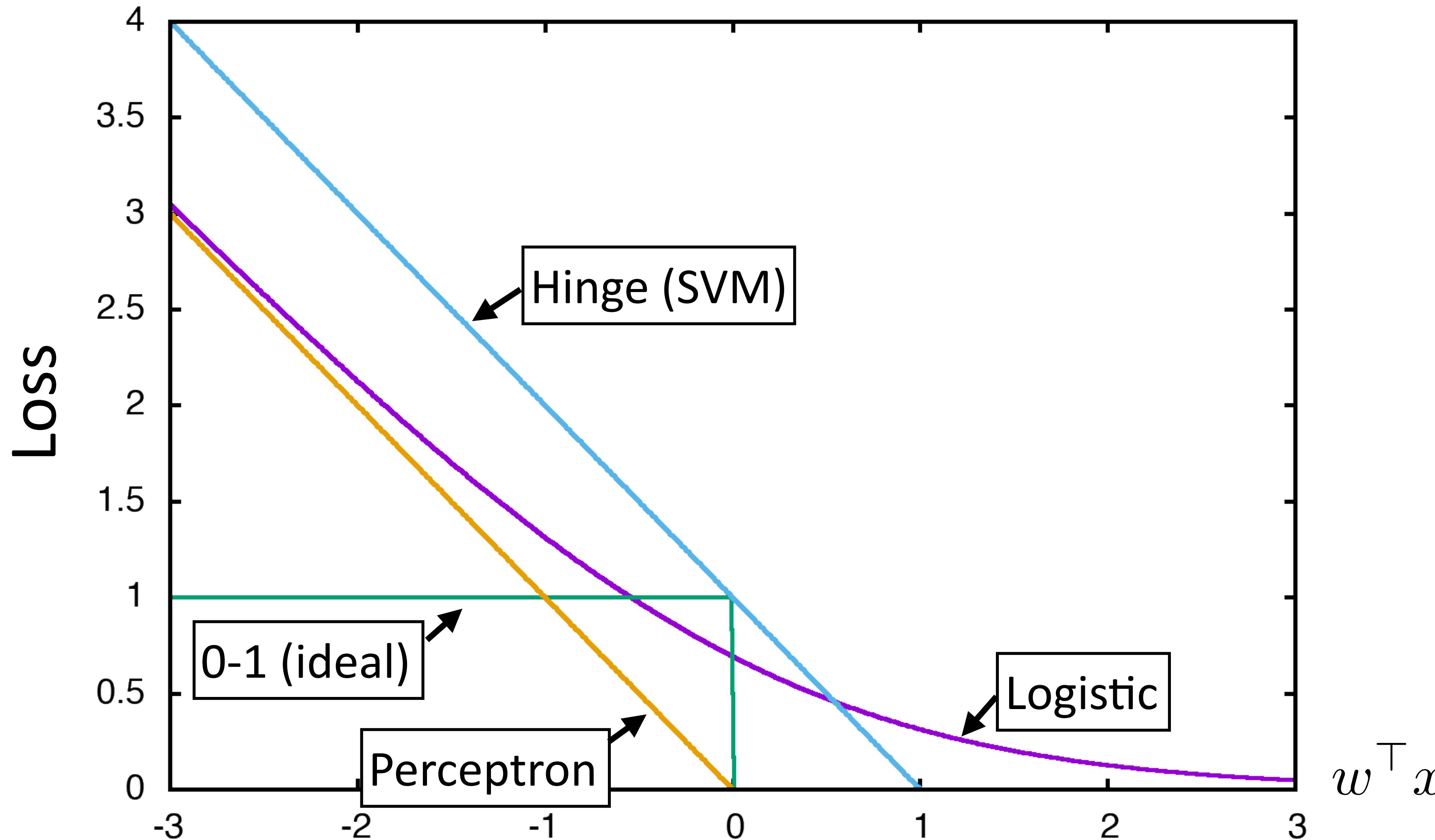


# Neural Networks

Wei Xu

(many slides from Greg Durrett)

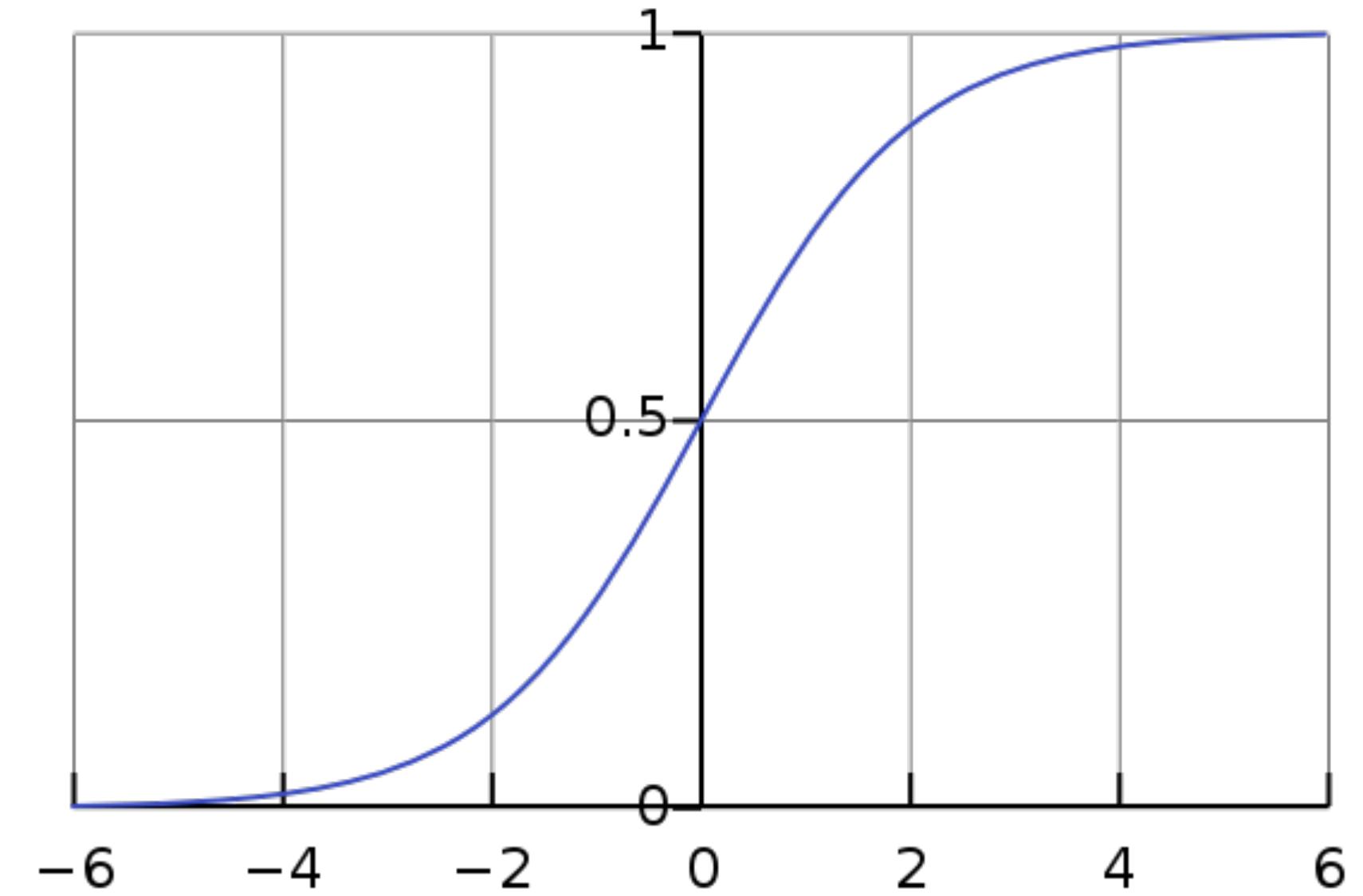
# Recap: Loss Functions



# 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  $P(y|x)$

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

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

# Recap: Multiclass Logistic Regression

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

↑  
sum over output  
space to normalize

*too many drug trials,  
too few patients*

Health: +2.2

Sports: +3.1

Science: -0.6

$w^\top f(x, y)$

probabilities  
must be  $\geq 0$

6.05  
22.2  
0.55

unnormalized  
probabilities

$\exp$

normalize

probabilities  
must sum to 1

0.21  
0.77  
0.02

probabilities

$\log(0.21) = -1.56$

compare

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

1.00  
0.00  
0.00  
correct (gold)  
probabilities

# Recap: Multiclass Logistic Regression

---

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



sum over output  
space to normalize

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

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

# Administrivia

---

- ▶ Homework 1 Graded (on Carmen)
- ▶ Homework 2 will be released soon.
- ▶ Reading: Eisenstein 3.1-3.3, Jurafsky+Martin 7.1-7.4

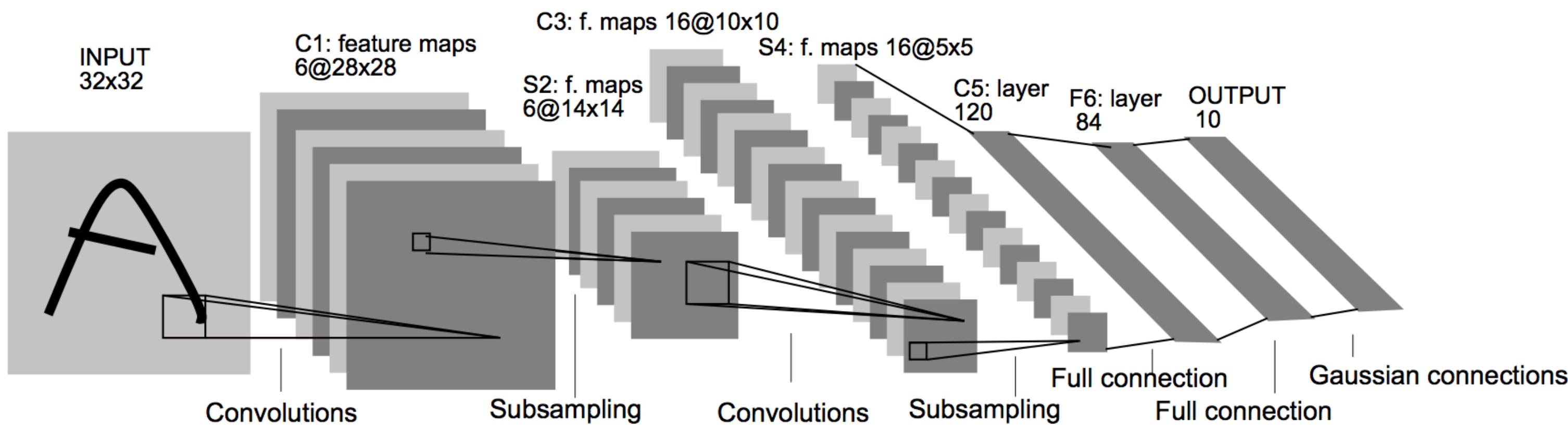
# This Lecture

---

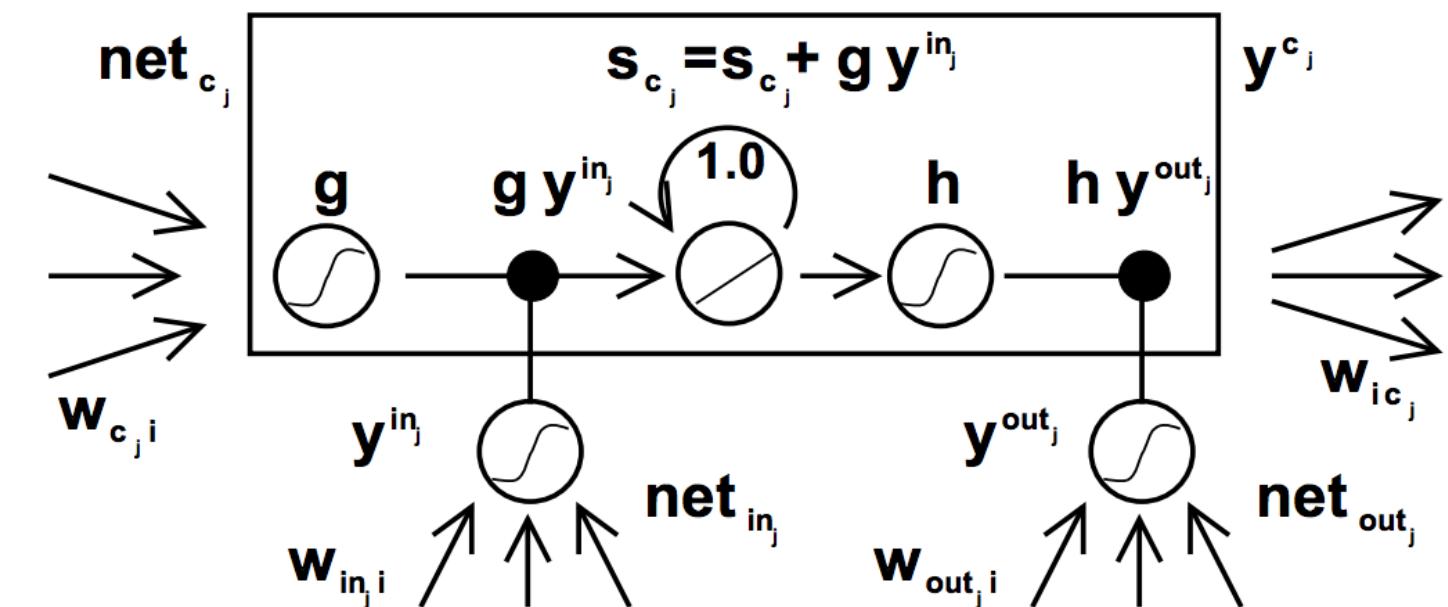
- ▶ Neural network history
- ▶ Neural network basics
- ▶ Feedforward neural networks + backpropagation
- ▶ Applications
- ▶ Implementing neural networks (if time)

