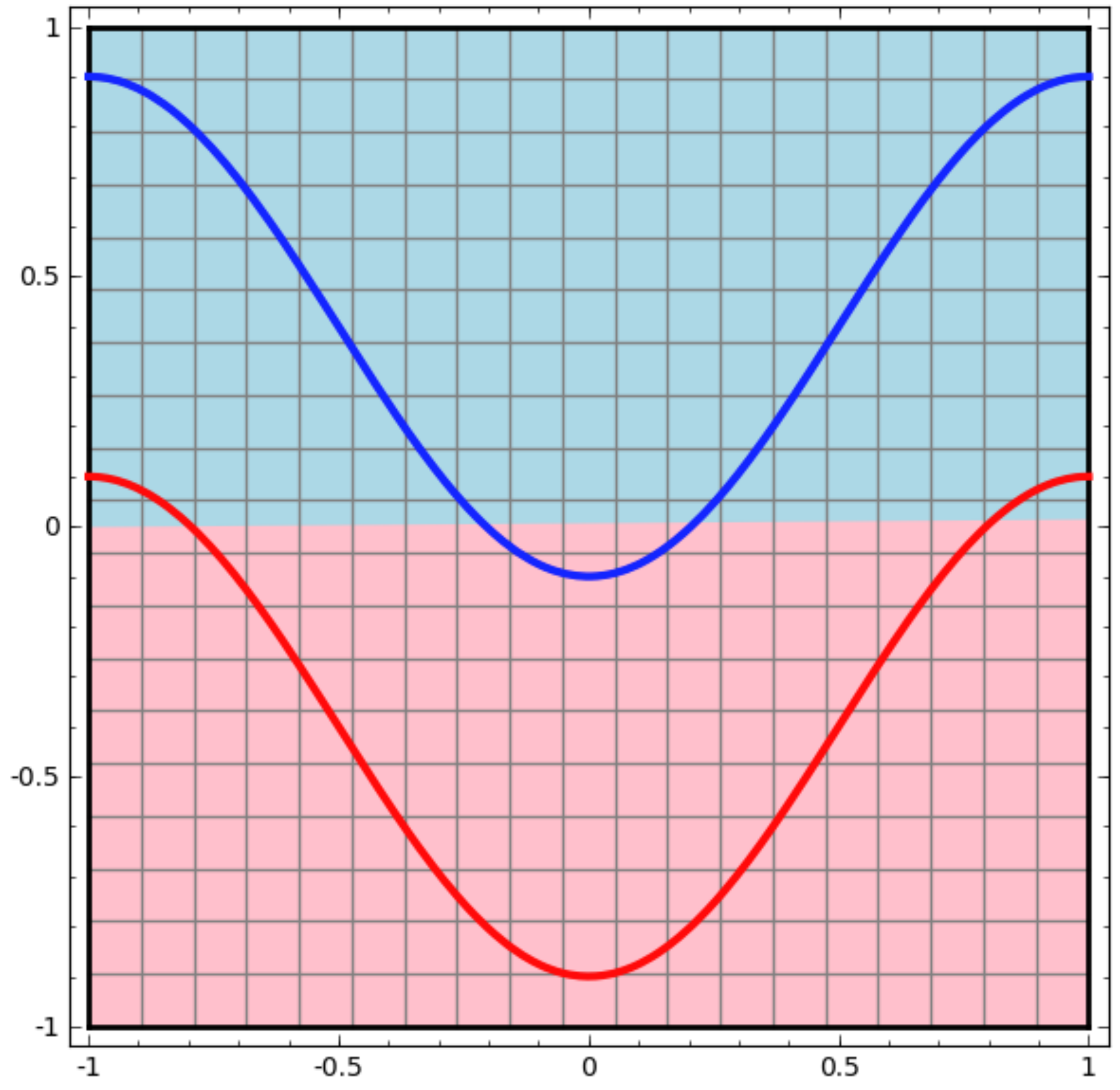
tanh



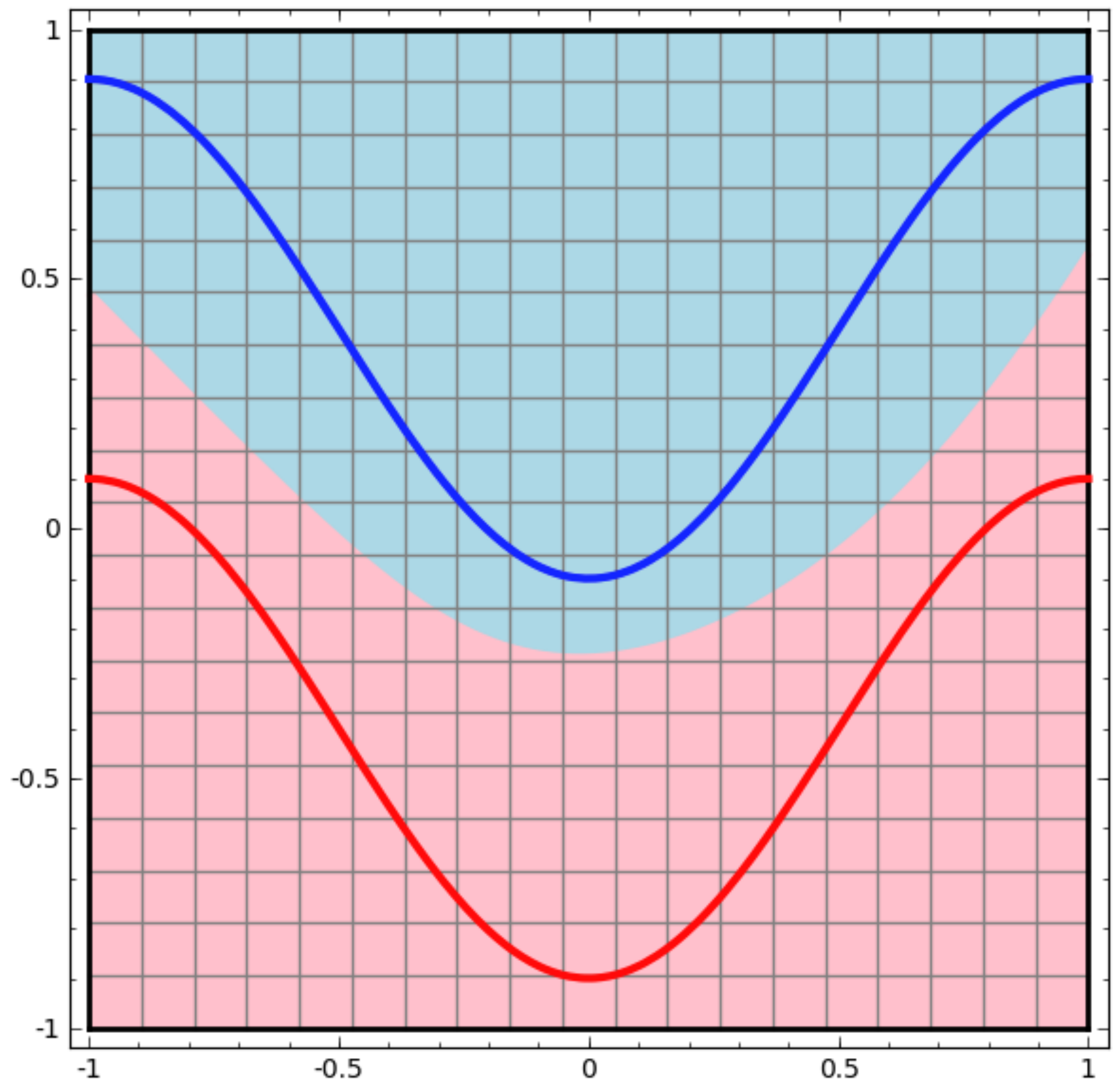
Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Neural Networks

