

Lecture 6: Neural Networks

Alan Ritter

(many slides from Greg Durrett)

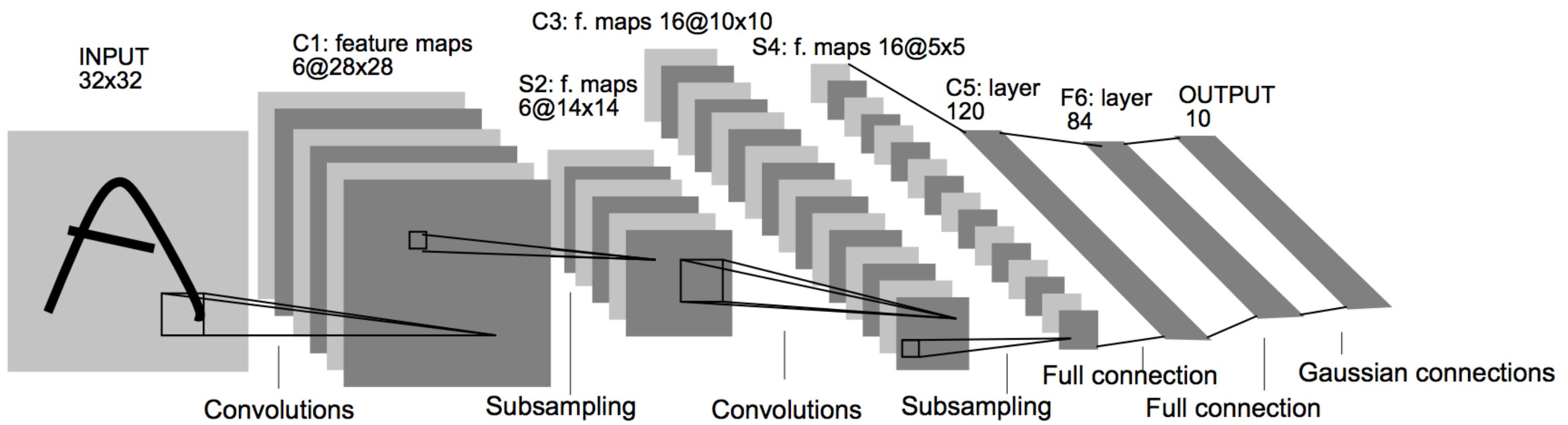
This Lecture

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

History: NN “dark ages”

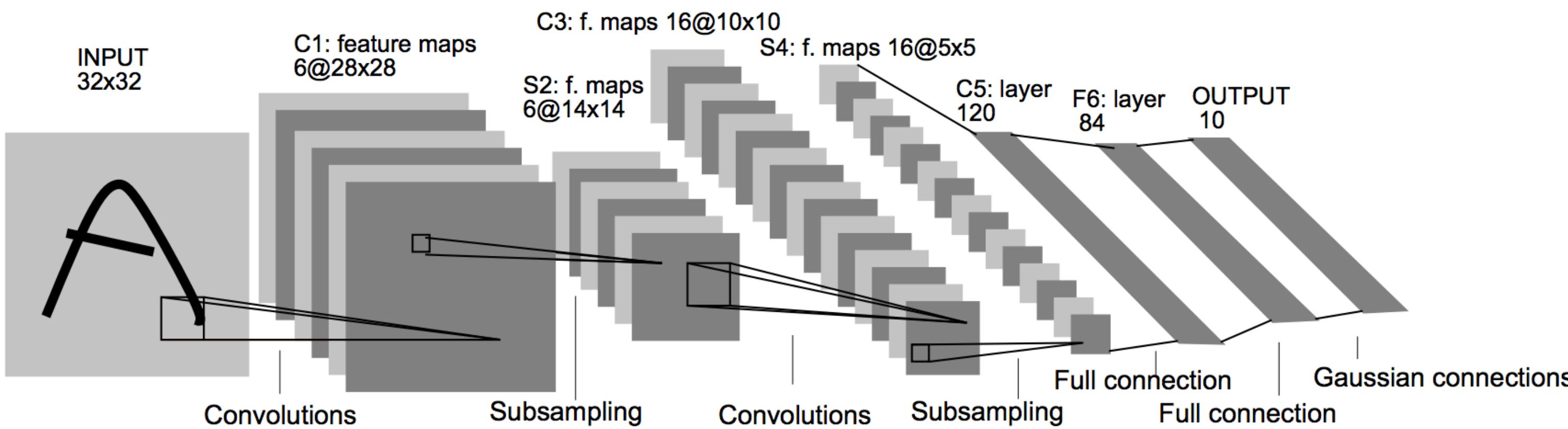
History: NN “dark ages”