# History: NN “dark ages”

- ▶ ConvNets: applied to MNIST by LeCun in 1998



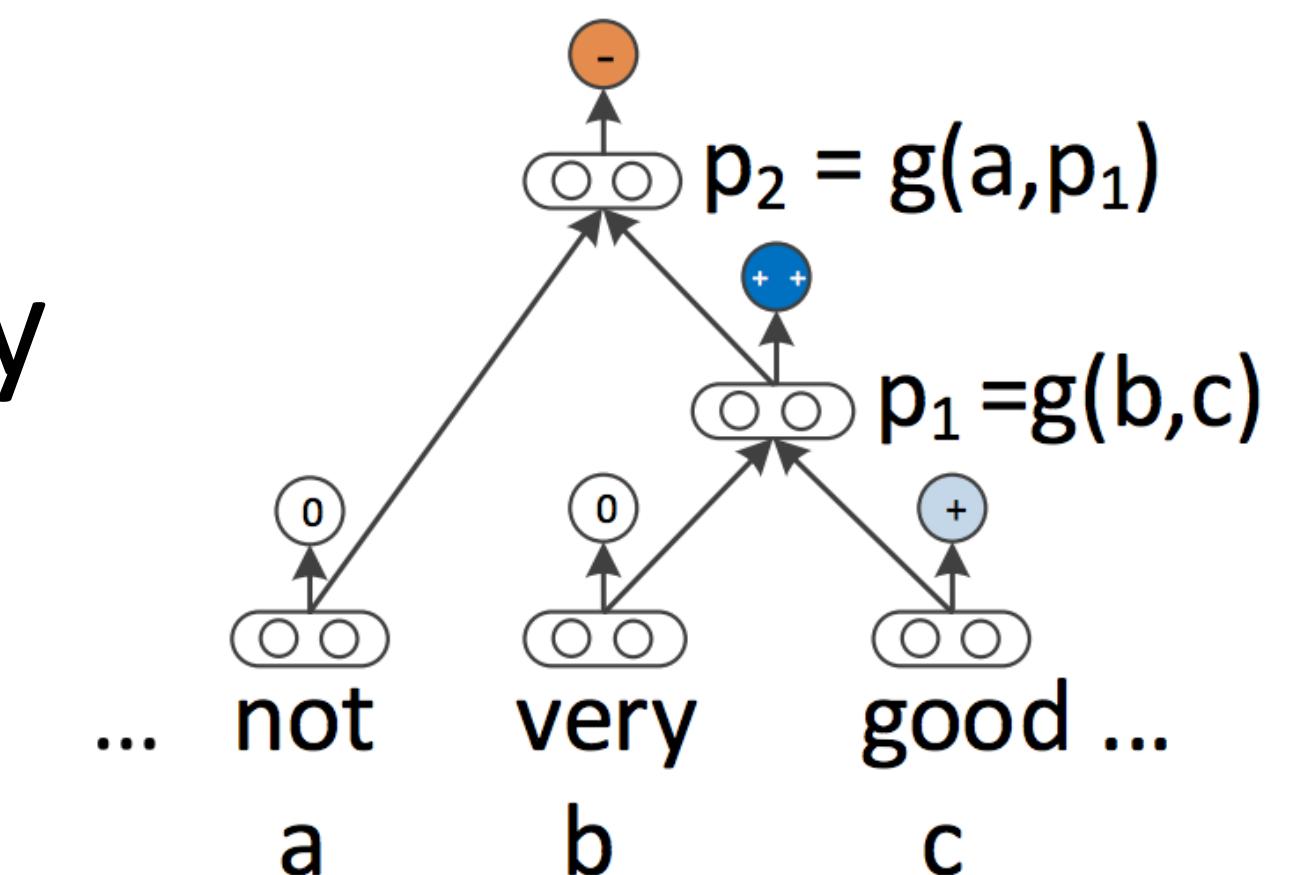
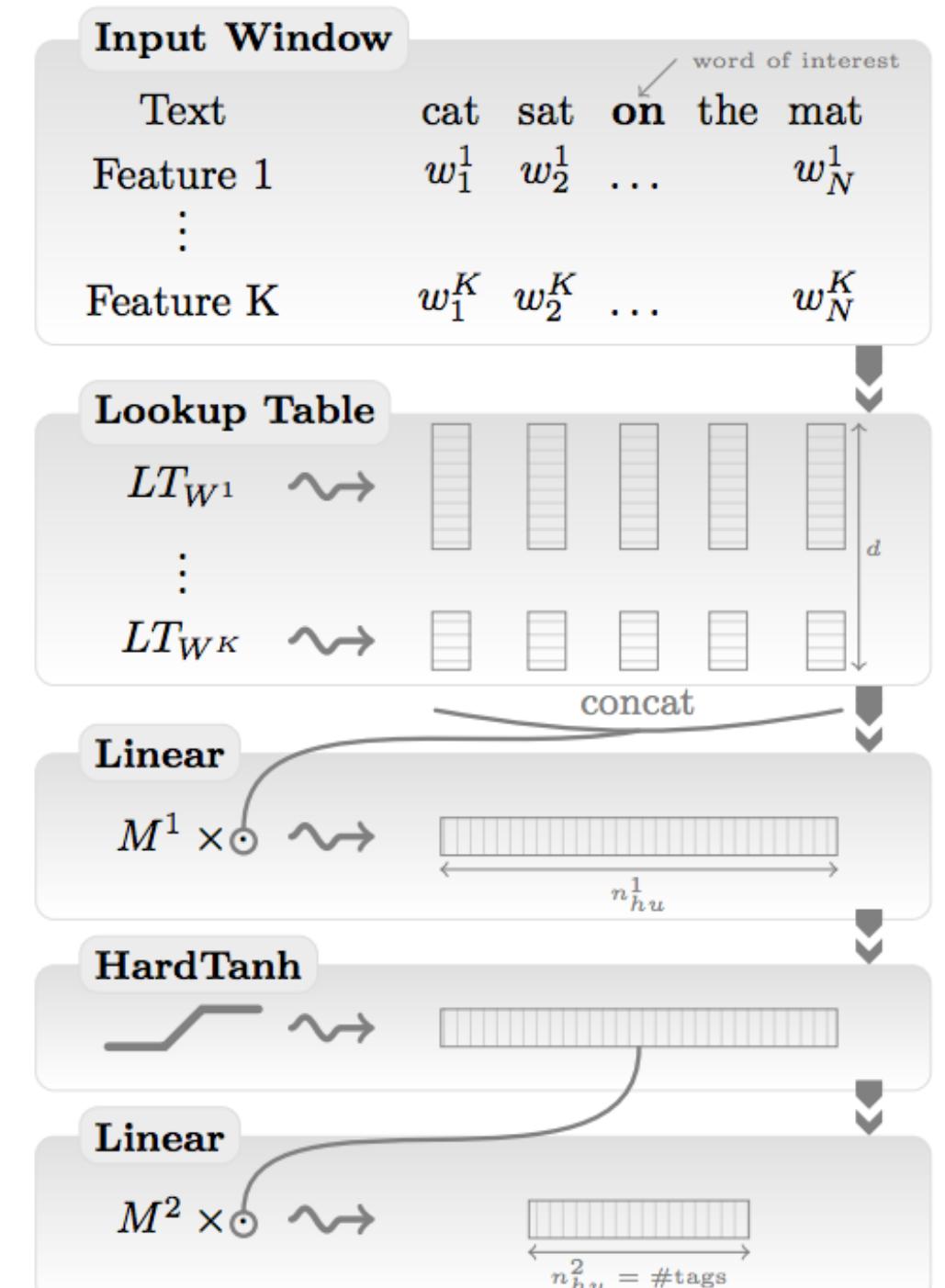
- ▶ LSTMs: Hochreiter and Schmidhuber (1997)



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

# 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
- ▶ Krizhevsky 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:  $\operatorname{argmax}_y w^\top f(x, y)$
- ▶ 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

---

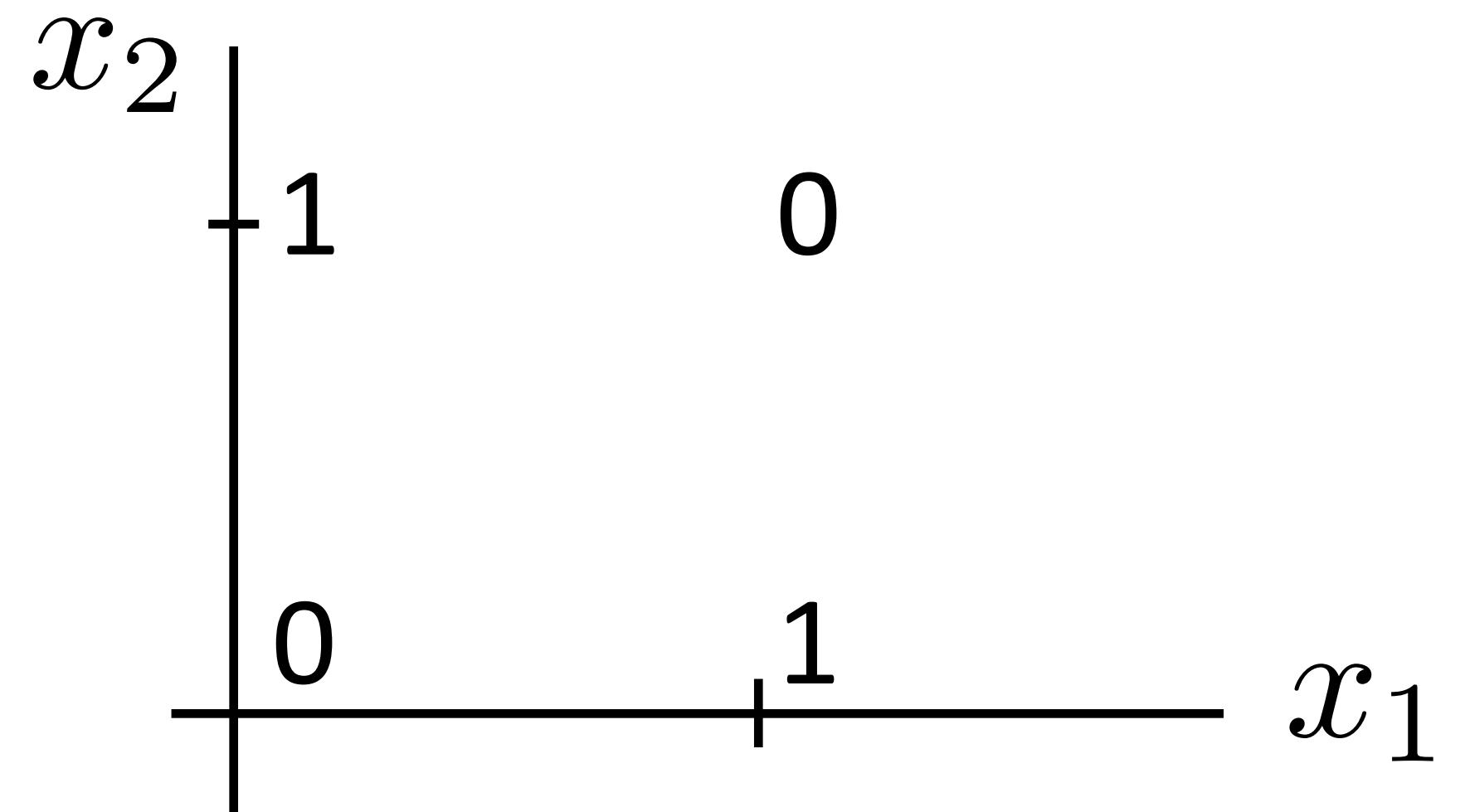
- ▶ Let's see how we can use neural nets to learn a simple nonlinear function

- ▶ Inputs  $x_1, x_2$

(generally  $\mathbf{x} = (x_1, \dots, x_m)$ )