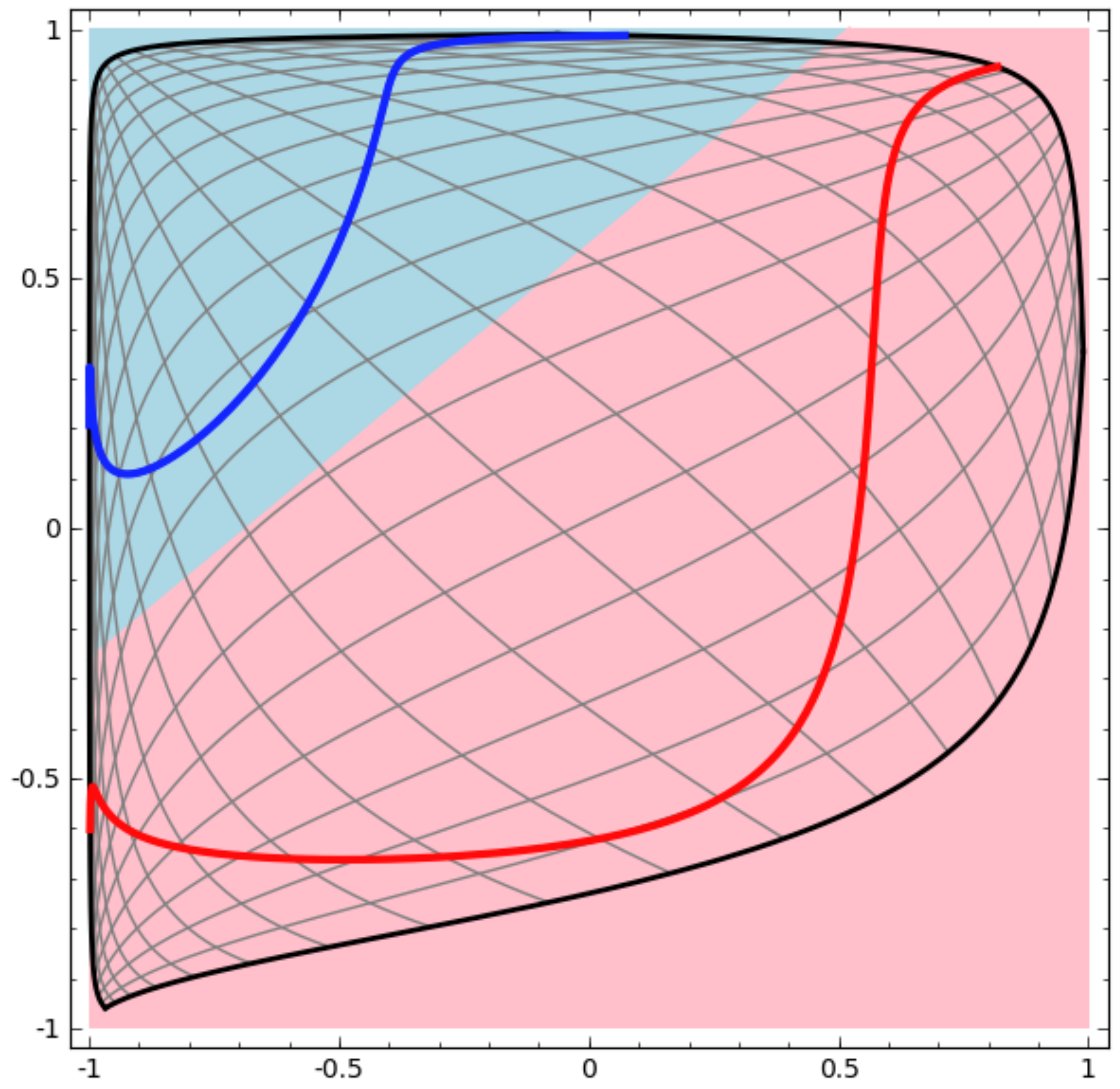
Linear classifier



Neural network

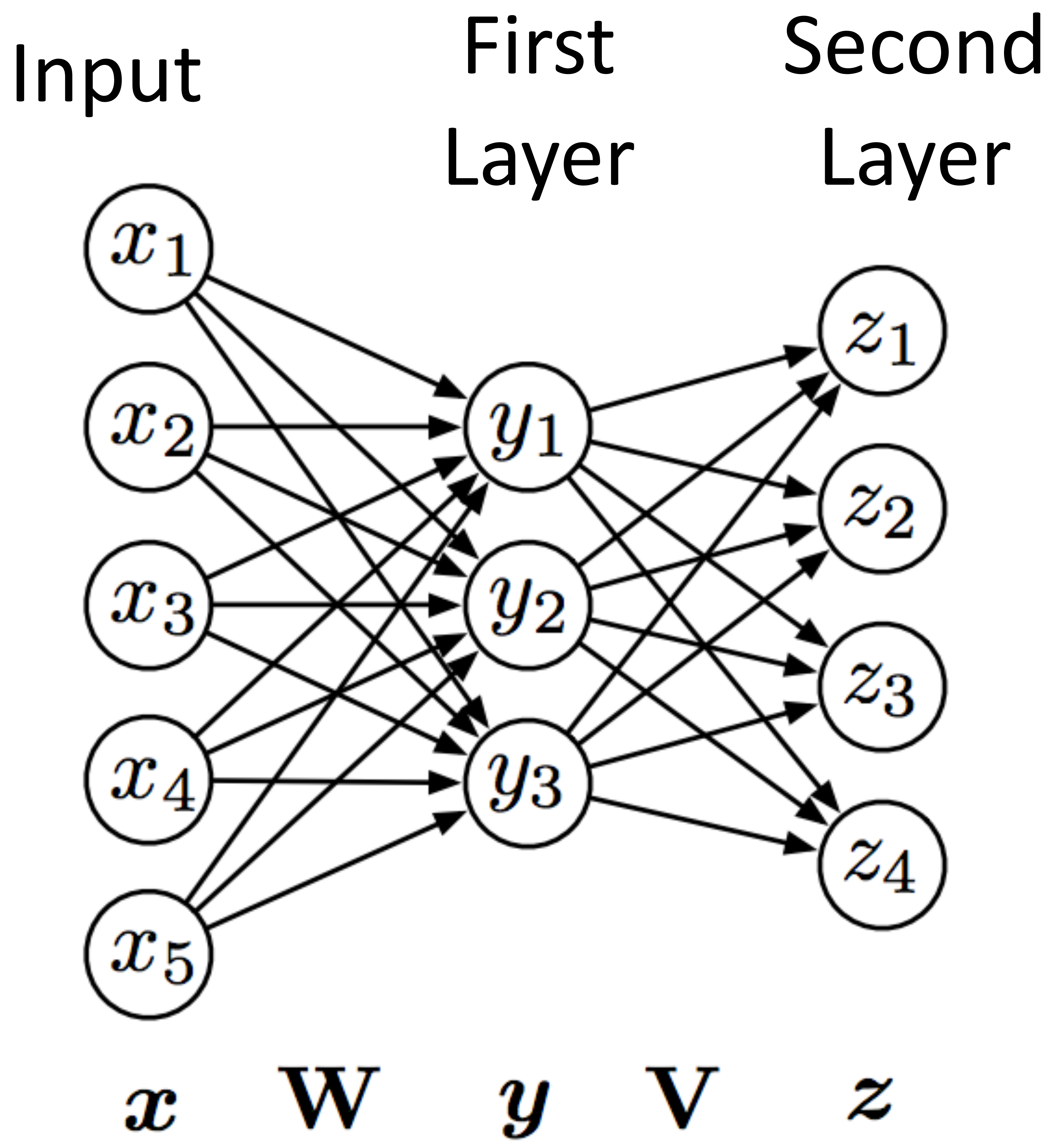


…possible because we transformed the space!



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

Deep Neural Networks



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

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

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

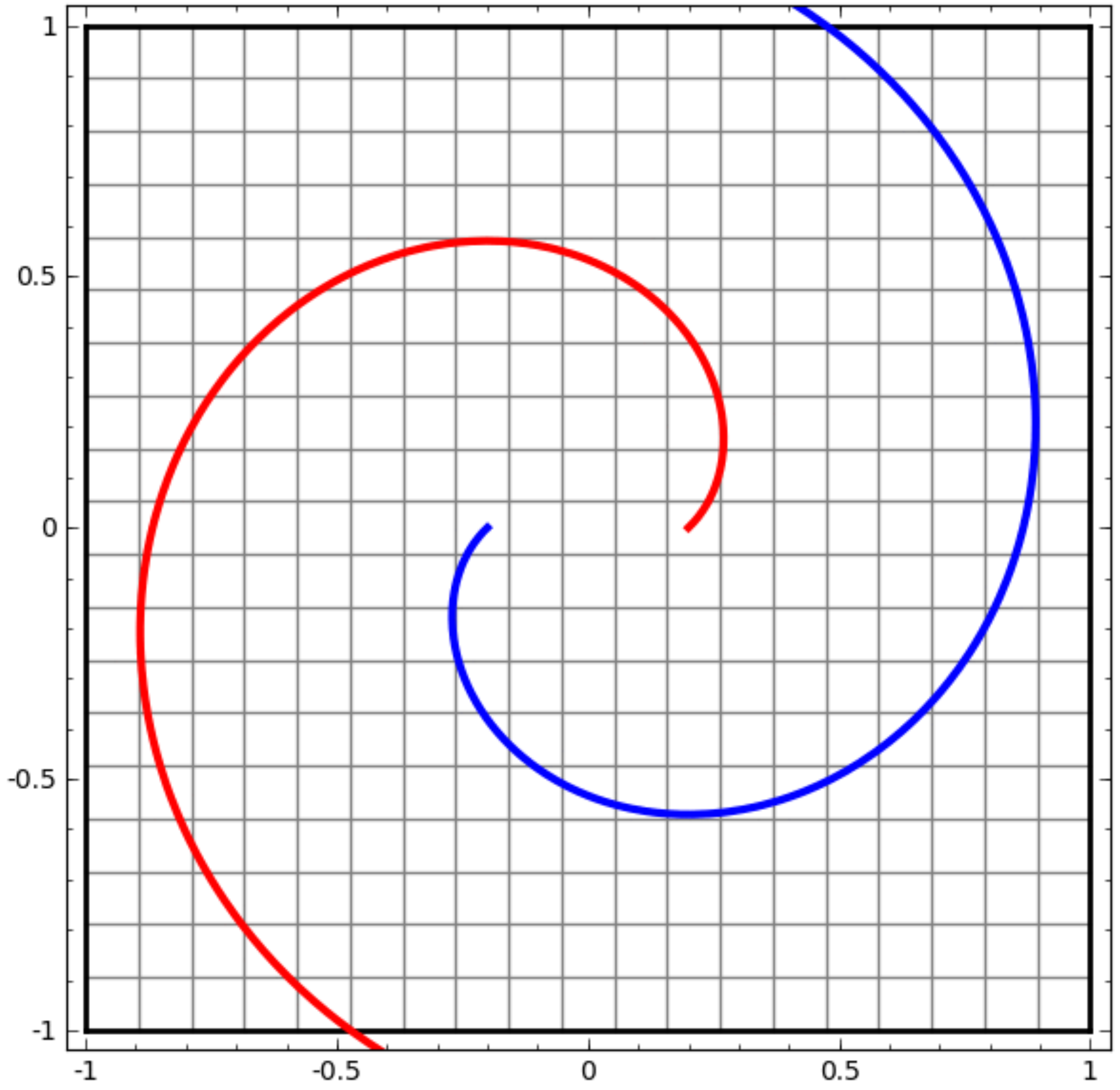
"Feedforward" computation (not recurrent)