- ▶ Convnets: applied to MNIST by LeCun in 1998

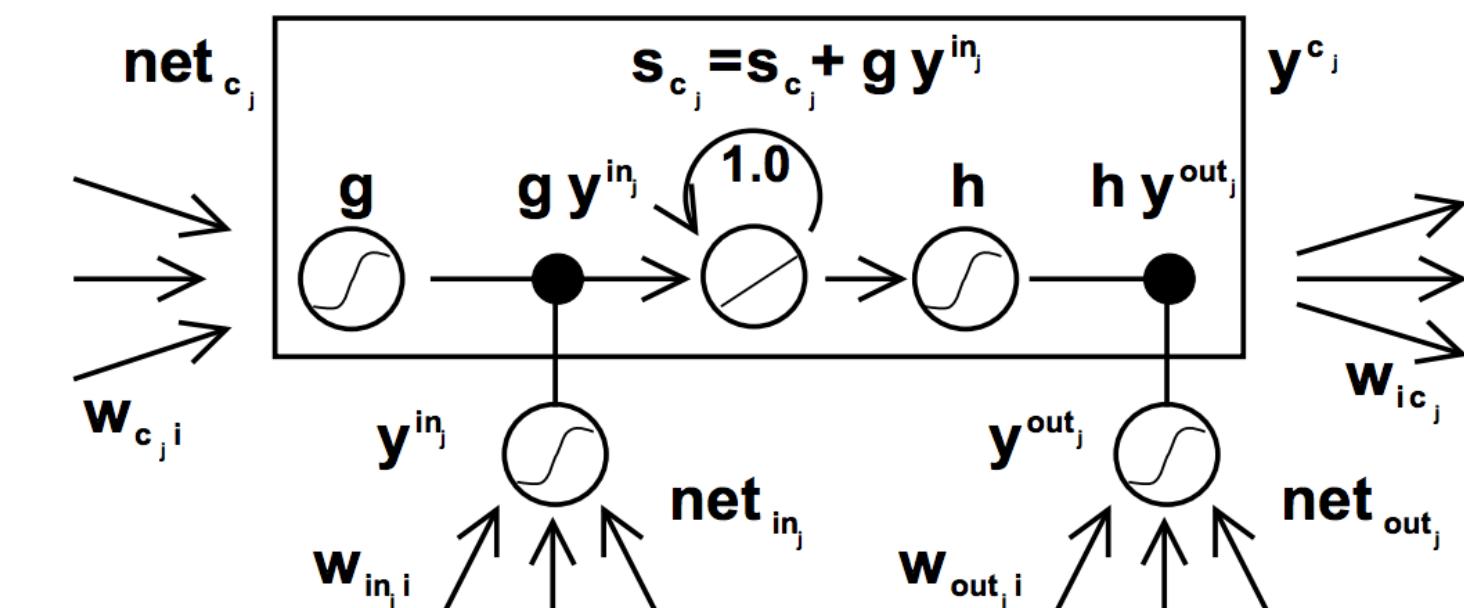


History: NN “dark ages”