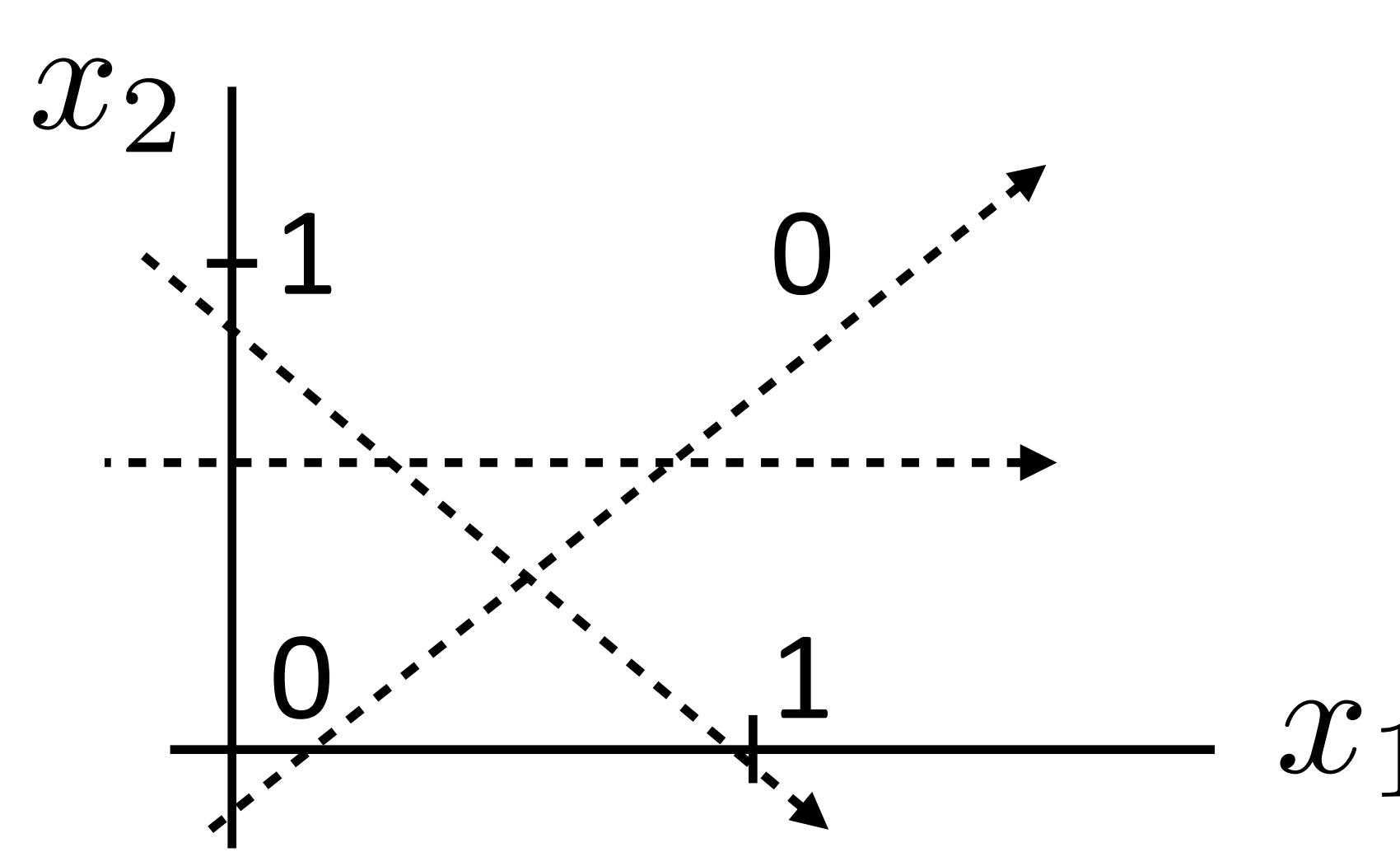
- ▶ Output  $y$

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



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

# Neural Networks: XOR



$$y = a_1x_1 + a_2x_2$$

$$y = a_1x_1 + a_2x_2 + a_3 \tanh(x_1 + x_2)$$

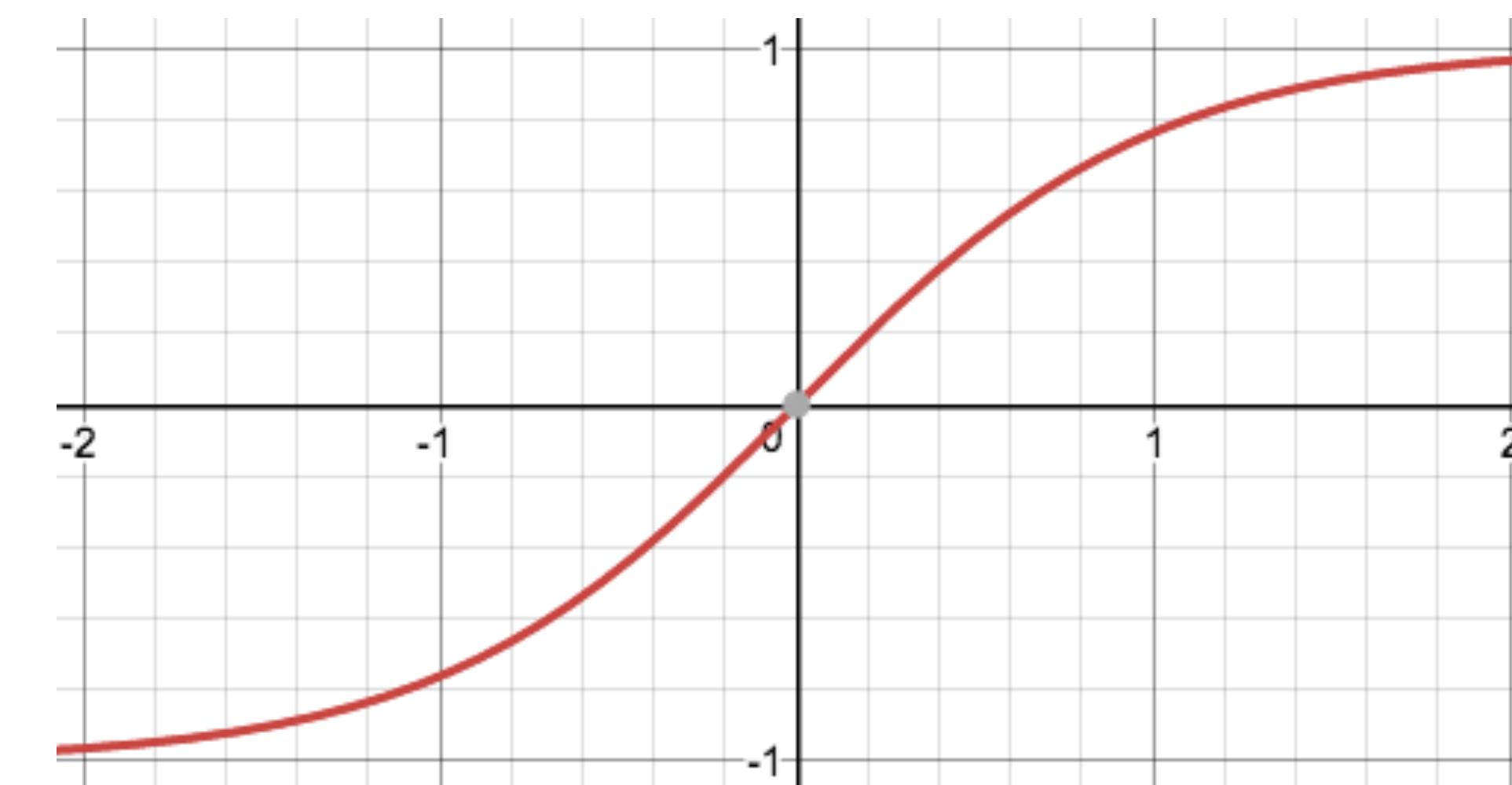
"or"

X

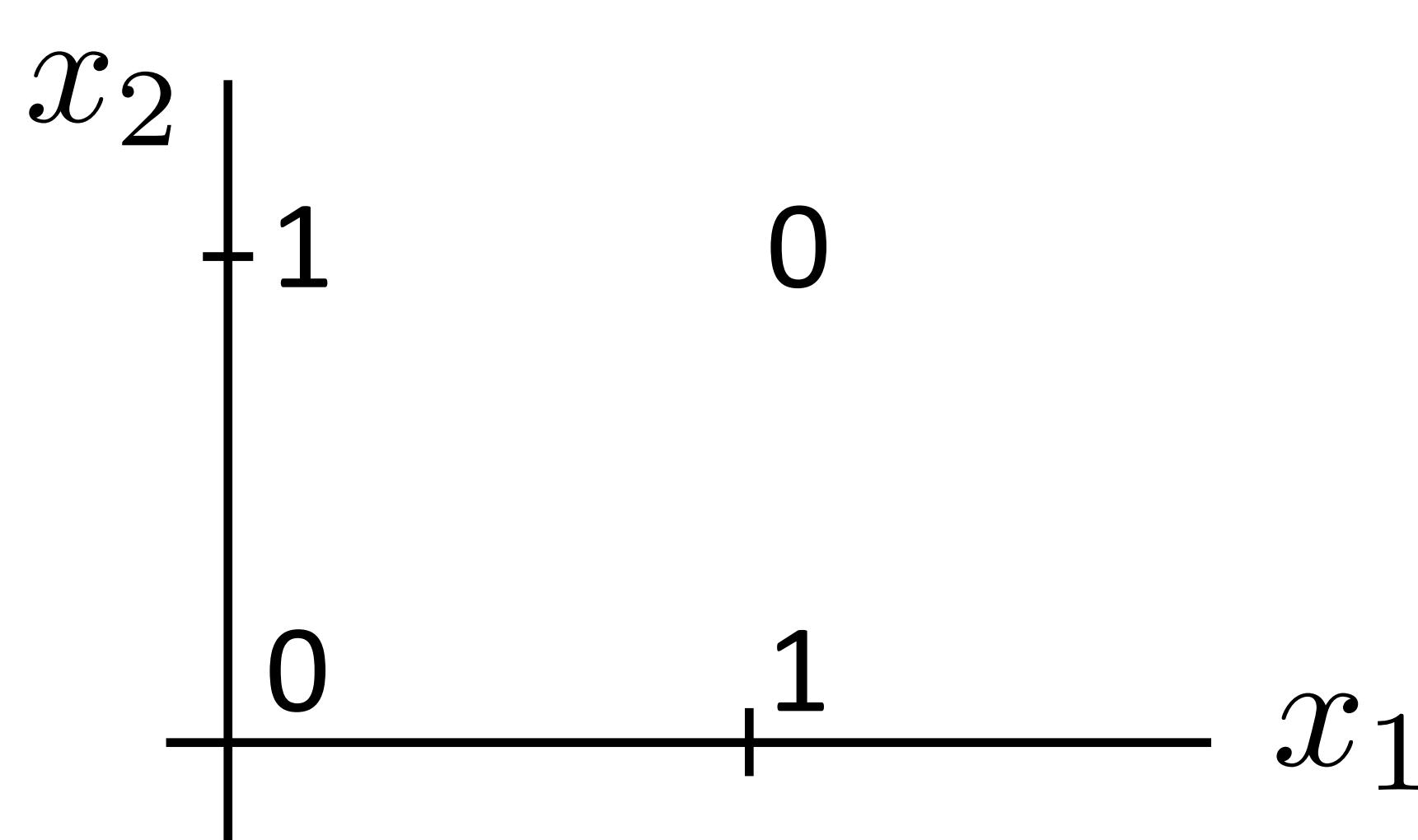


(looks like action potential in neuron)

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



# 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                      |

$$y = a_1 x_1 + a_2 x_2$$

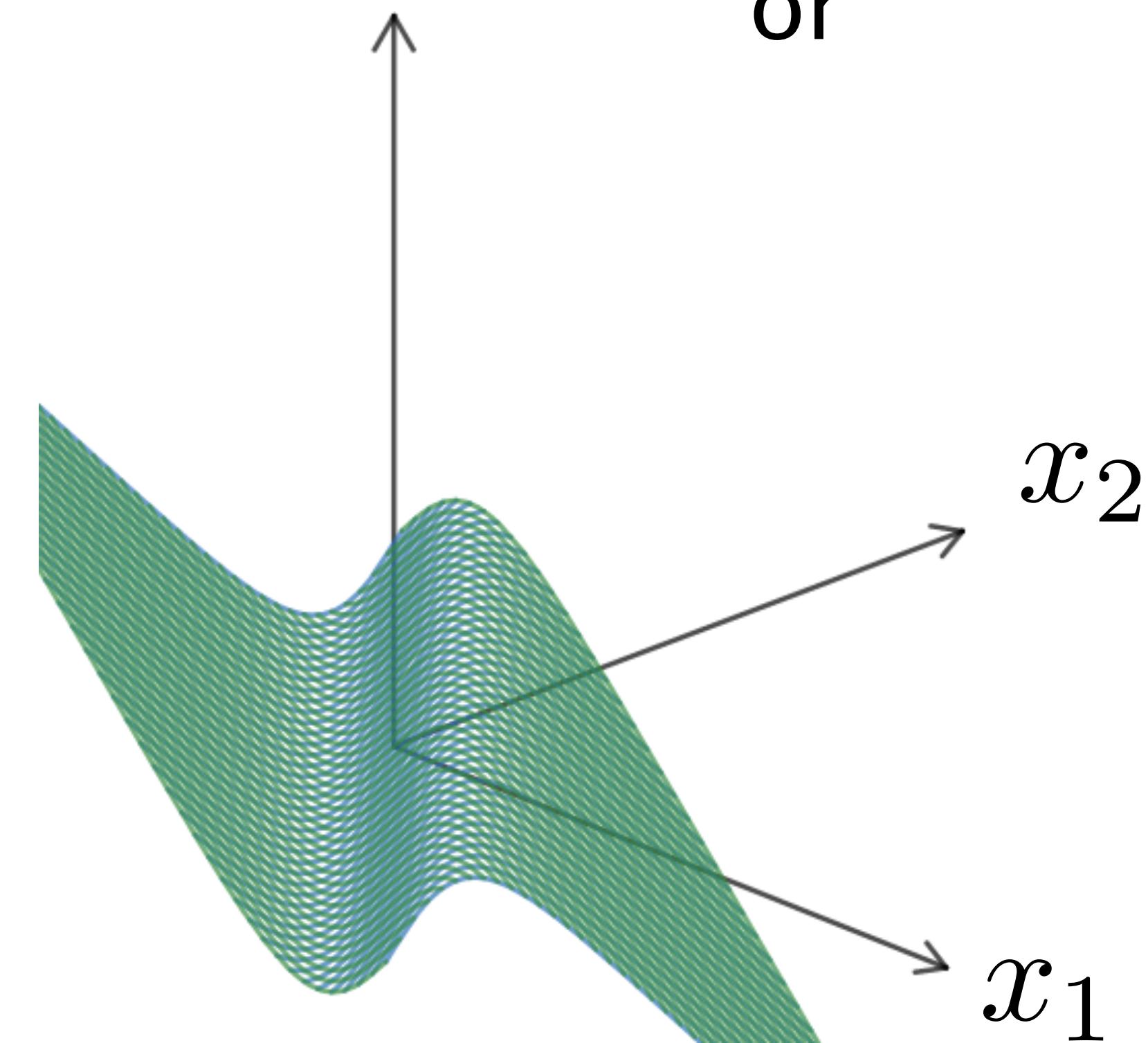
$$y = a_1 x_1 + a_2 x_2 + a_3 \tanh(x_1 + x_2)$$