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

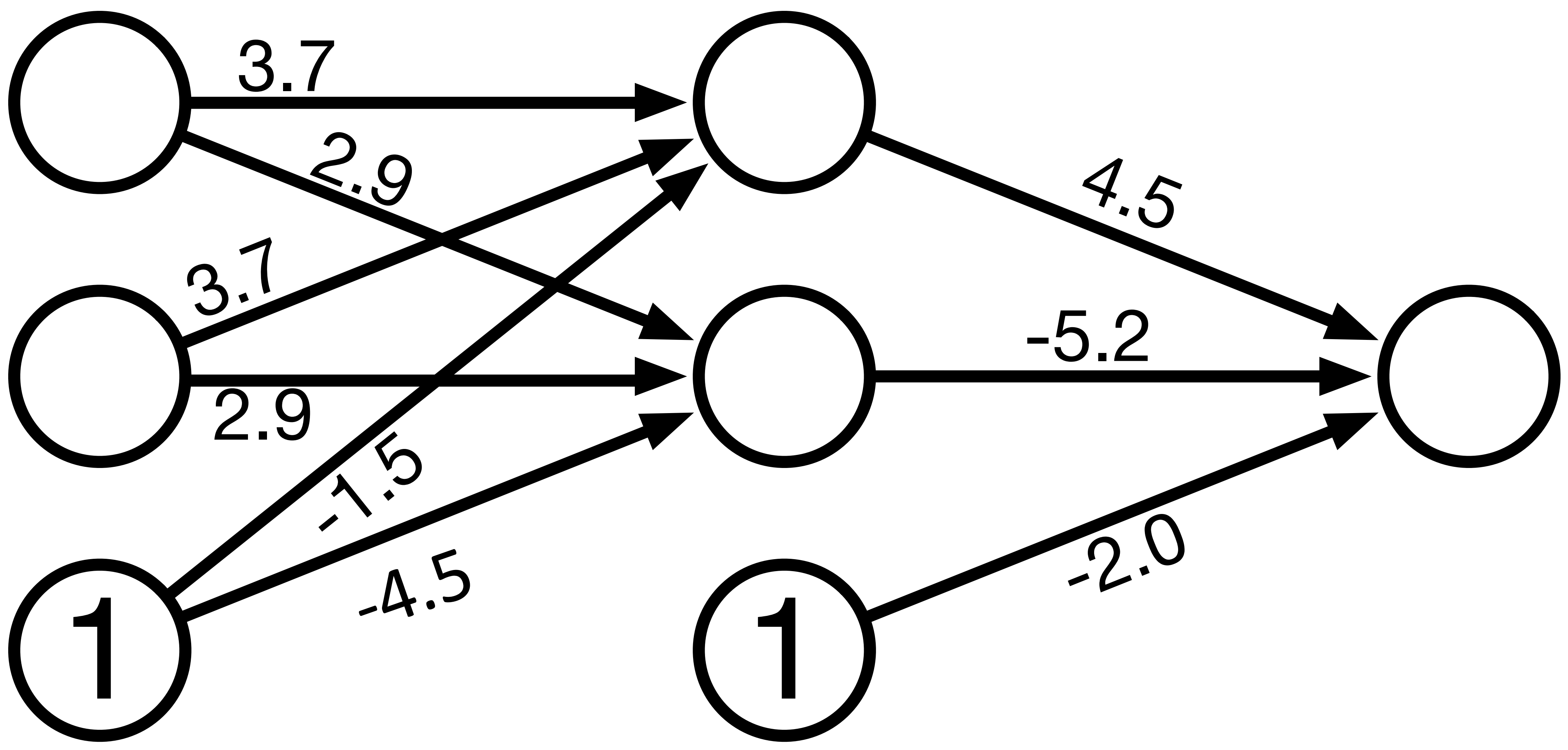
Deep Neural Networks



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

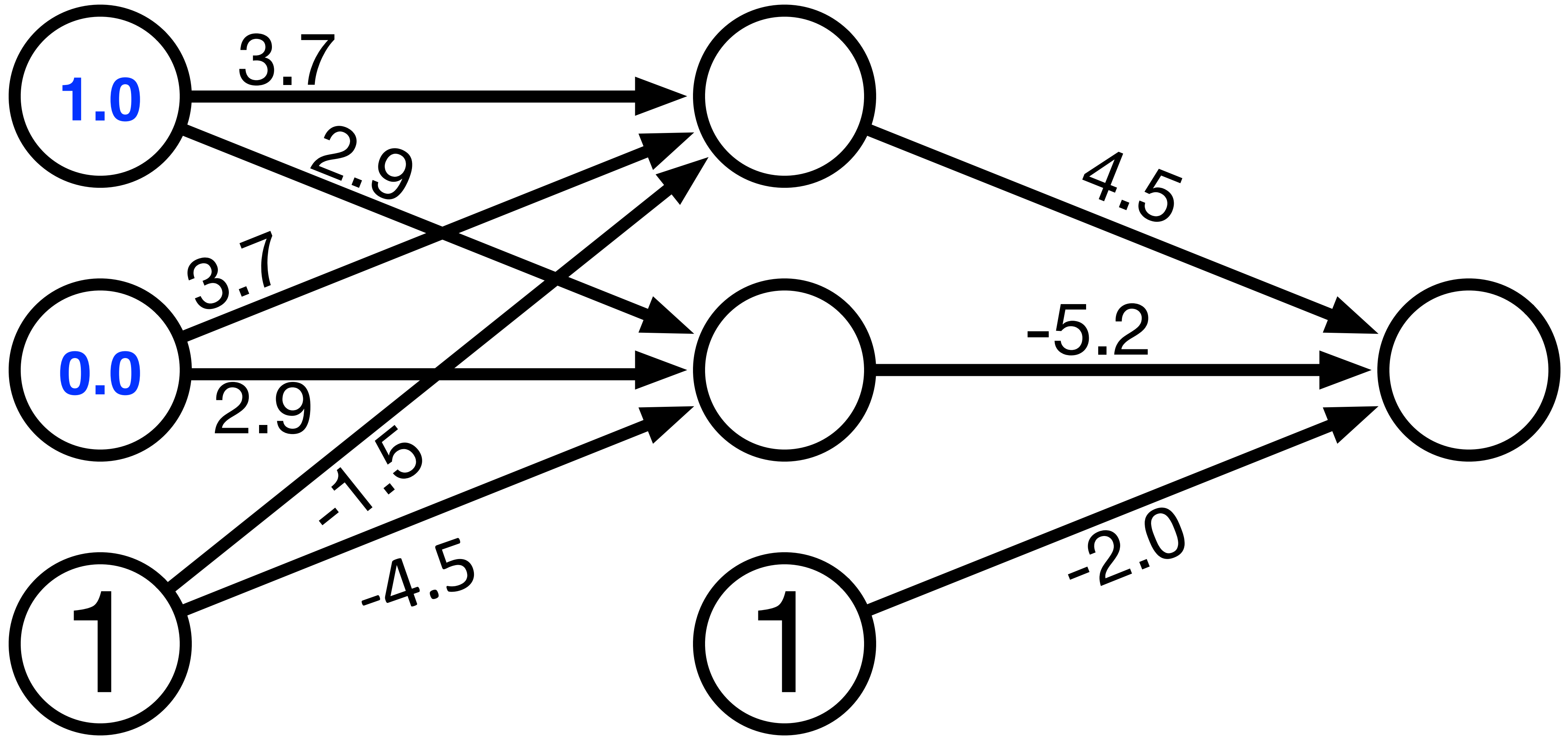
Feedforward Networks, Backpropagation

Simple Neural Network



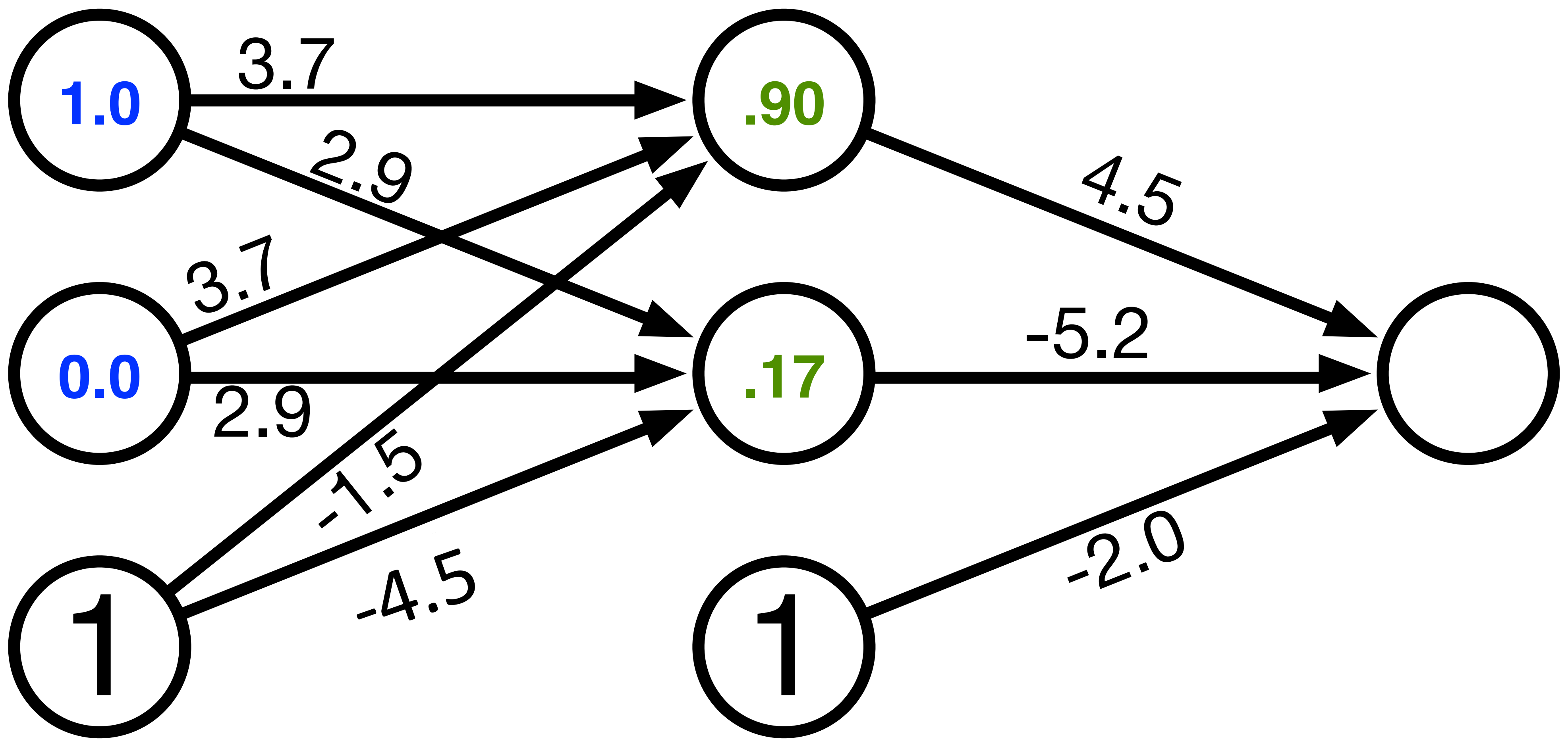
https://inst.eecs.berkeley.edu/~cs182/sp06/notes/backprop.pdf