- ▶ Convnets: applied to MNIST by LeCun in 1998

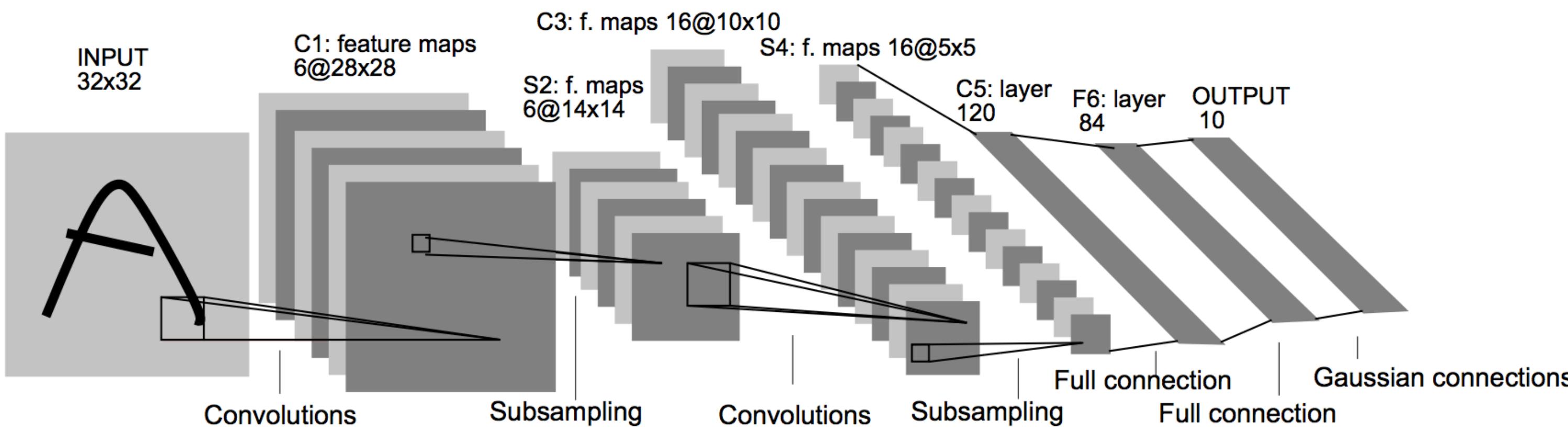


- ▶ LSTMs: Hochreiter and Schmidhuber (1997)

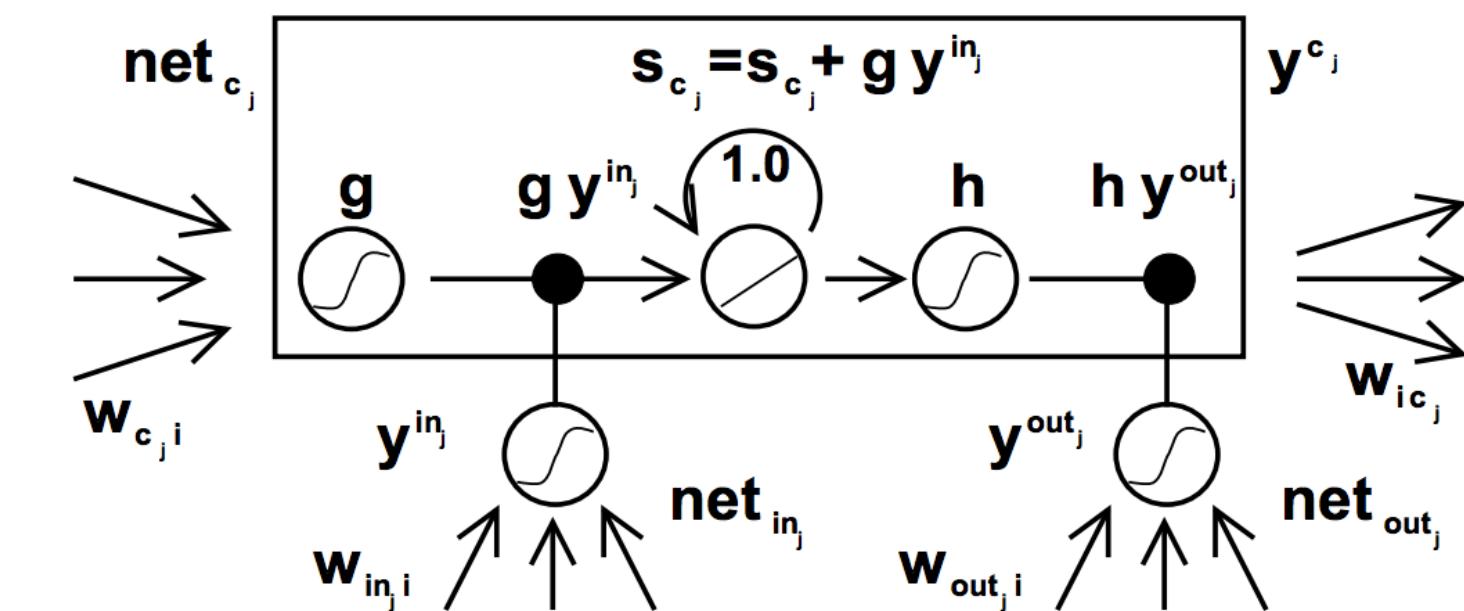


History: NN “dark ages”