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

“or”

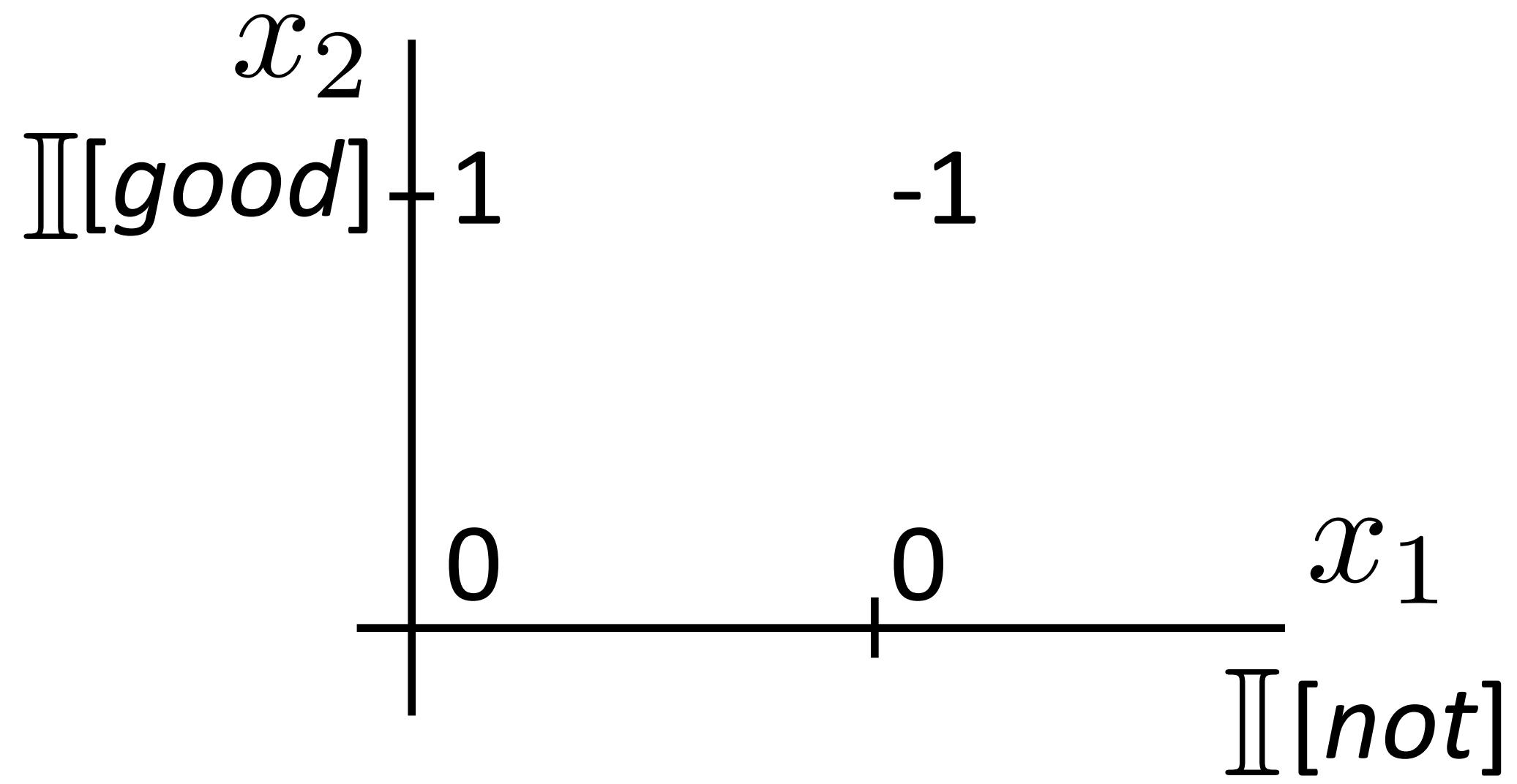
X

✓



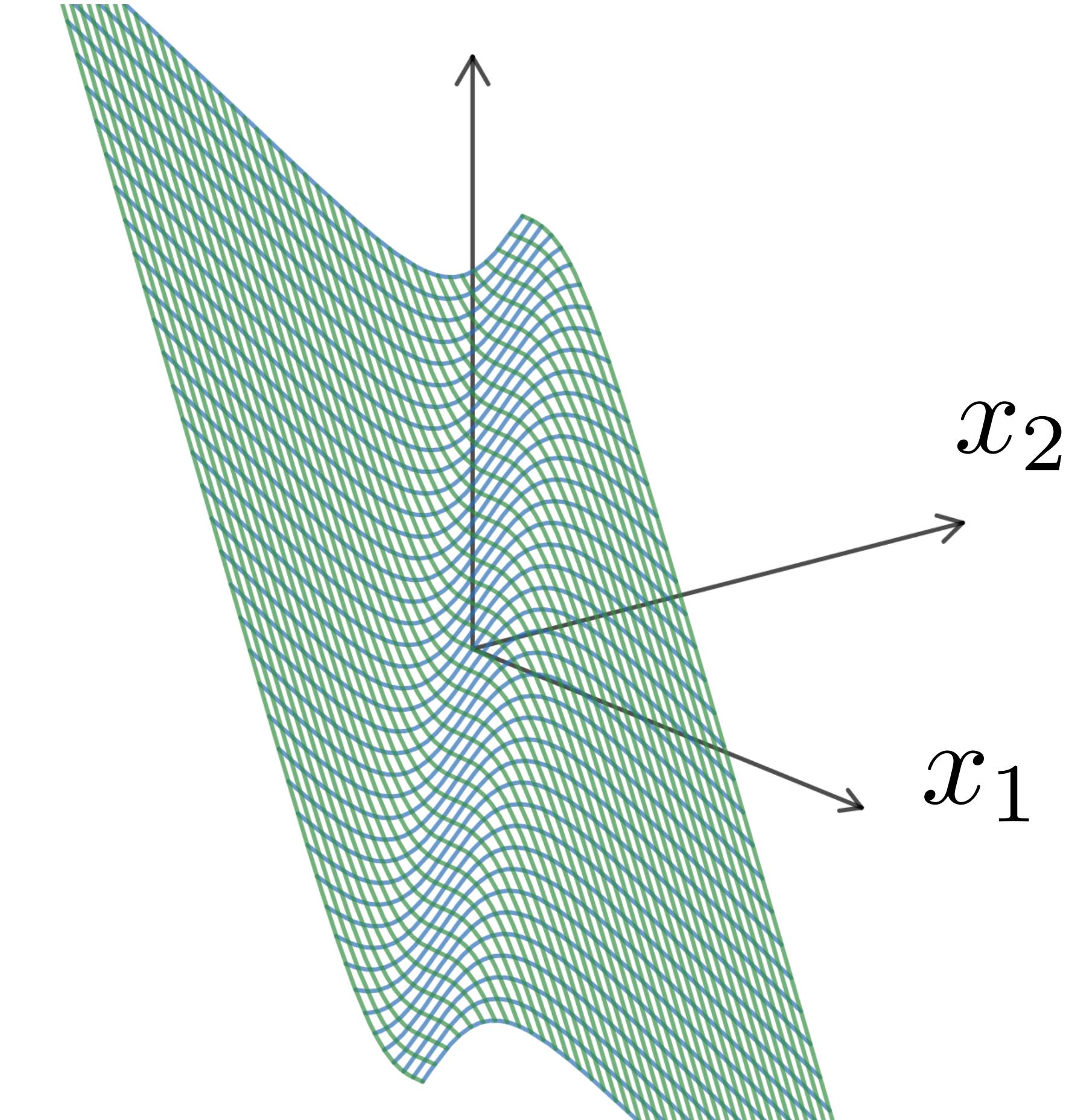
# Neural Networks: XOR

---



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

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



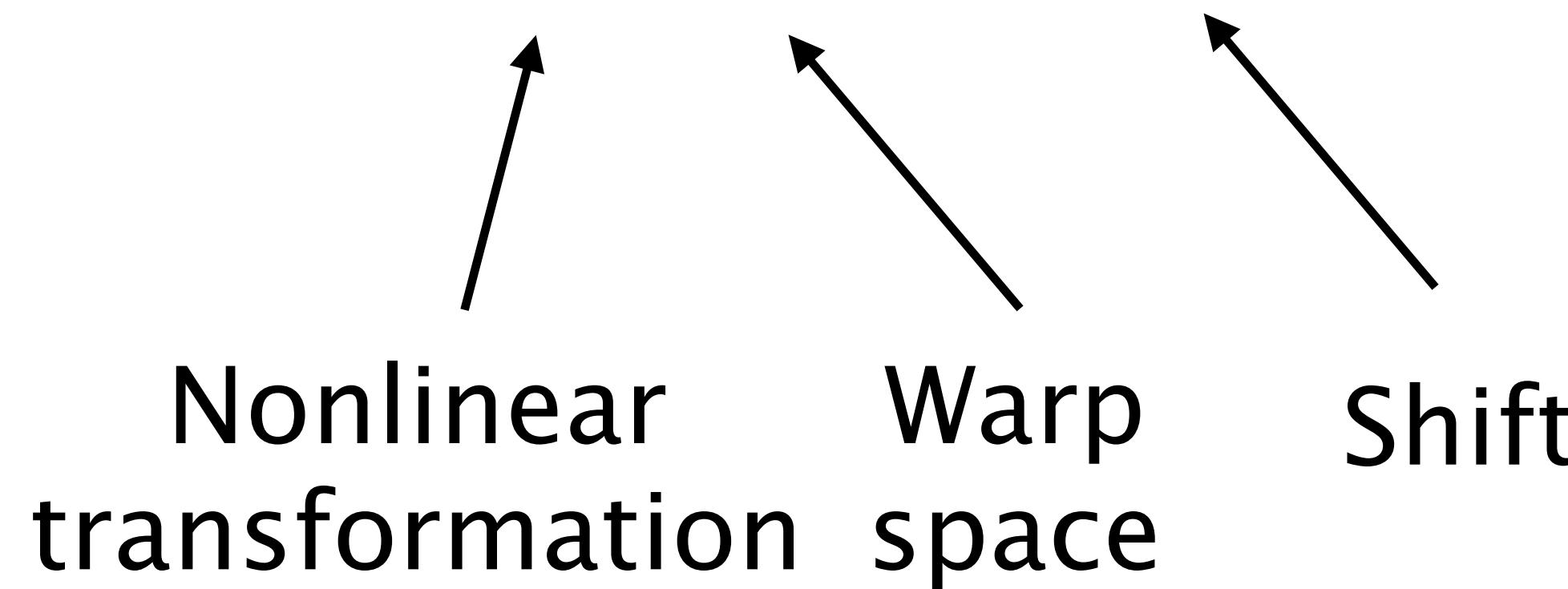
# Neural Networks

---

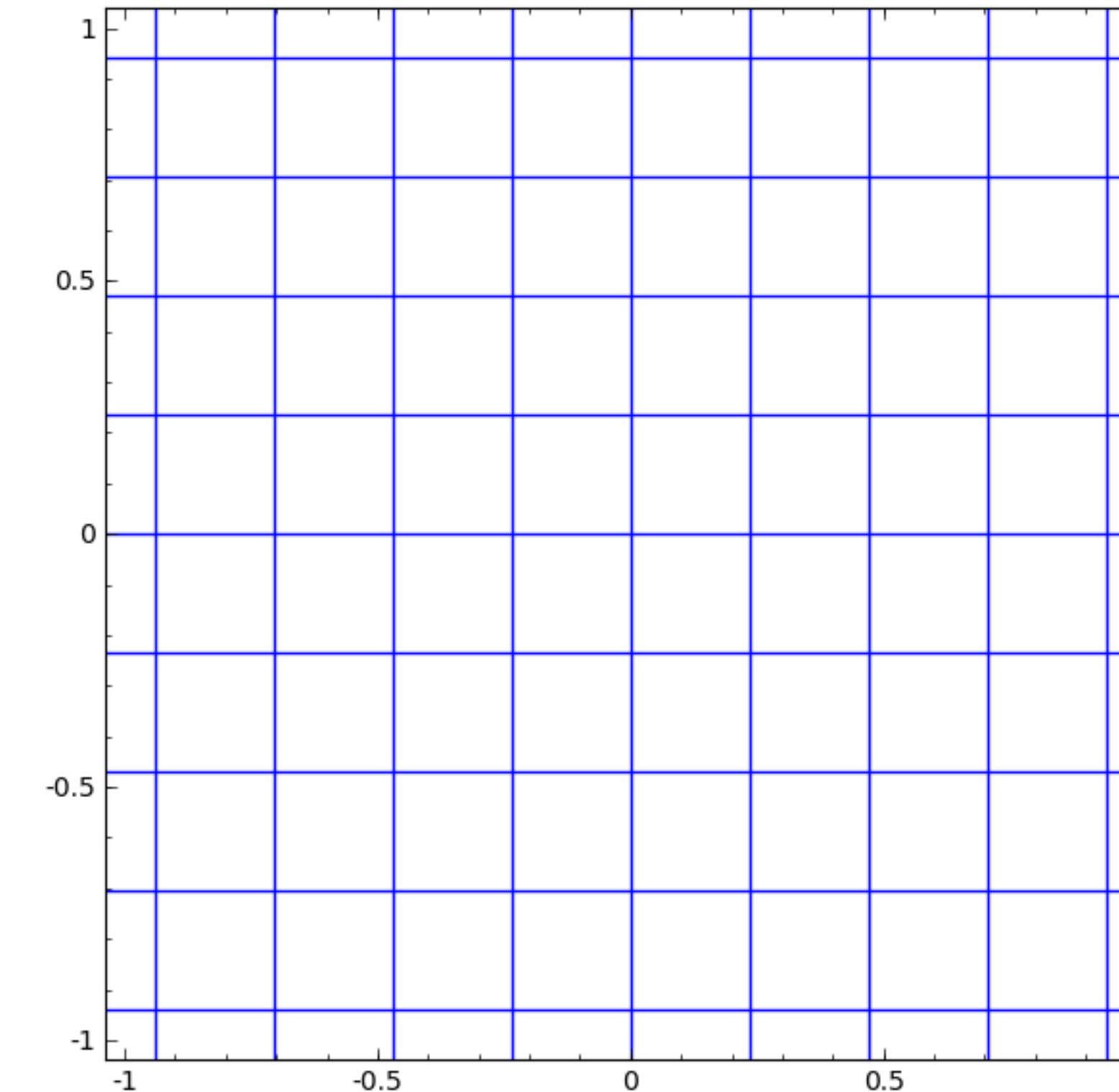
Linear model:  $y = \mathbf{w} \cdot \mathbf{x} + b$