Simple Neural Network



- Try out two input values

Simple Neural Network

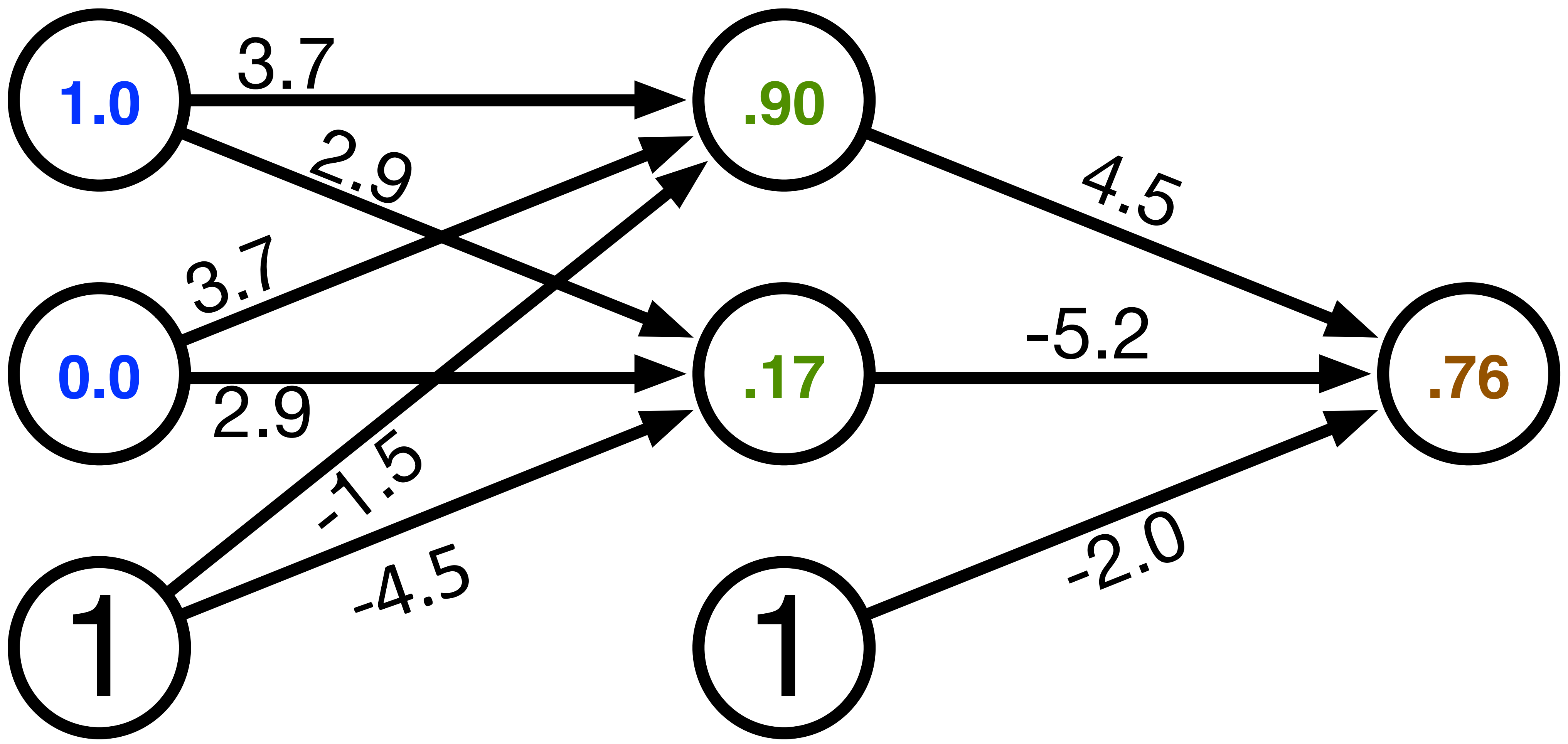


- Try out two input values
- Hidden unit computation

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

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

Simple Neural Network



- Output unit computation

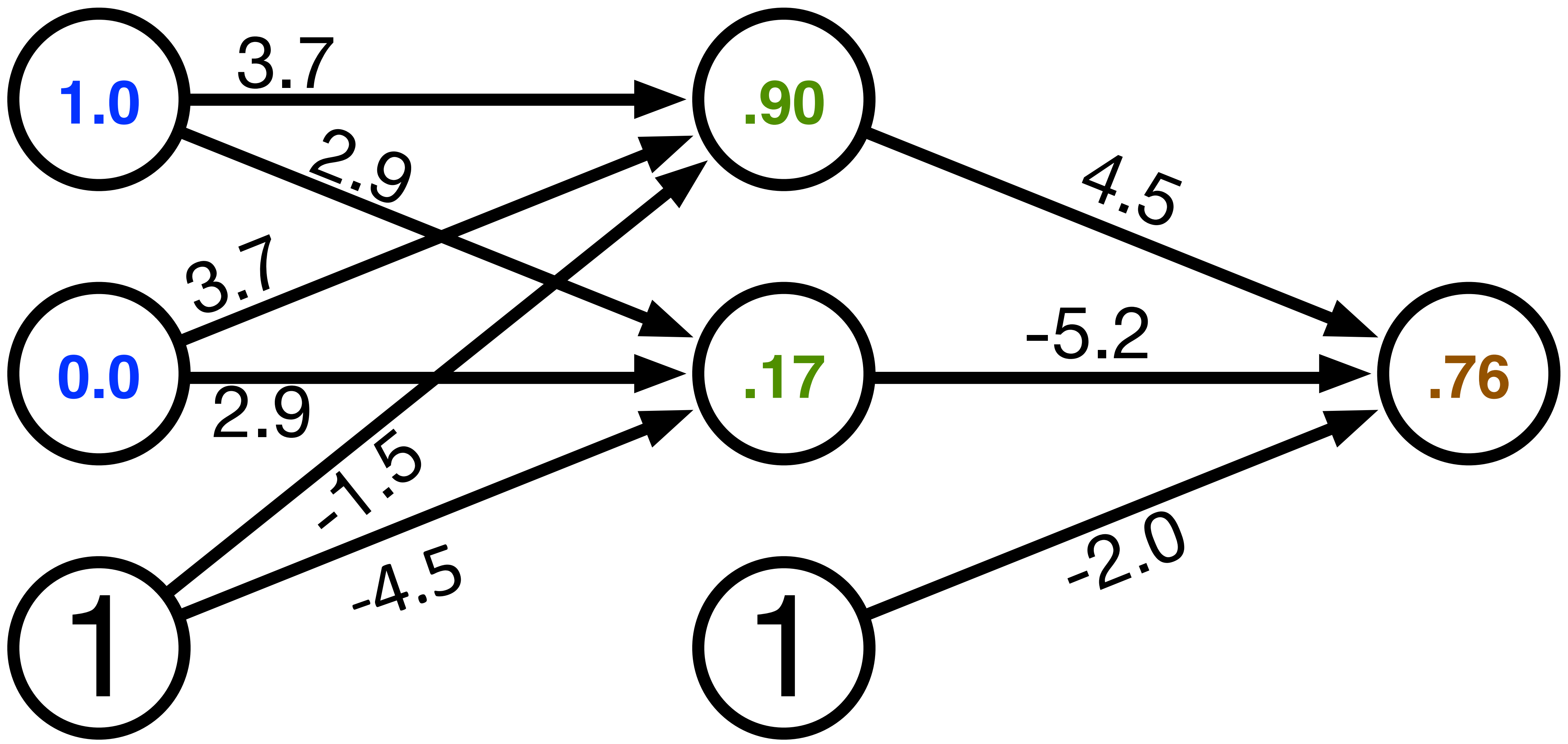
$$\text{sigmoid}(.90 \times 4.5 + .17 \times -5.2 + 1 \times -2.0) = \text{sigmoid}(1.17) = \frac{1}{1 + e^{-1.17}} = 0.76$$

Simple Neural Network

Input x_0	Input x_1	Hidden h_0	Hidden h_1	Output y_0
0	0	0.12	0.02	$0.18 \rightarrow 0$
0	1	0.88	0.27	$0.74 \rightarrow 1$
1	0	0.73	0.12	$0.74 \rightarrow 1$
1	1	0.99	0.73	$0.33 \rightarrow 0$

- Network implements XOR
- h_0 is OR, h_1 is AND

Error



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

- Q: how do we adjust the weights?

Derivative of Sigmoid

- Sigmoid function:

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

- Derivative:

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

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 :

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

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 :

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

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 $\text{sigmoid}(s)$:

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

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 :

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

Putting it All Together

- Derivative of error with regard to one weight w_k :

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

$$= \underbrace{-(t-y)}_{\text{error}} \quad \underbrace{y(1-y)}_{\text{derivative of sigmoid: } y'} \quad h_k$$

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

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

Multiple Output Nodes

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

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

- But, typically 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:

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

Hidden Layer Update

- 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:

$$\delta_j = (t_j - y_j)\, y_j'$$

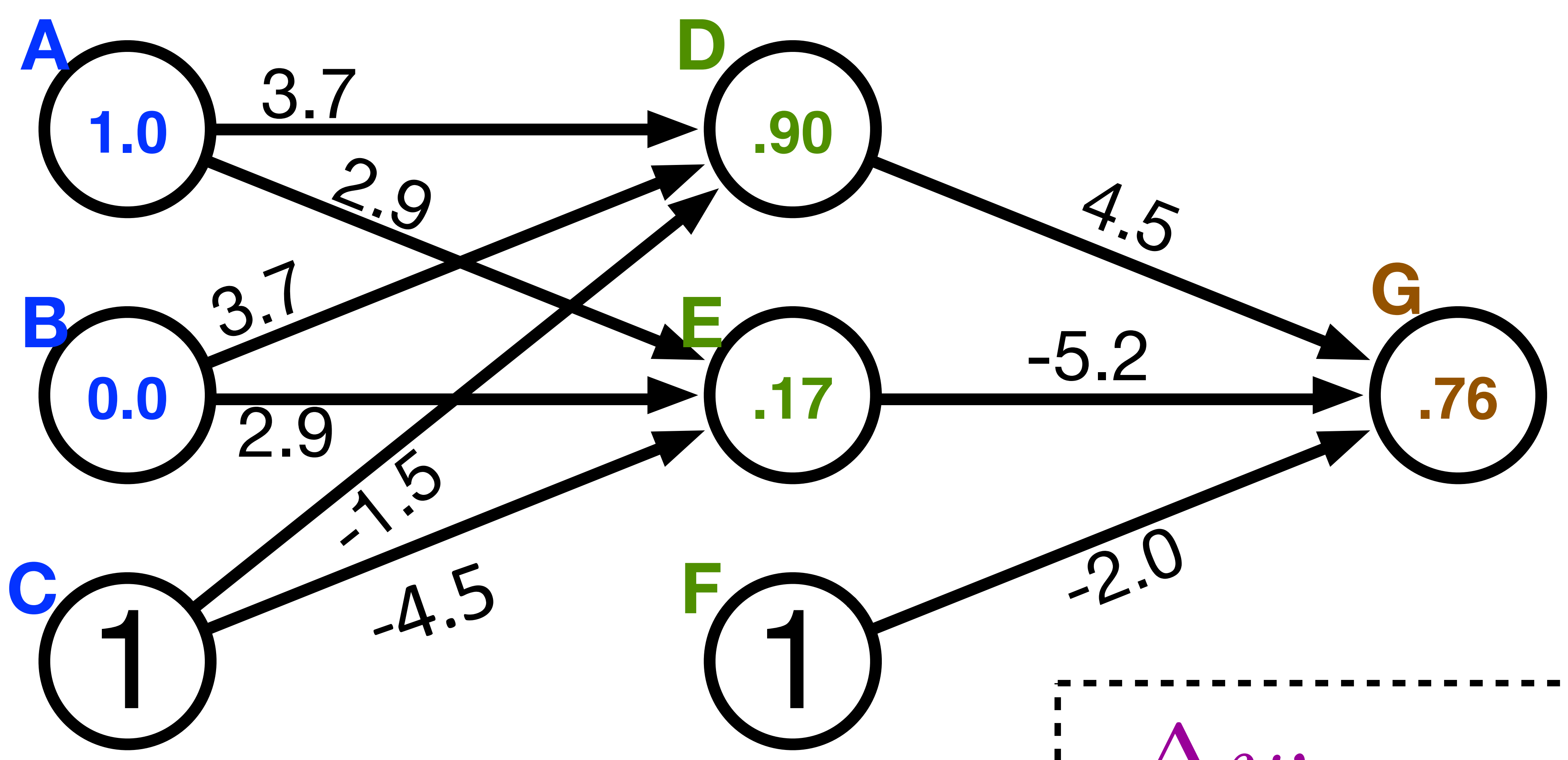
- Back-propagate the error term:

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

- Universal update formula:

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

Our Example



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

learning rate

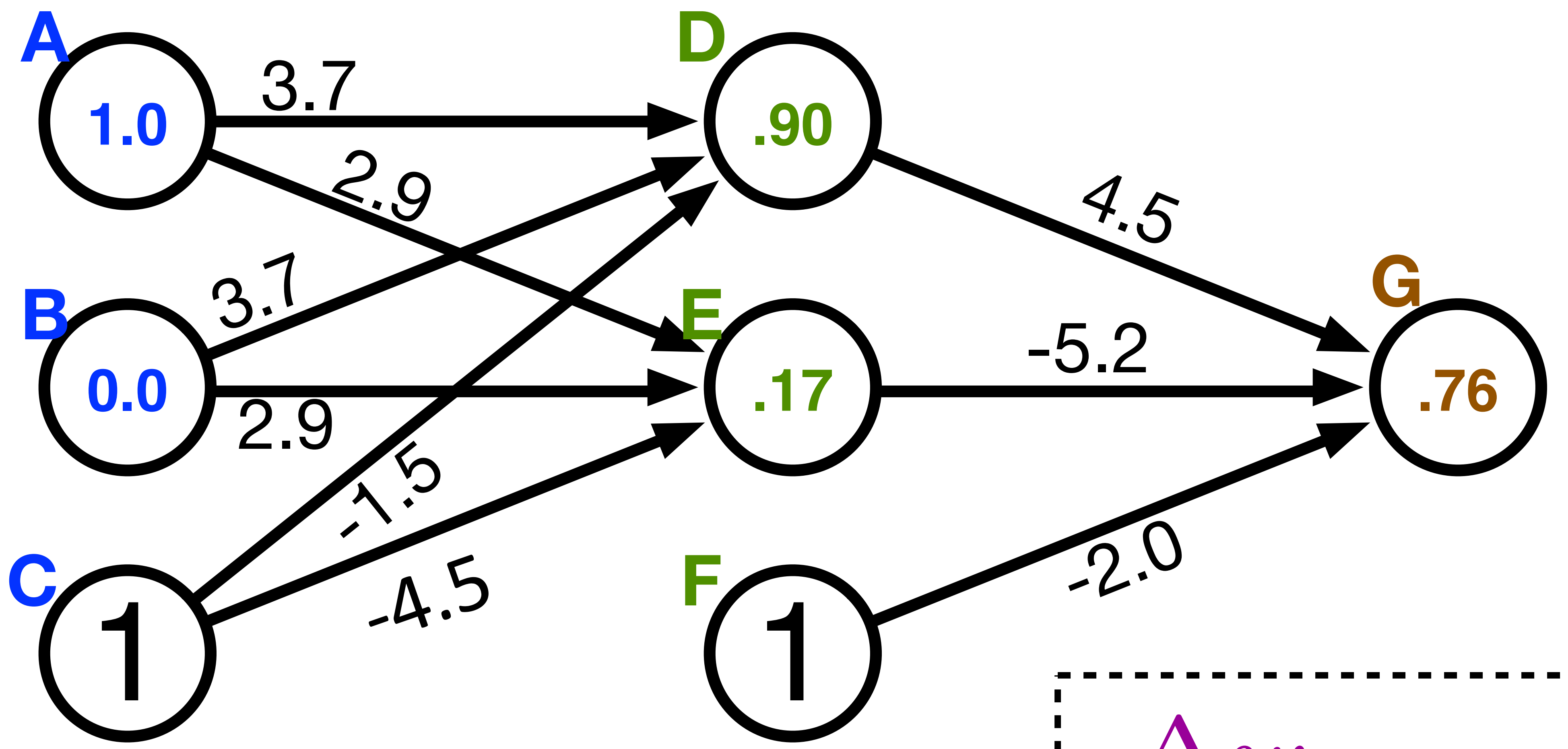
error term

hidden node value

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

- Q: how do we adjust the weights?