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

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



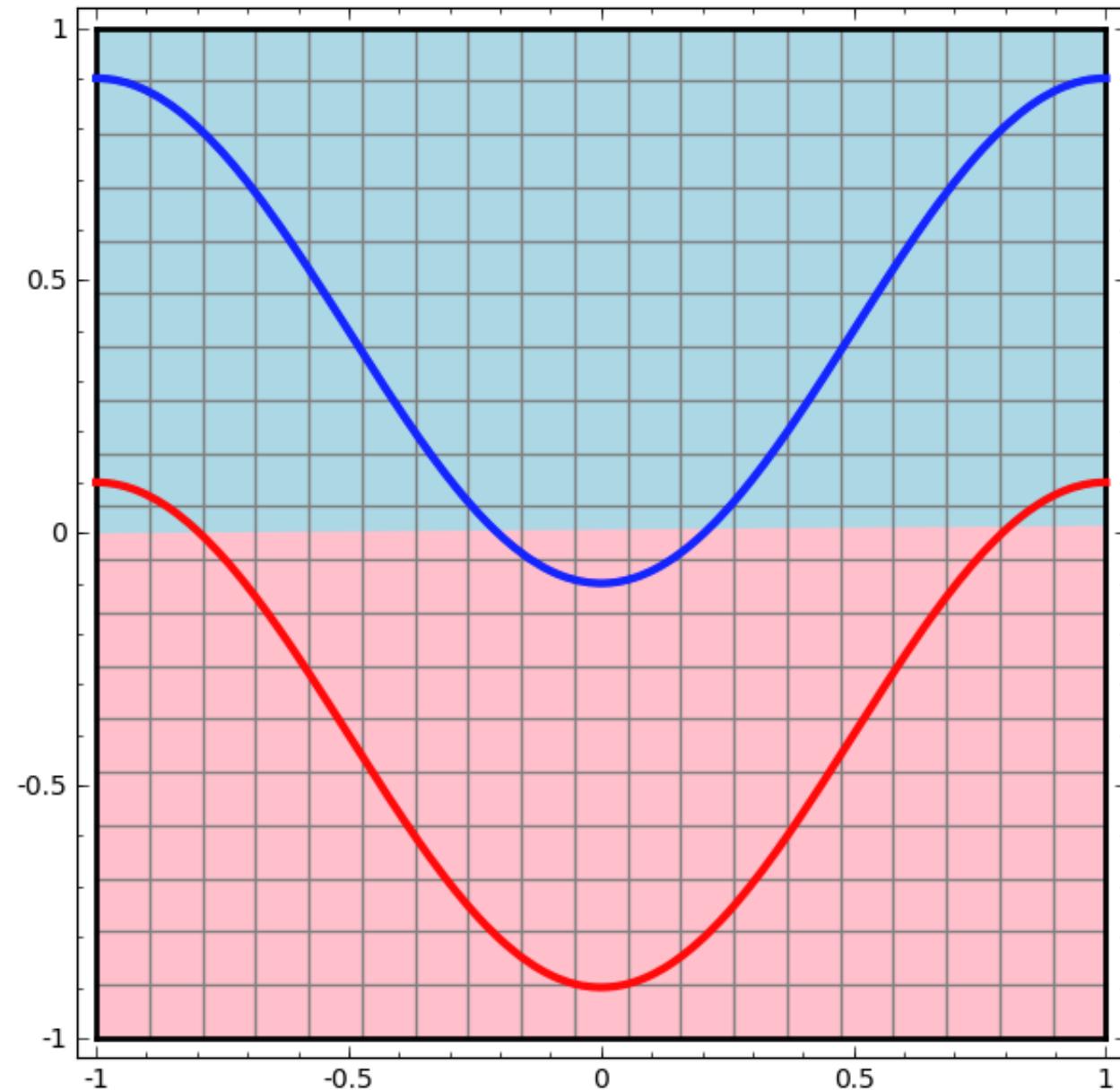
tanh



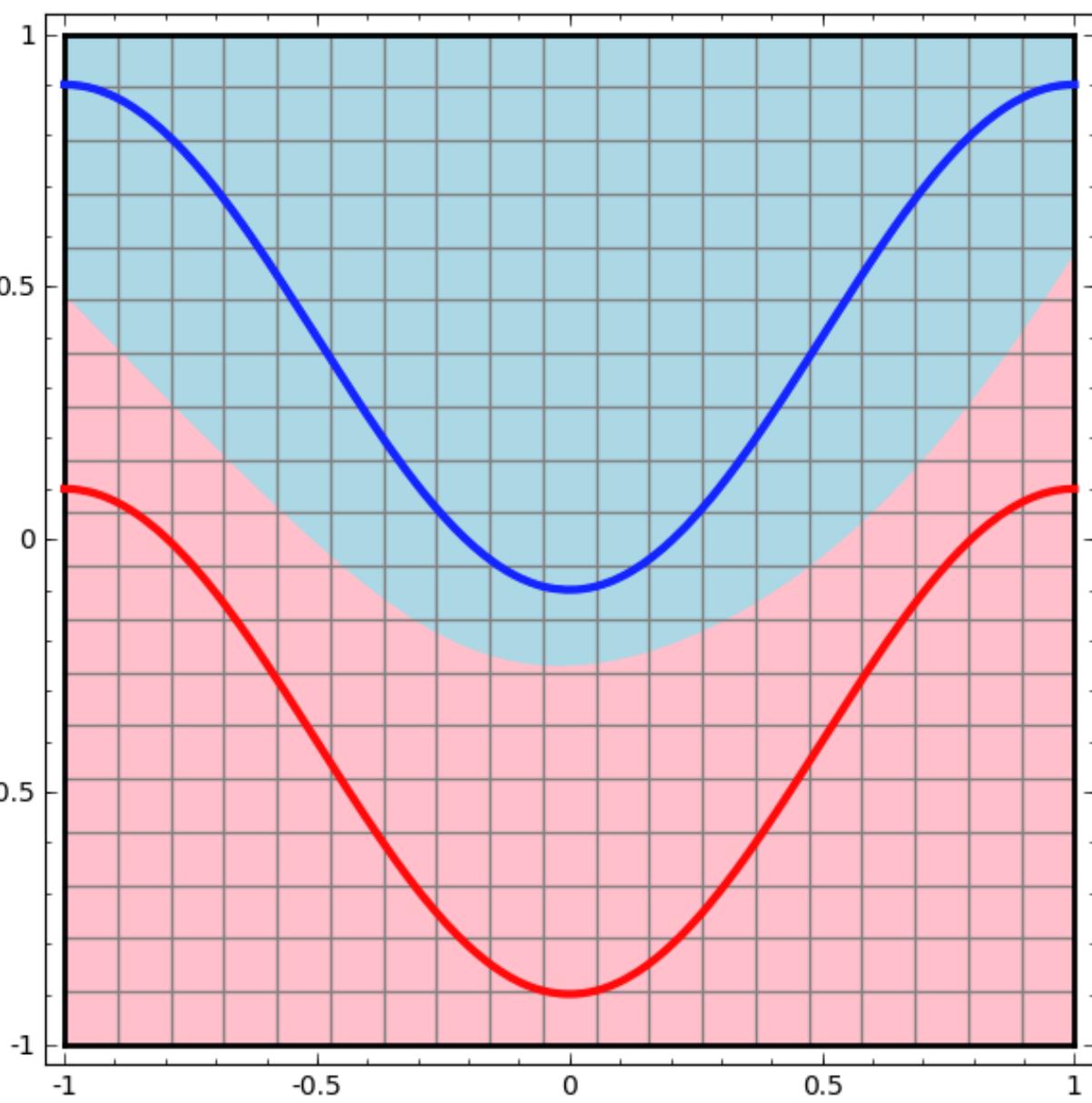
# Neural Networks

---

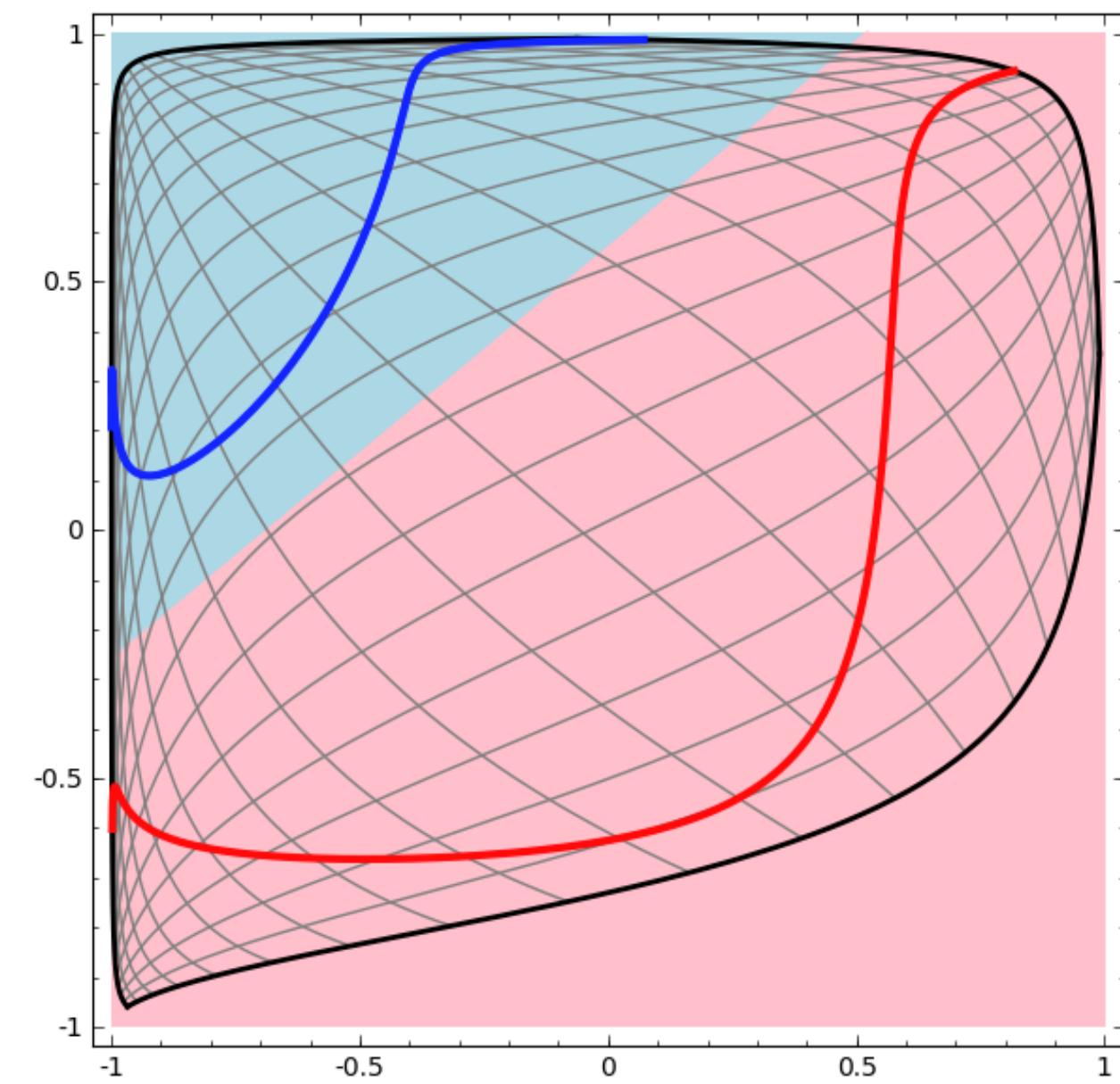
Linear classifier



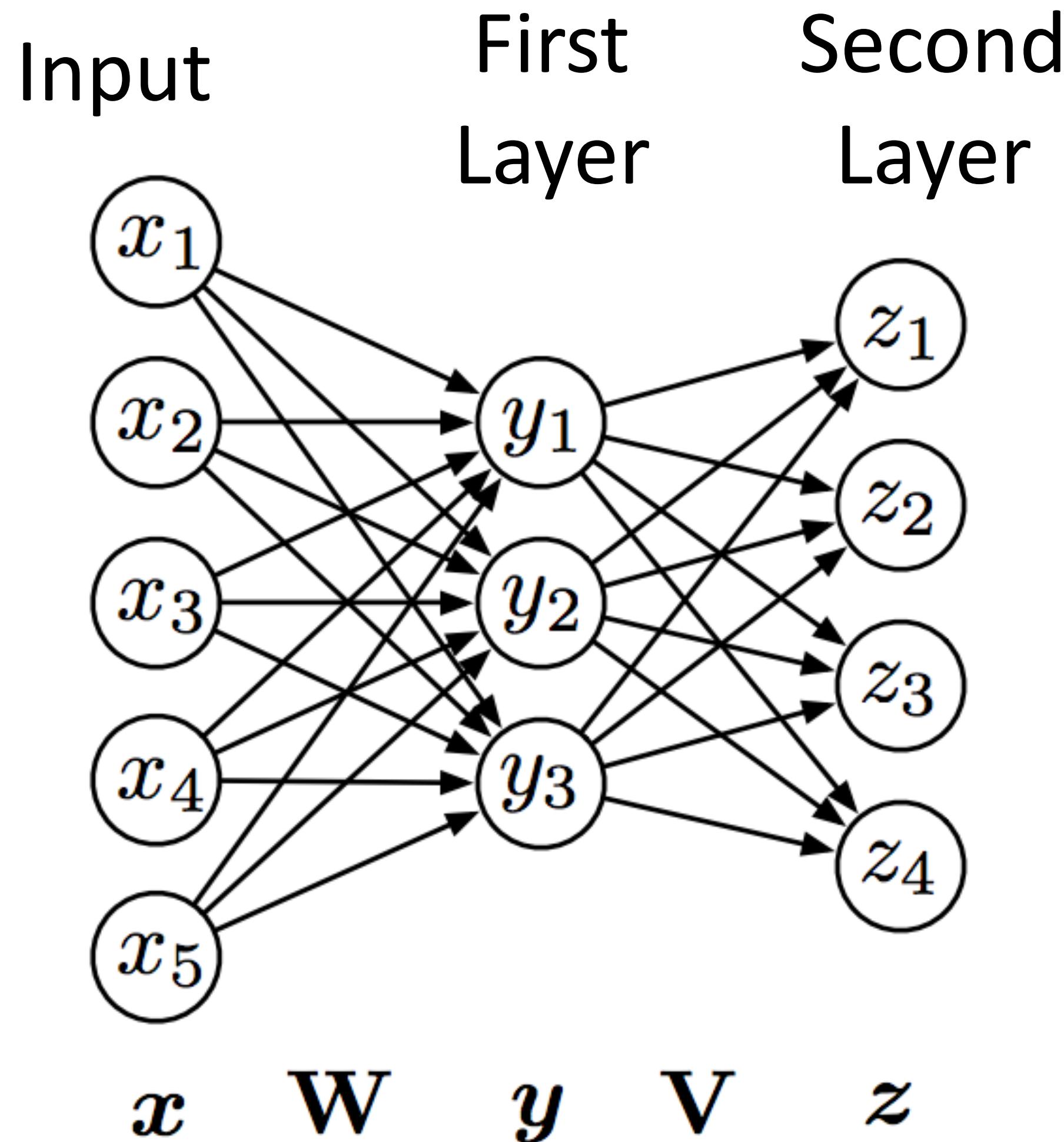
Neural network



...possible because  
we transformed the  
space!



# Deep Neural Networks



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

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

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

output of first layer

“Feedforward” computation (not recurrent)

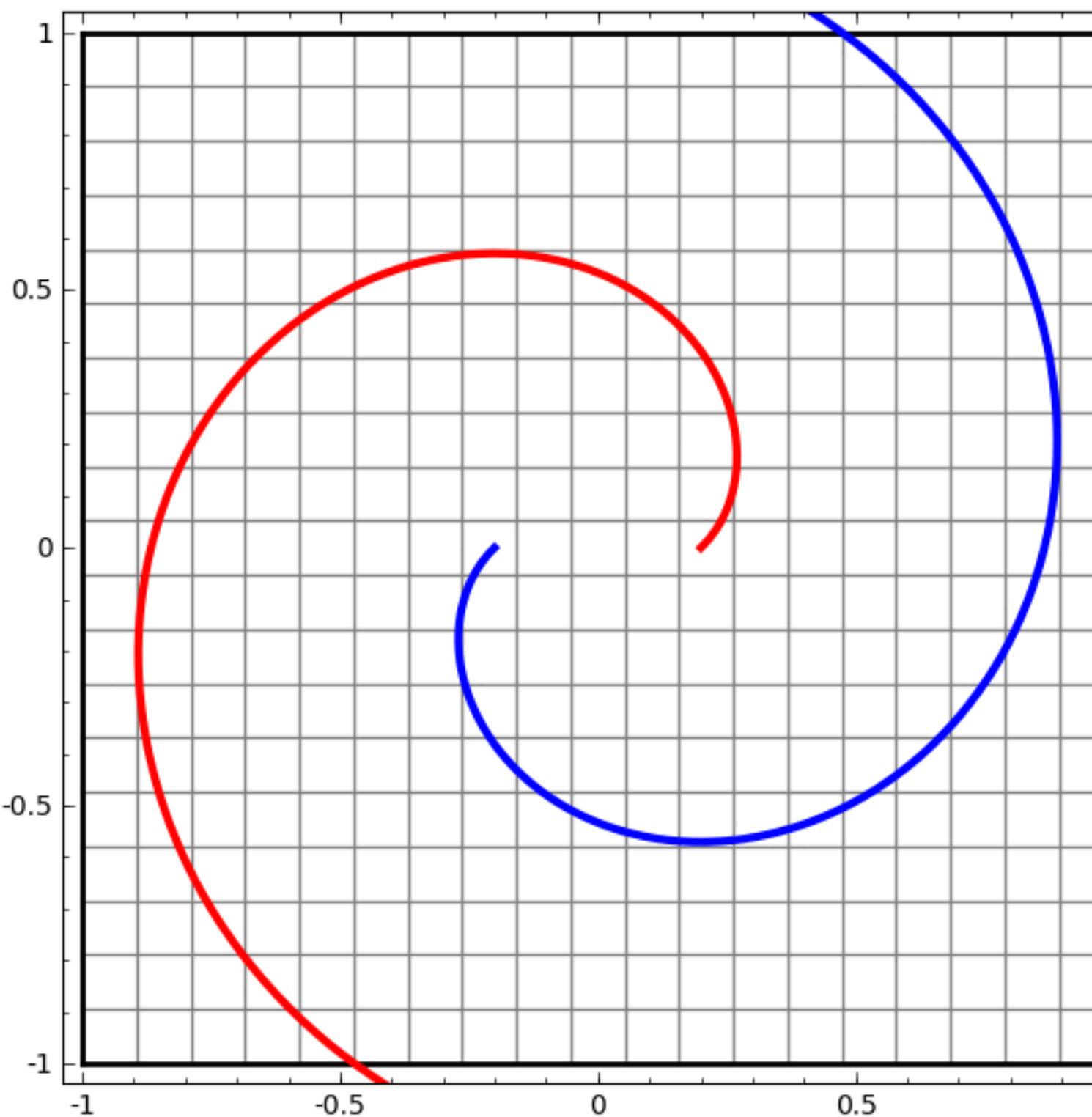
Check: what happens if no nonlinearity?  
More powerful than basic linear models?

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

Adopted from Chris Dyer

# Deep Neural Networks

---



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

# Feedforward Networks, Backpropagation

# Logistic Regression with NNs

---

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

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

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

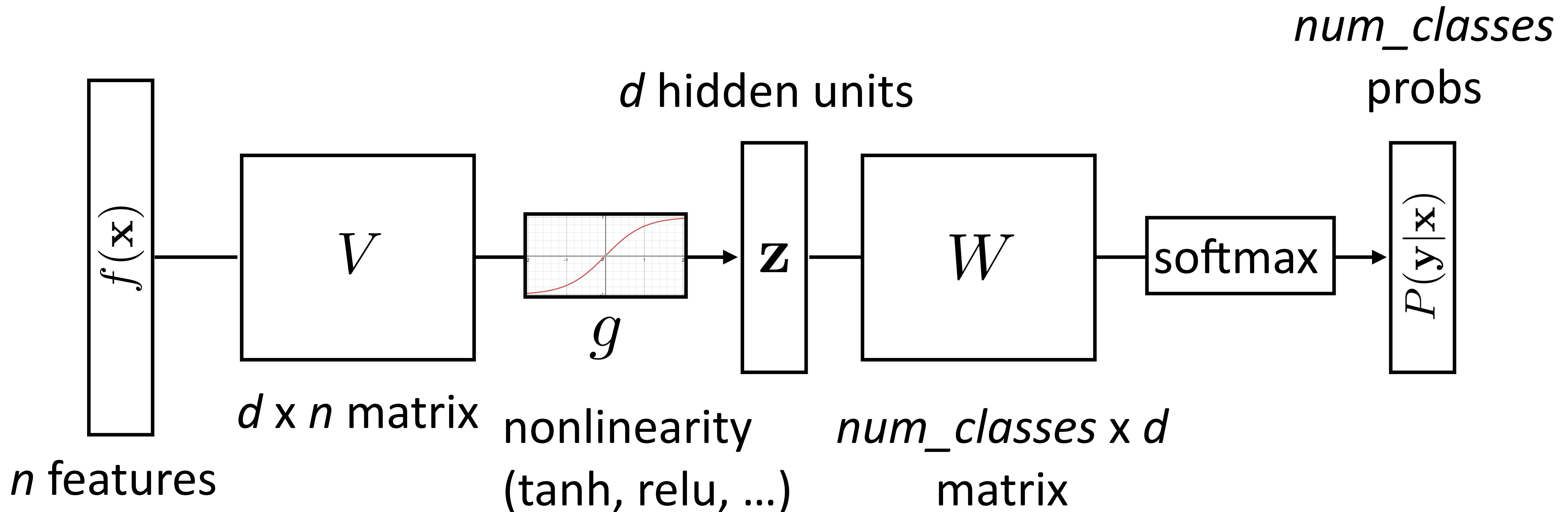
$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wf(\mathbf{x}))$$

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

- ▶ Single scalar probability
- ▶ Compute scores for all possible labels at once (returns vector)
- ▶ softmax: exps and normalizes a given vector
- ▶ Weight vector per class; W is [num classes x num feats]
- ▶ 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  $\mathbf{z}$  which is a hidden layer representation of the input, and then running standard logistic regression on the features that the network develops in  $\mathbf{z}$ .