Our Example

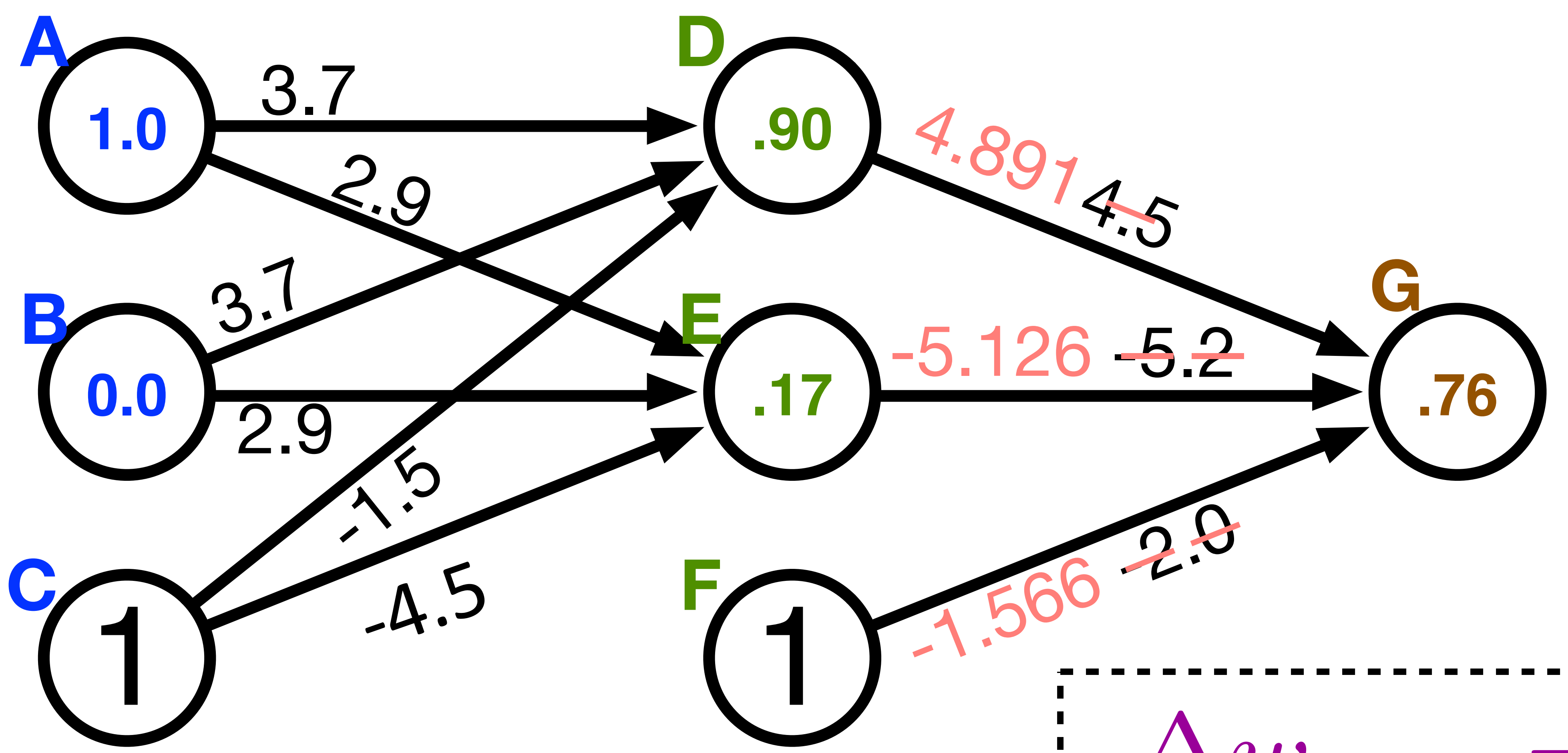


$$\Delta w_{j\leftarrow k} = \underbrace{\mu}_{\text{learning rate}}\underbrace{(t_j - y_j)\, y'_j}_{\text{error term}}\; \underbrace{h_k}_{\text{hidden node value}}$$

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

$$\delta_{\mathsf{G}} = (t-y)\, y' = (1 - .76)\; 0.181 = .0434$$
$$\Delta w_{\mathsf{GD}} = \mu\; \delta_{\mathsf{G}}\; h_{\mathsf{D}} = 10 \times .0434 \times .90 = .391$$
$$\Delta w_{\mathsf{GE}} = \mu\; \delta_{\mathsf{G}}\; h_{\mathsf{E}} = 10 \times .0434 \times .17 = .074$$
$$\Delta w_{\mathsf{GF}} = \mu\; \delta_{\mathsf{G}}\; h_{\mathsf{F}} = 10 \times .0434 \times 1 = .434$$

Our Example



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

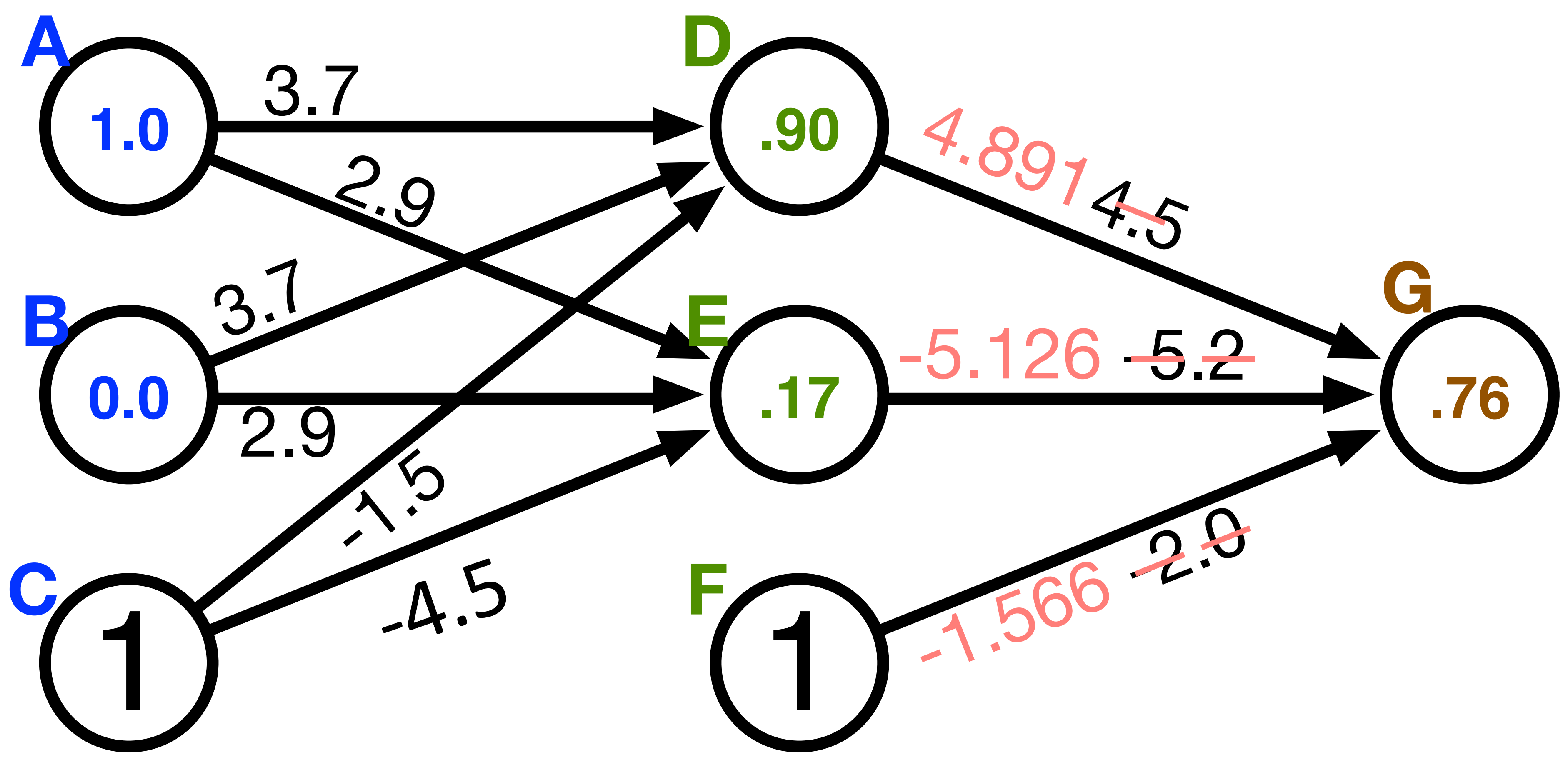
learning rate

hidden node value

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

$$\delta_\text{G} = (t - y)\, y' = (1 - .76)\; 0.181 = .0434$$
$$\Delta w_\text{GD} = \mu\; \delta_\text{G}\; h_\text{D} = 10 \times .0434 \times .90 = .391$$
$$\Delta w_\text{GE} = \mu\; \delta_\text{G}\; h_\text{E} = 10 \times .0434 \times .17 = .074$$
$$\Delta w_\text{GF} = \mu\; \delta_\text{G}\; h_\text{F} = 10 \times .0434 \times 1 = .434$$

Hidden Layer Updates



- Hidden node **D**:

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

- Hidden node **E**:

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

etc.

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

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

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

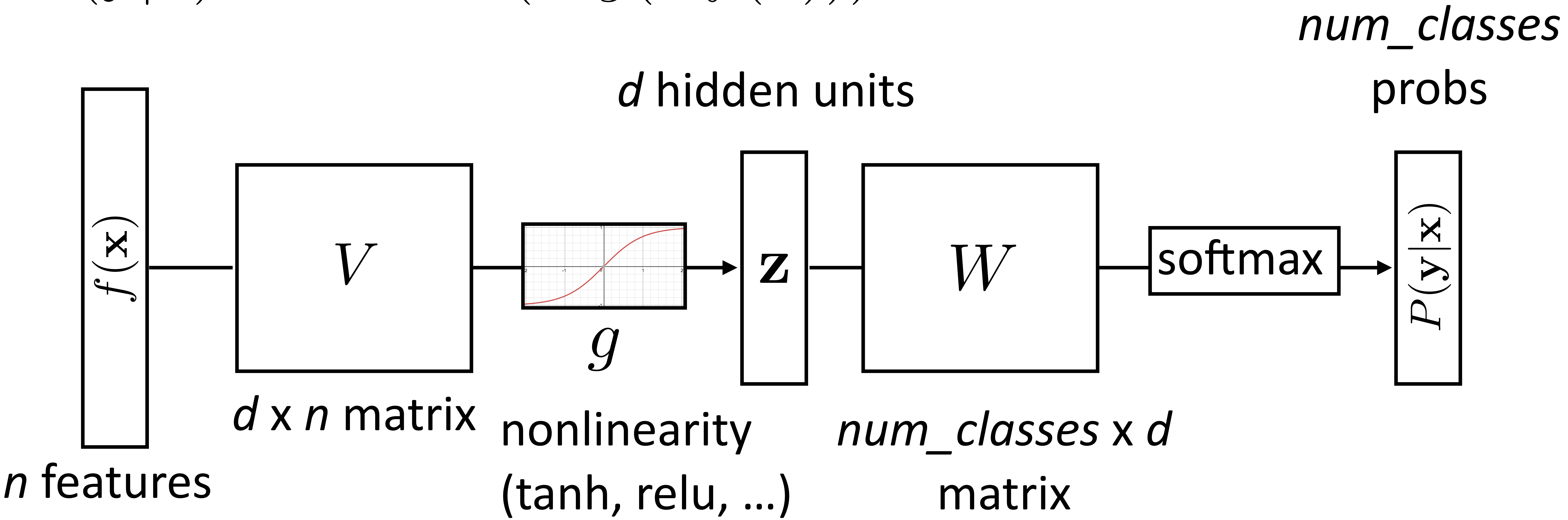
- Weight vector per class;
  W is [num classes x num feats]