# Training Neural Networks

---

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

- ▶ Maximize log likelihood of training data

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

- ▶  $i^*$ : 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

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

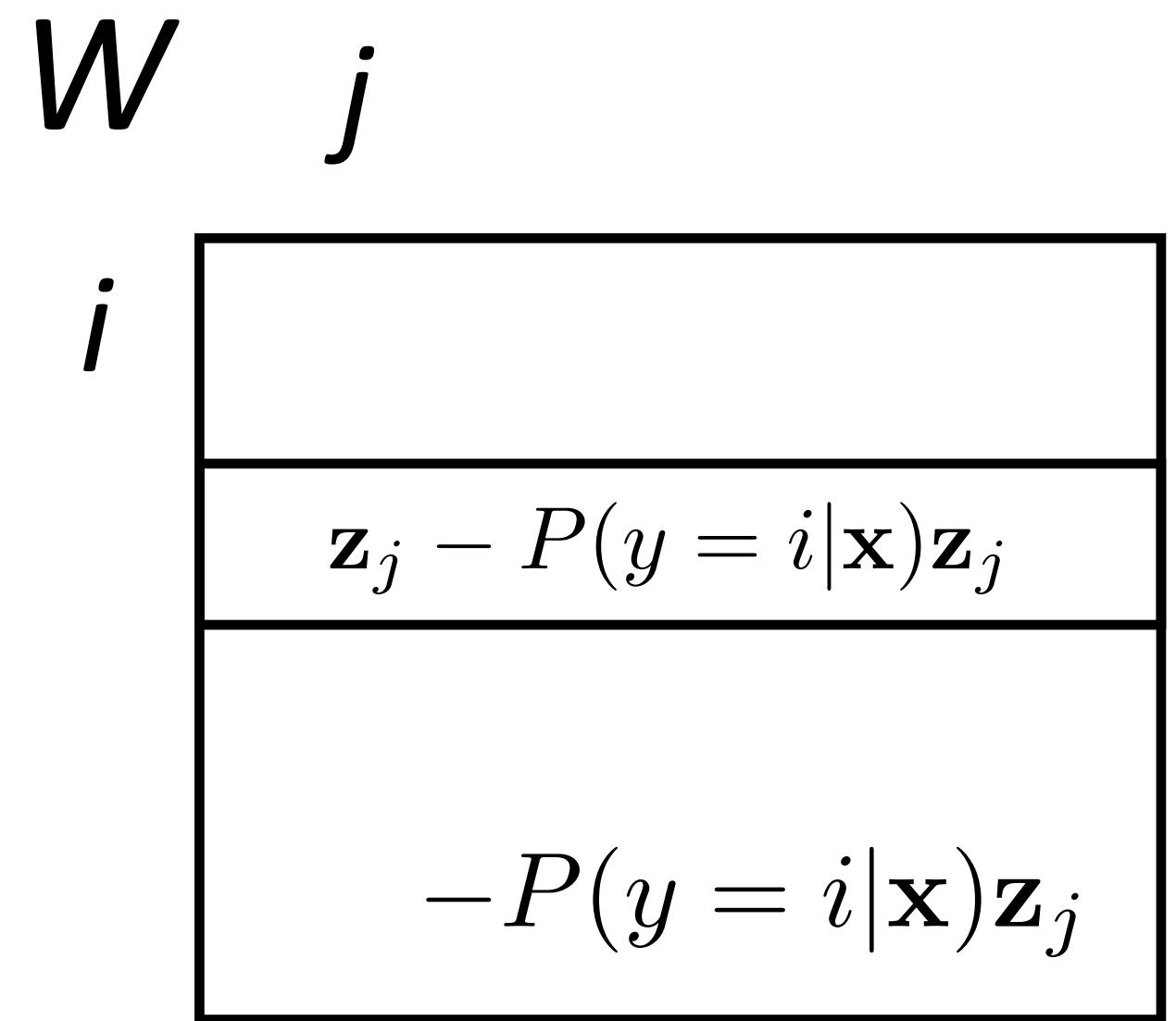
- Gradient with respect to  $W$

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

index of  
output space  $\gamma$

index of  
gold label

index of vector  $\mathbf{z}$

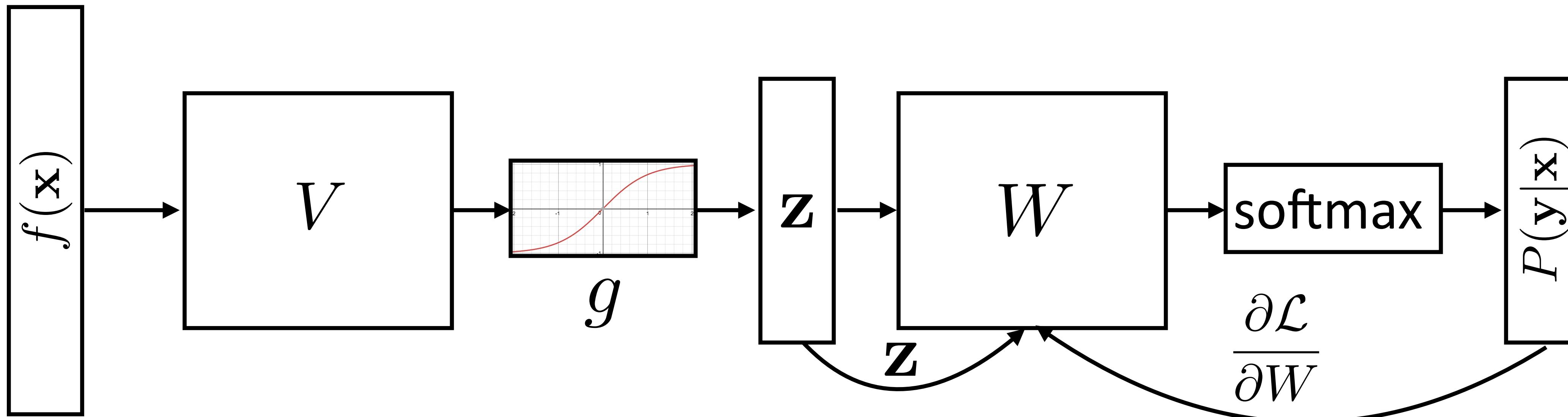


- Looks like logistic regression with  $\mathbf{z}$  as the features!

# Neural Networks for Classification

---

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



# 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...]

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

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

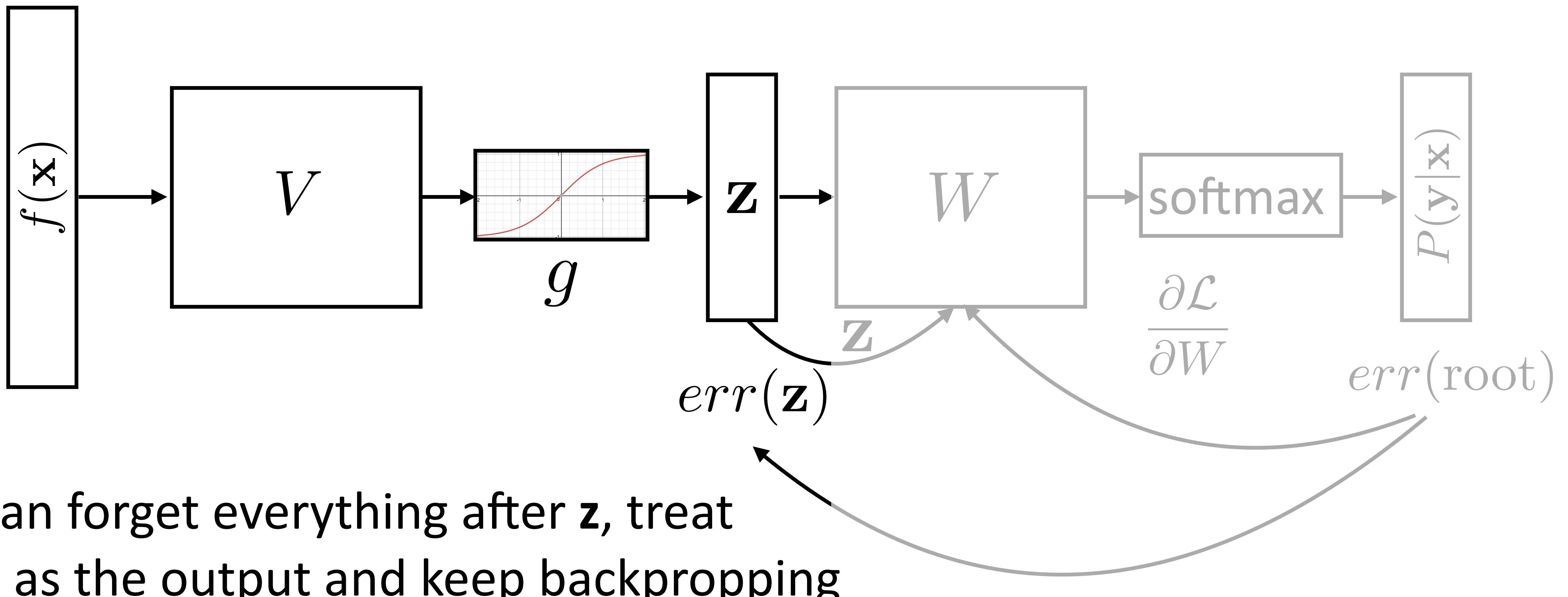
$\dim = d$

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

$\dim = \text{num\_classes}$

# Backpropagation: Picture

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



# Computing Gradients: Backpropagation

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

Activations at  
hidden layer

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

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

[some math...]

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

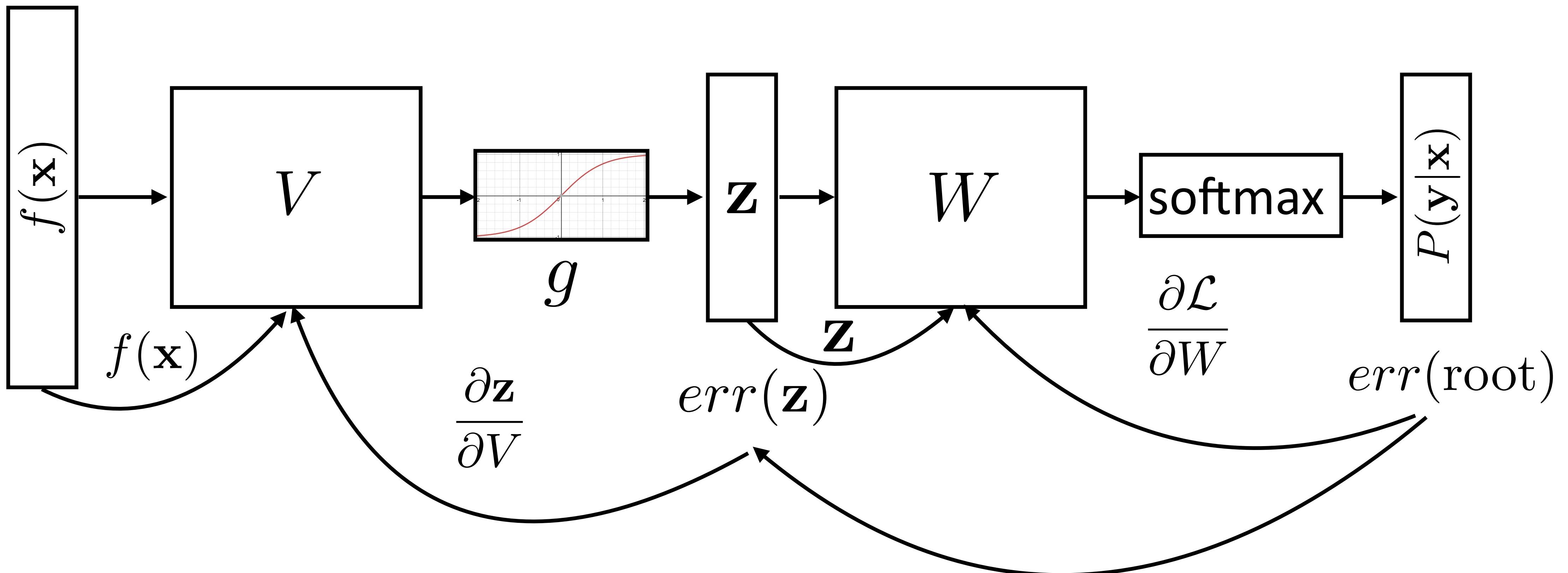
dim = num\_classes

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

dim = d

# Backpropagation: Picture

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



# 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  $err(\text{root})$  (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  $err(\mathbf{z})$  (matrix)
- ▶ Step 5+: continue backpropagation (compute  $err(f(\mathbf{x}))$  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

# Applications

# NLP with Feedforward Networks

- ▶ Part-of-speech tagging with FFNNs

??

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

previous word

- ▶ Word embeddings for each word form input

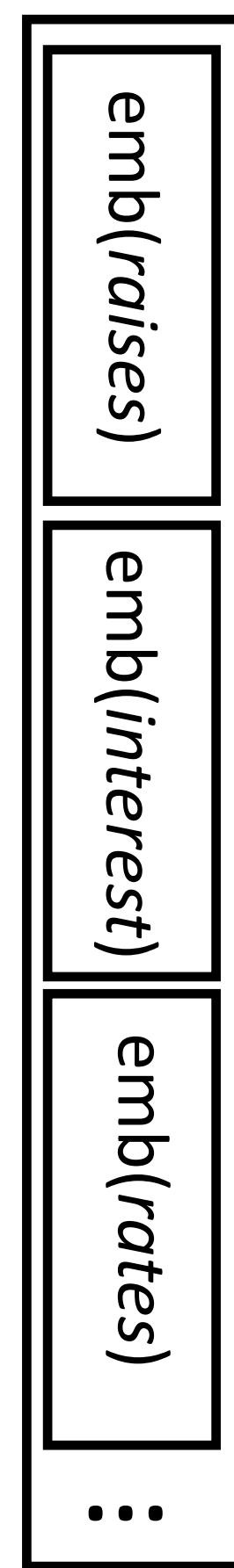
~1000 features here – smaller feature vector  
than in sparse models, but every feature fires on  
every example