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

- 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

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

- Maximize log likelihood of training data

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

- *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**

num_classes x *d* matrix

W *j*

i

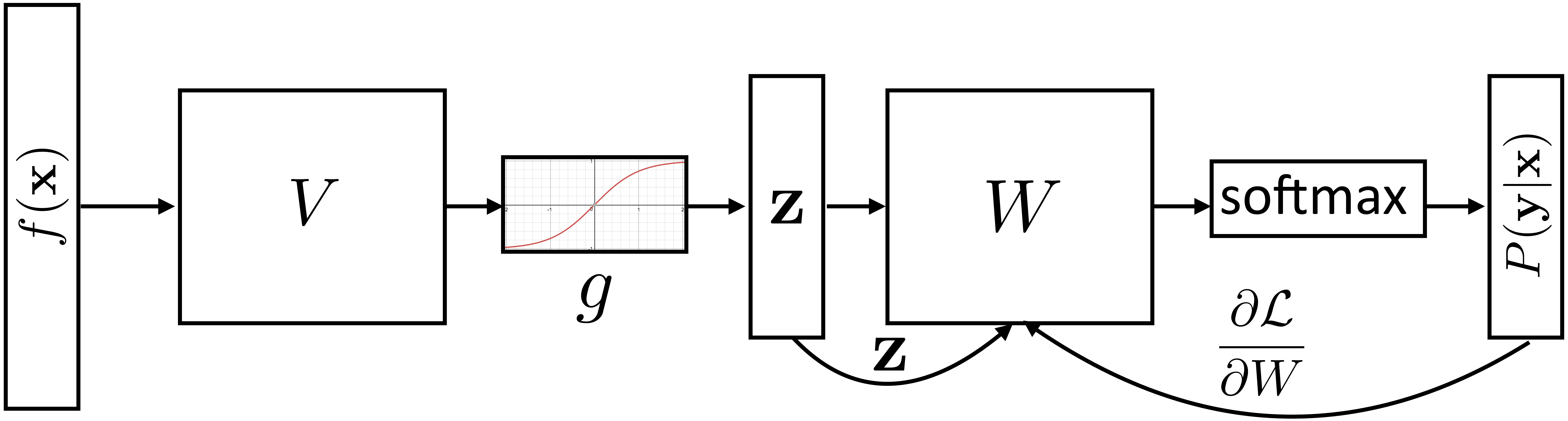
$\mathbf{z}_j - P(y = i|\mathbf{x})\mathbf{z}_j$

$-P(y = i|\mathbf{x})\mathbf{z}_j$

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

Neural Networks for Classification

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



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

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 $\mathbf{z}$ [some math...]

$$\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} = W_{i^*} - \sum_j P(y = j|\mathbf{x}) W_j$$

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

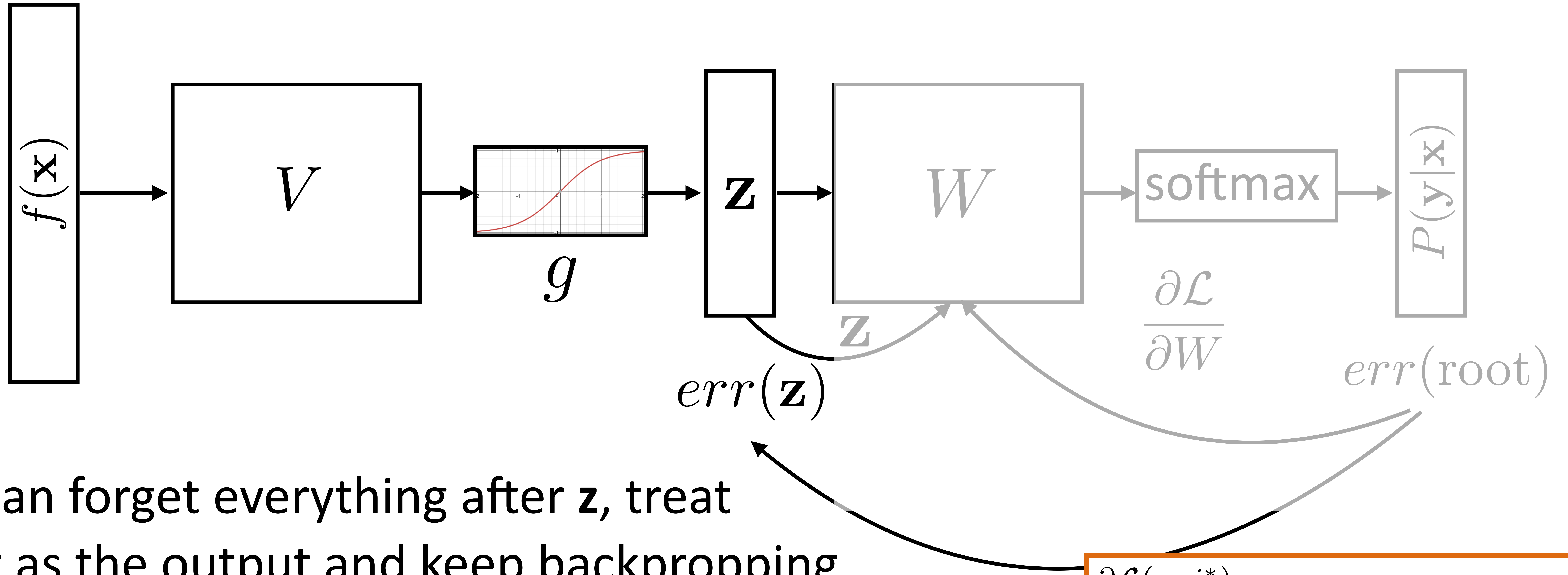
dim = d

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

dim = num_classes

Backpropagation: Picture

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



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

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

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

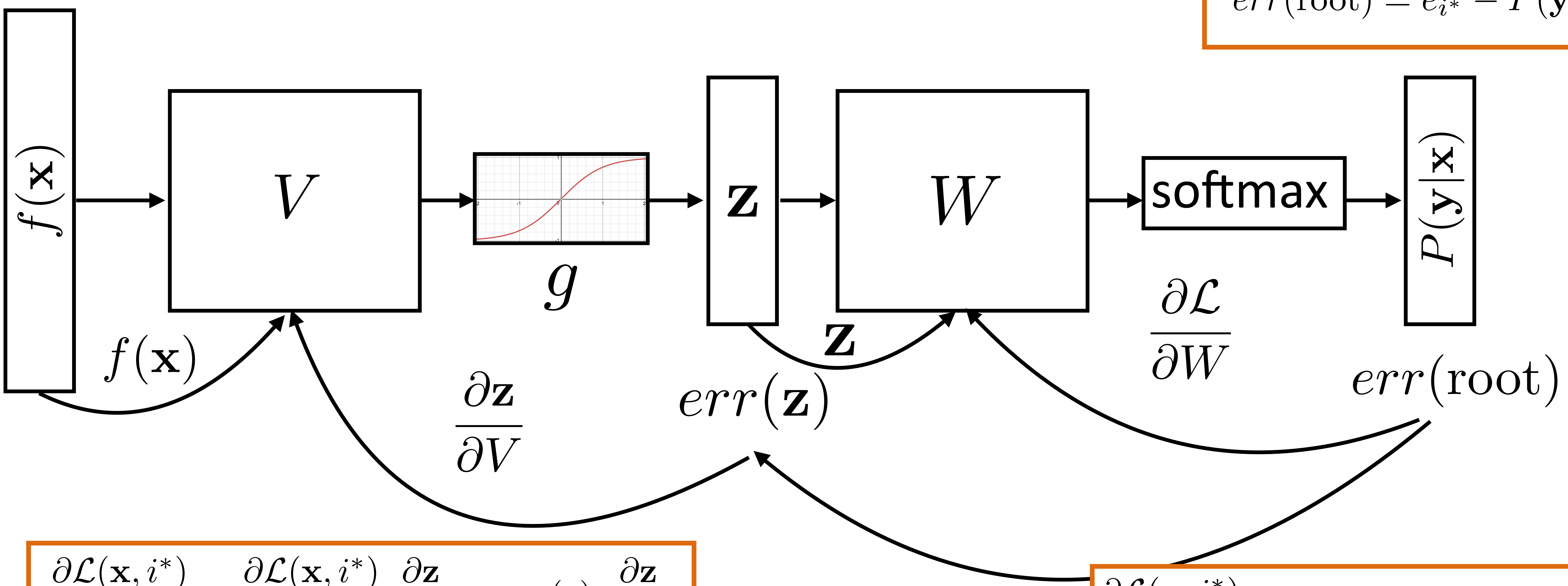
$$\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial V_{ij}} = \frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} \boxed{\frac{\partial \mathbf{z}}{V_{ij}}} \qquad \frac{\partial \mathbf{z}}{V_{ij}} = \boxed{\frac{\partial g(\mathbf{a})}{\partial \mathbf{a}}} \boxed{\frac{\partial \mathbf{a}}{\partial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

- 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

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

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



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

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

Backpropagation

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

- Step 1: compute $err(\mathrm{root}) = e_{i^*} - P(\mathbf{y}|\mathbf{x})$ (vector)
- Step 2: compute derivatives of *W* using *err*(root) (matrix)
- Step 3: compute $\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} = err(\mathbf{z}) = W^\top err(\mathrm{root})$ (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 $\boldsymbol{z}$ as feature vector
- Can compute derivative of loss with respect to $\boldsymbol{z}$ 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

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

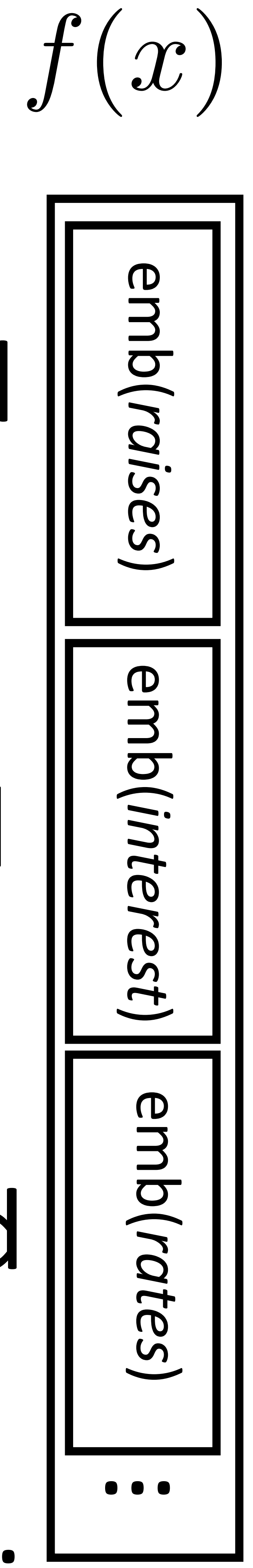
- Weight matrix learns position-dependent processing of the words

previous word

curr word

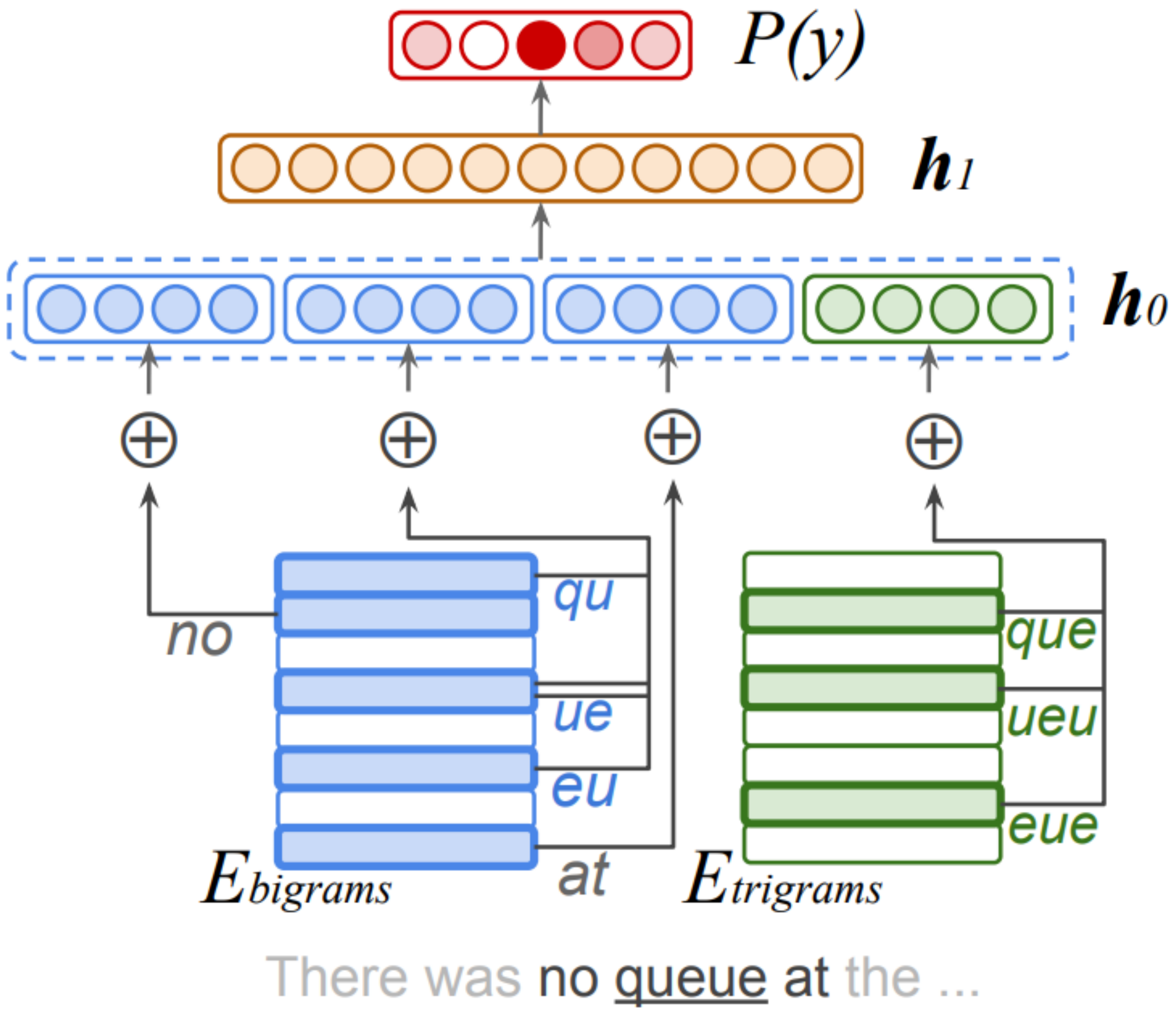
next word

other words, feats, etc.



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:

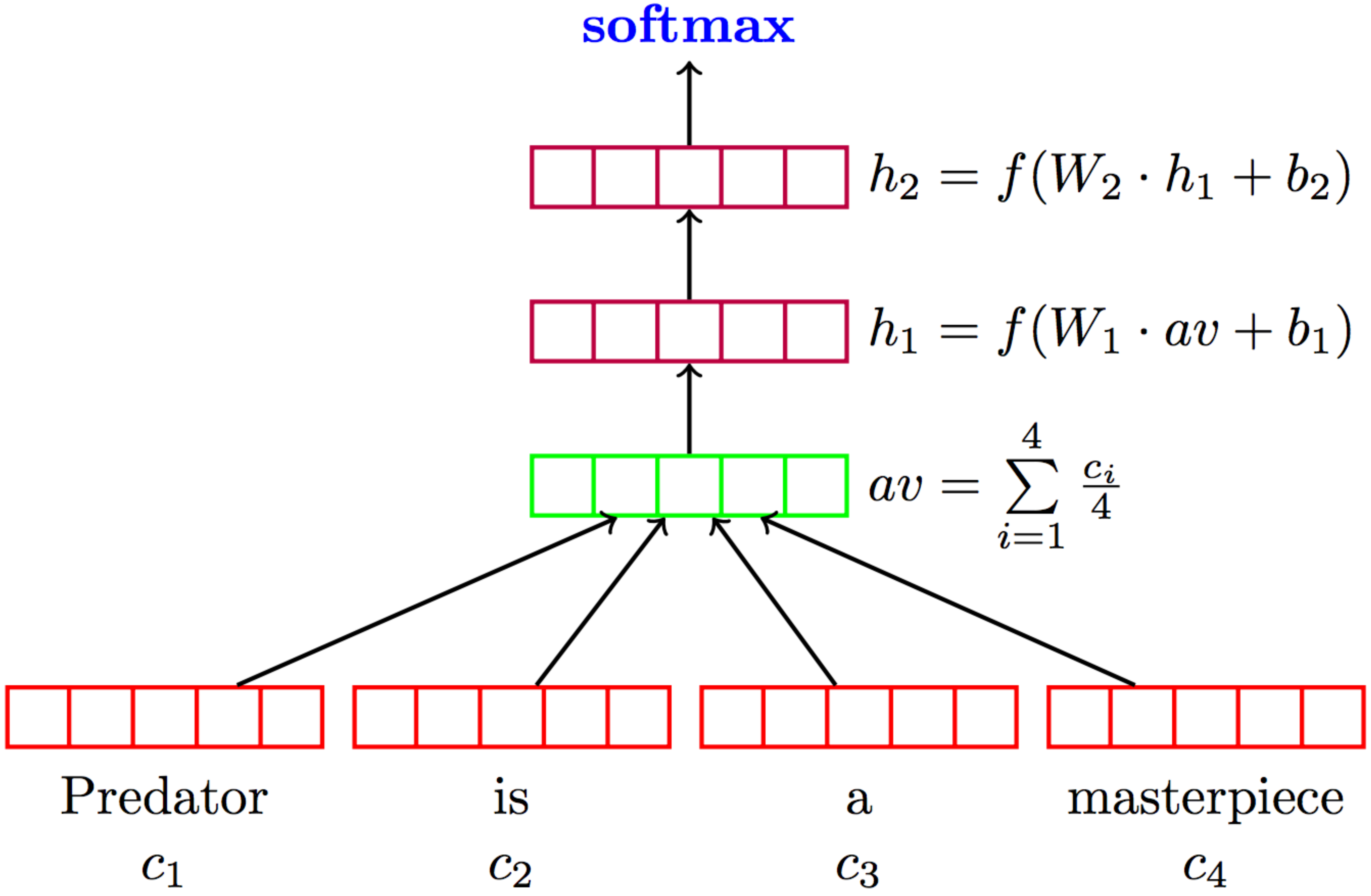
Model	Acc.	Wts.	MB	Ops.
Gillick et al. (2016)	95.06	900k	-	6.63m
Small FF	94.76	241k	0.6	0.27m
+Clusters	95.56	261k	1.0	0.31m
$\frac{1}{2}$ Dim.	95.39	143k	0.7	0.18m

- 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
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	—	—
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**	—
CNN-MC	**81.1**	47.4	**88.1**	—	2,452
WRRBM*	—	—	—	89.2	—

Iyyer et al. (2015)

Bag-of-words

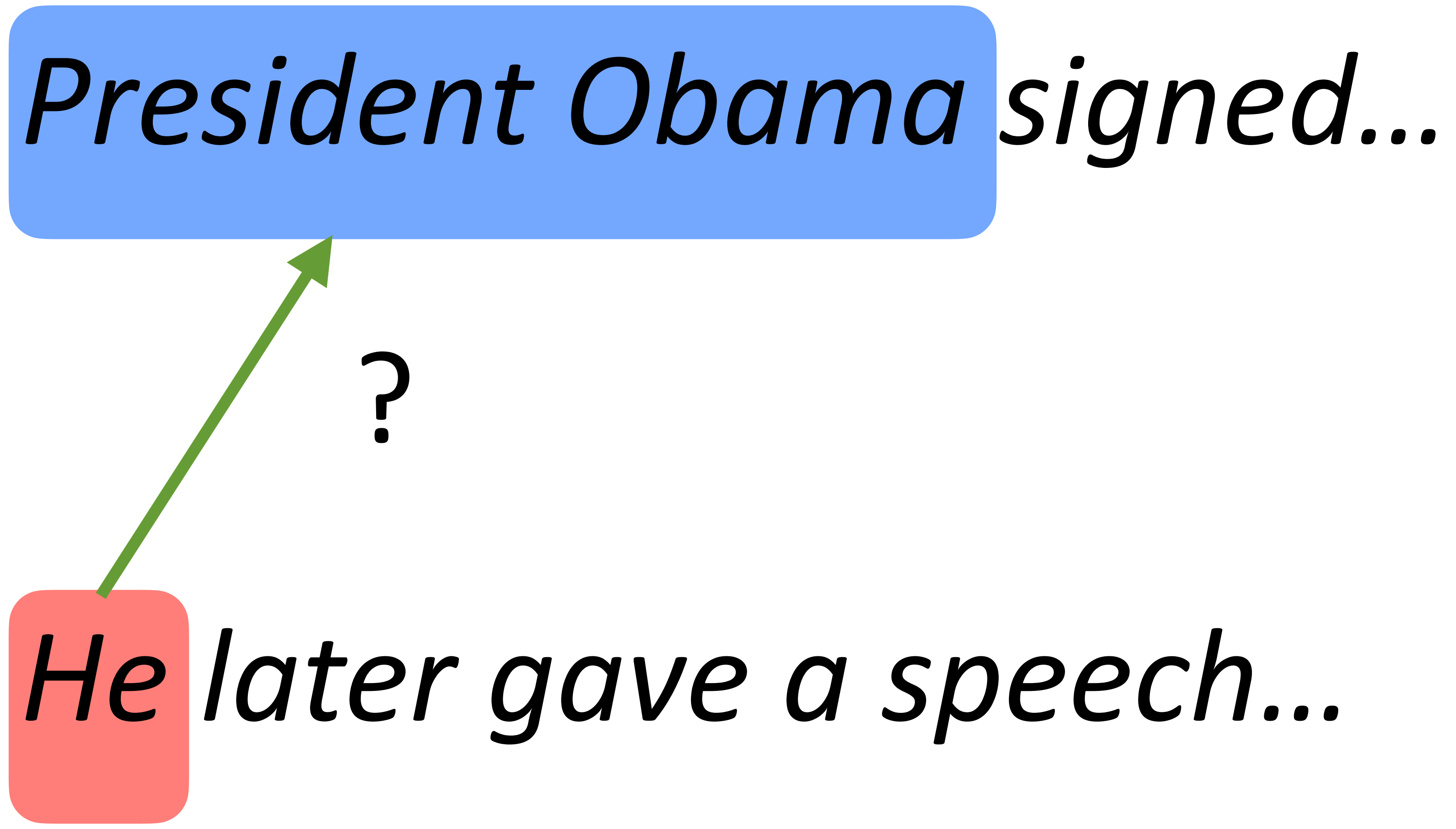
Wang and Manning (2012)