curr word

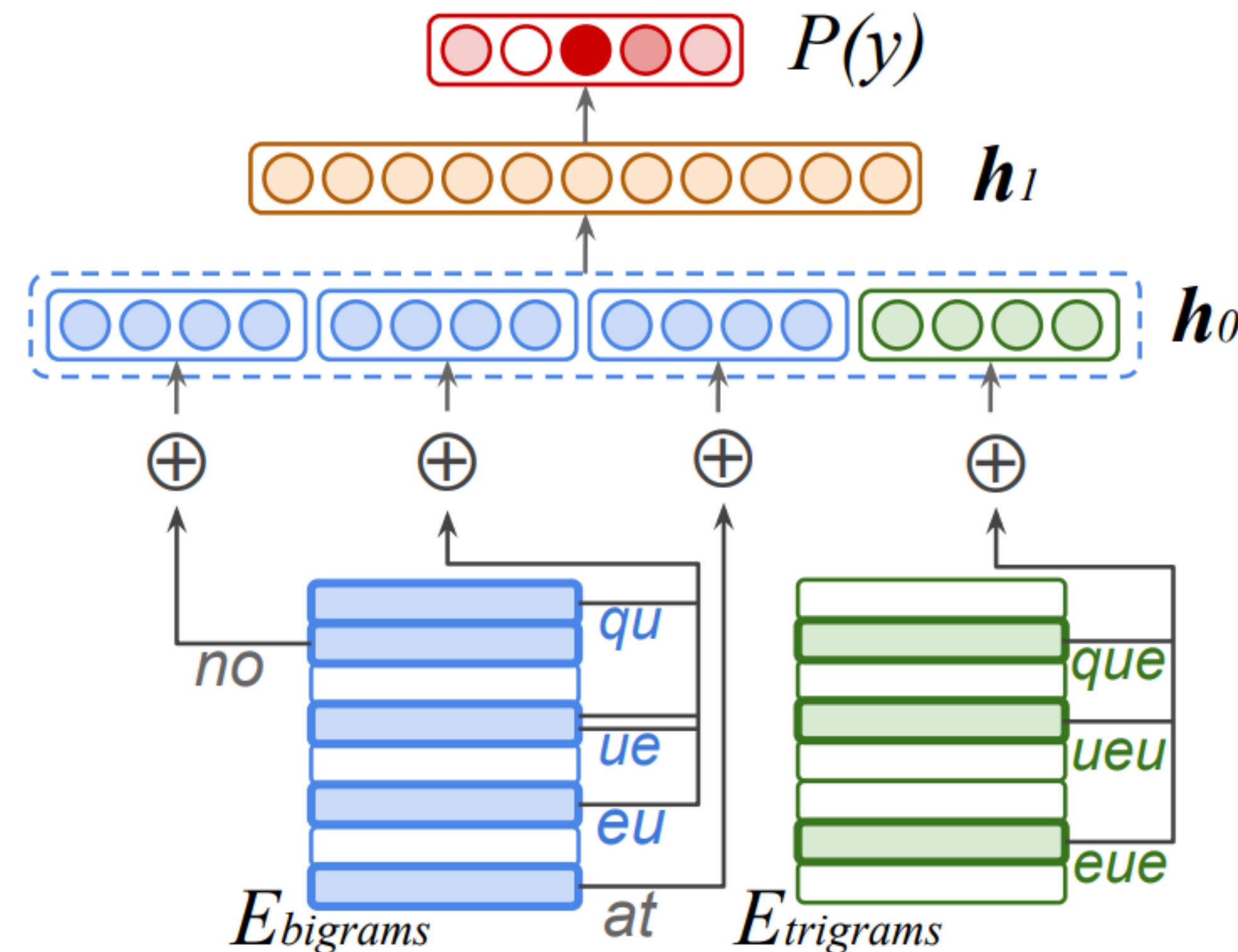
- ▶ Weight matrix learns position-dependent  
processing of the words

other words, feats, etc.

$$f(x)$$



# NLP with Feedforward Networks



- ▶ Hidden layer mixes these different signals and learns feature conjunctions

There was no queue at the ...

# NLP with Feedforward Networks

---

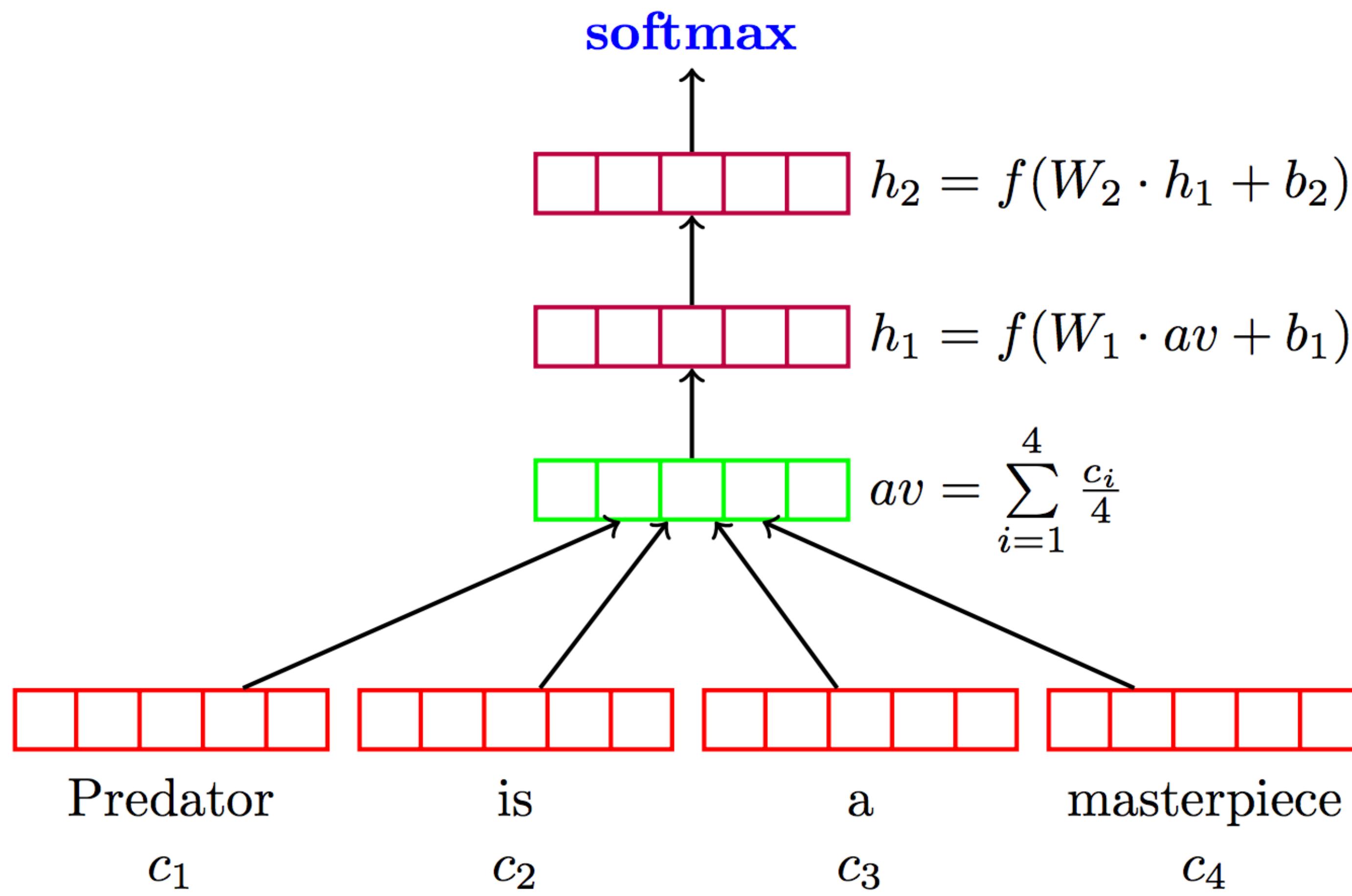
- ▶ Multilingual tagging results:

| <b>Model</b>          | <b>Acc.</b> | <b>Wts.</b> | <b>MB</b> | <b>Ops.</b> |
|-----------------------|-------------|-------------|-----------|-------------|
| 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

# Sentiment Analysis

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



Iyyer et al. (2015)

# Sentiment Analysis

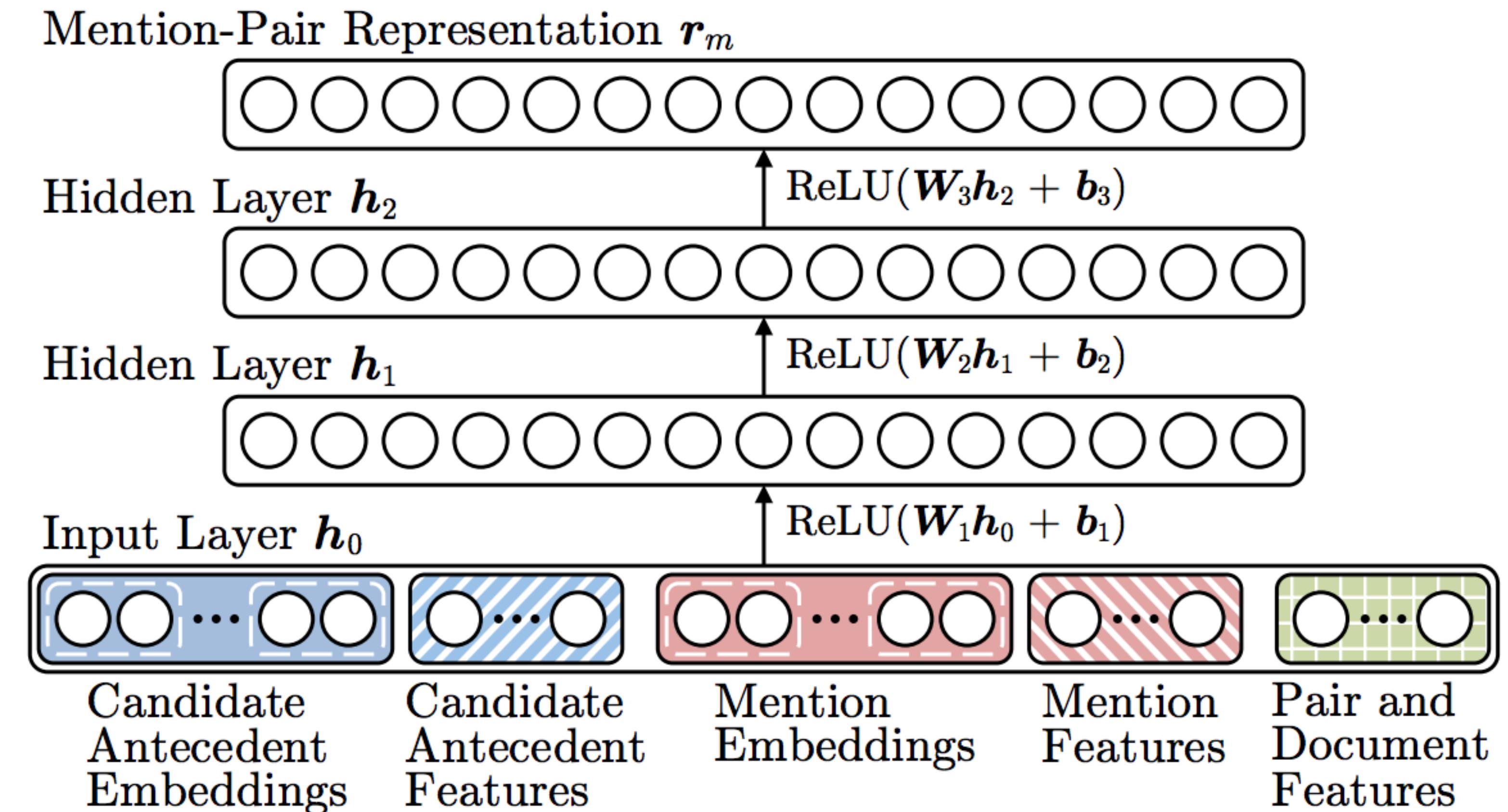
|                             | Model     | RT          | SST<br>fine | SST<br>bin  | IMDB        | Time<br>(s) |                            |
|-----------------------------|-----------|-------------|-------------|-------------|-------------|-------------|----------------------------|
| Bag-of-words                | DAN-ROOT  | —           | 46.9        | 85.7        | —           | 31          | Iyyer et al. (2015)        |
|                             | 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          |                            |
| Tree RNNs /<br>CNNs / LSTMs | NBOW      | 79.0        | 43.6        | 83.6        | 89.0        | 91          | Wang and<br>Manning (2012) |
|                             | BiNB      | —           | 41.9        | 83.1        | —           | —           |                            |
|                             | NBSVM-bi  | 79.4        | —           | —           | 91.2        | —           |                            |
|                             | RecNN*    | 77.7        | 43.2        | 82.4        | —           | —           |                            |
| Tree RNNs /<br>CNNs / LSTMs | RecNTN*   | —           | 45.7        | 85.4        | —           | —           | Kim (2014)                 |
|                             | DRecNN    | —           | 49.8        | 86.6        | —           | 431         |                            |
|                             | TreeLSTM  | —           | <b>50.6</b> | 86.9        | —           | —           |                            |
|                             | DCNN*     | —           | 48.5        | 86.9        | 89.4        | —           |                            |
|                             | PVEC*     | —           | 48.7        | 87.8        | <b>92.6</b> | —           |                            |
|                             | CNN-MC    | <b>81.1</b> | 47.4        | <b>88.1</b> | —           | 2,452       |                            |
|                             | WRRBMs*   | —           | —           | —           | 89.2        | —           |                            |

# Coreference Resolution

- Feedforward networks identify coreference arcs

*President Obama signed...*  
?

*He later gave a speech...*



# Implementation Details

# Computation Graphs

---

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

$y = x * x \longrightarrow (y, dy) = (x * x, 2 * x * dx)$   
codegen

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

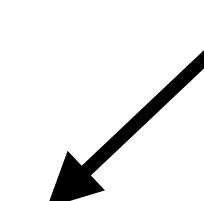
```
class FFNN(nn.Module):  
    def __init__(self, inp, hid, out):  
        super(FFNN, self).__init__()  
        self.v = nn.Linear(inp, hid)  
        self.g = nn.Tanh()  
        self.w = nn.Linear(hid, out)  
        self.softmax = nn.Softmax(dim=0)  
  
    def forward(self, x):  
        return self.softmax(self.w(self.g(self.v(x)))))
```

# Computation Graphs in Pytorch

---

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

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