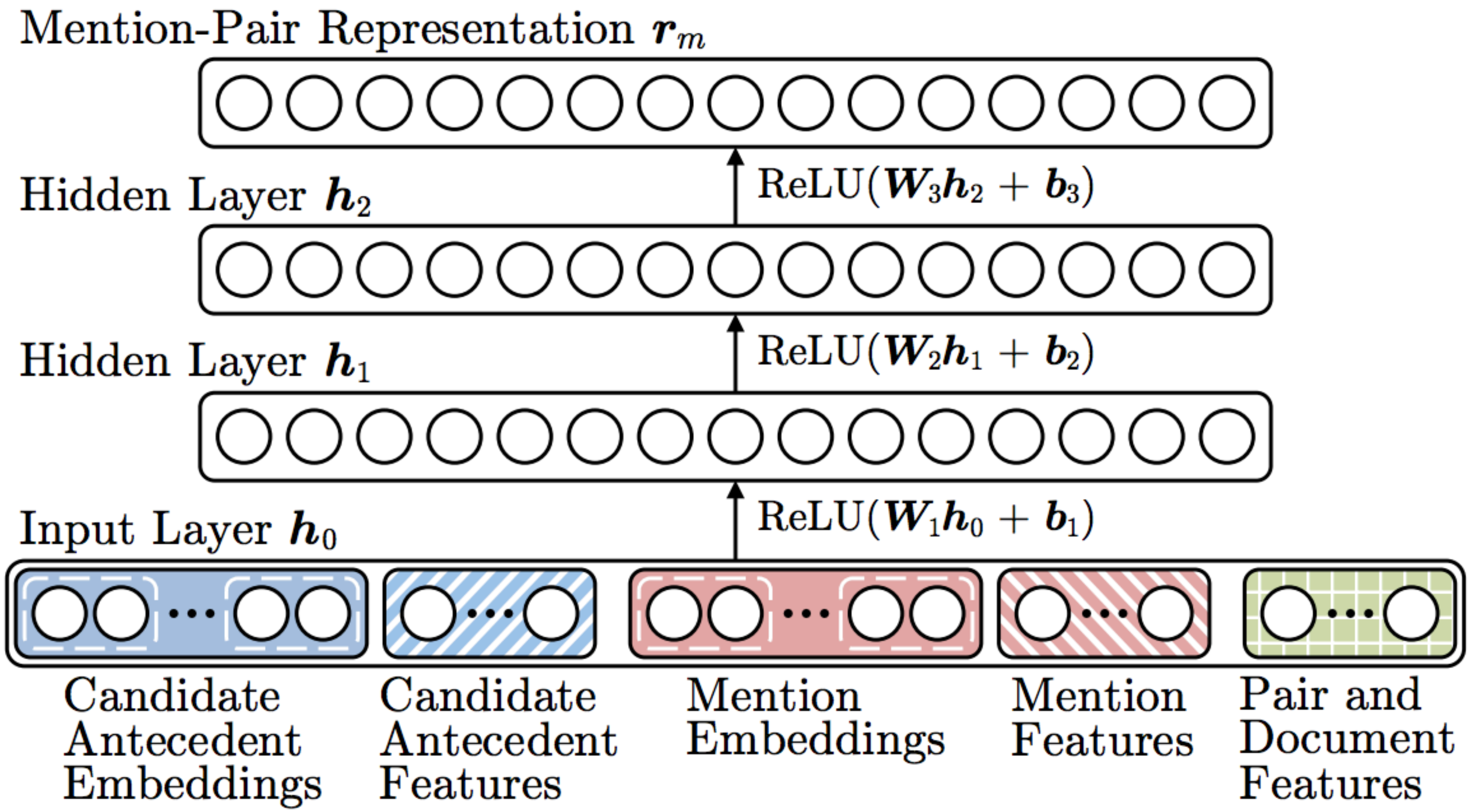
Tree RNNs / CNNS / LSTMS

Kim (2014)

Coreference Resolution

- Feedforward networks identify coreference arcs





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

Implementation Details

Computation Graphs

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

```
y = x * x   ──codegen──>   (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

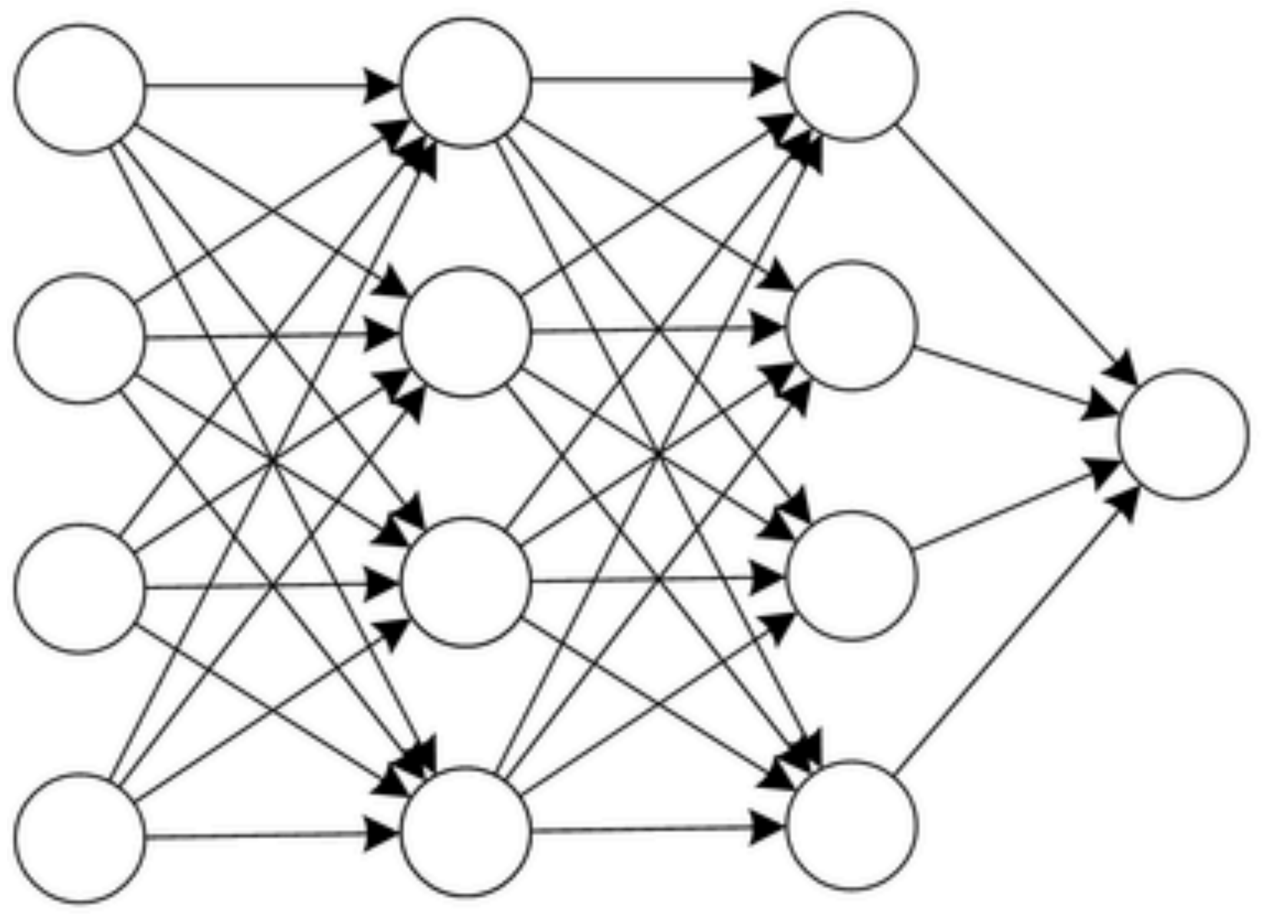
- Need to make the computation graph process a batch at the same time

```
# 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))
    ...
```

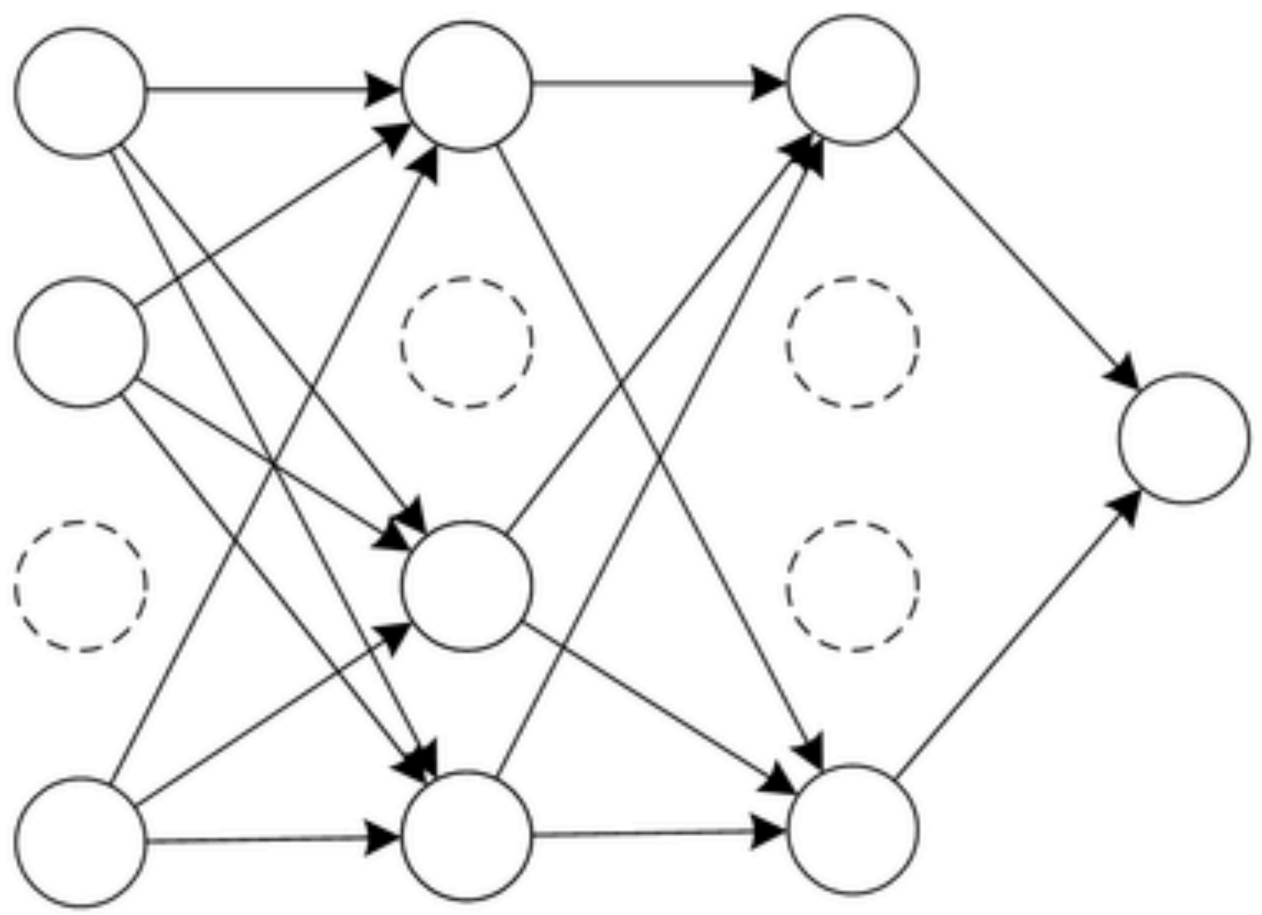
- Batch sizes from 1-100 often work well

Regularization: Dropout

- Very simple!
- In each forward pass, randomly set the activations for some nodes (neurons) to zero.
- Probability of dropping is a hyper-parameter; 0.5 is common.



(a) Standard Neural Network



(b) Network after Dropout