- ▶ Convnets: applied to MNIST by LeCun in 1998



- ▶ LSTMs: Hochreiter and Schmidhuber (1997)

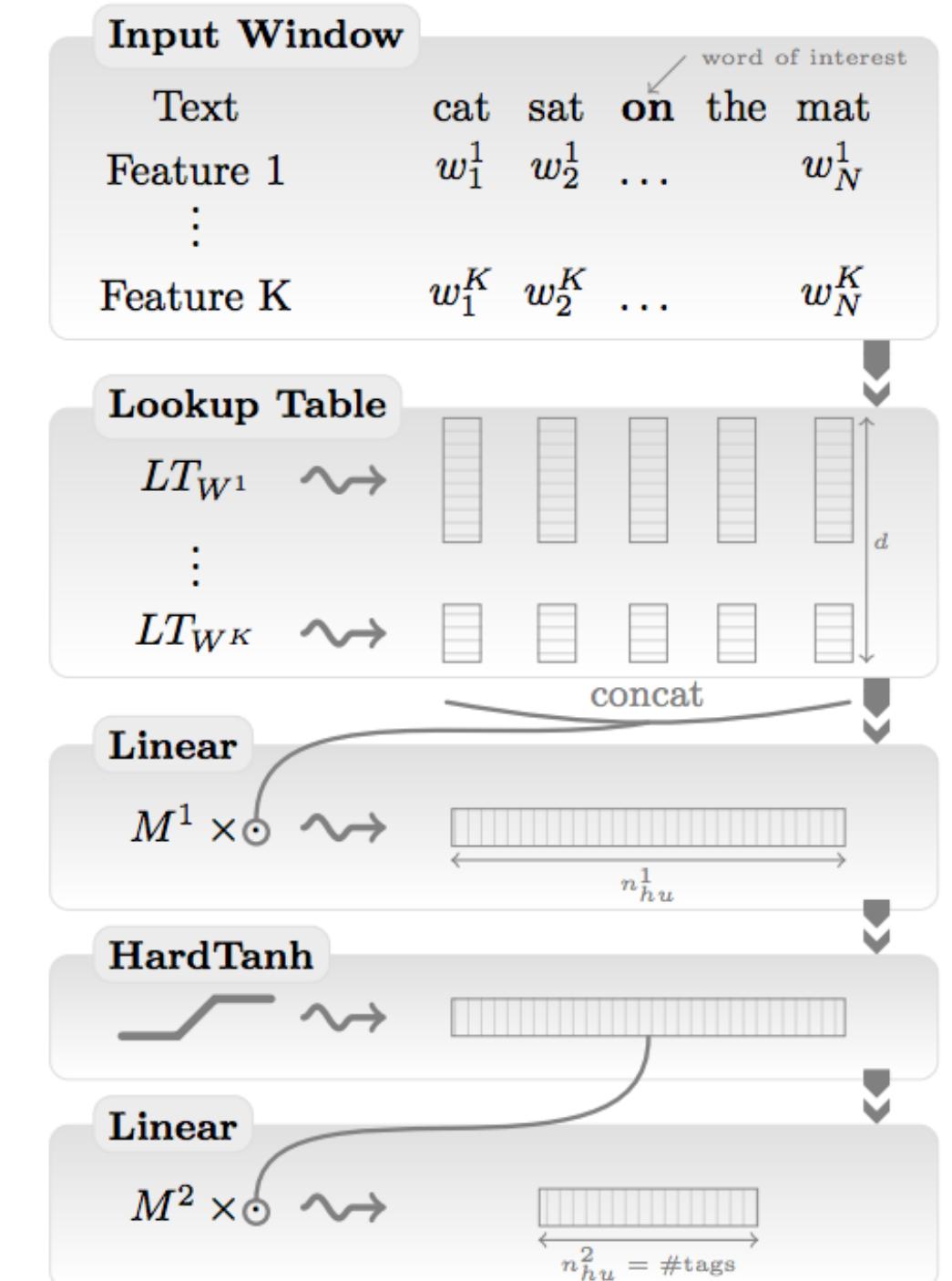


- ▶ Henderson (2003): neural shift-reduce parser, not SOTA

2008-2013: A glimmer of light...

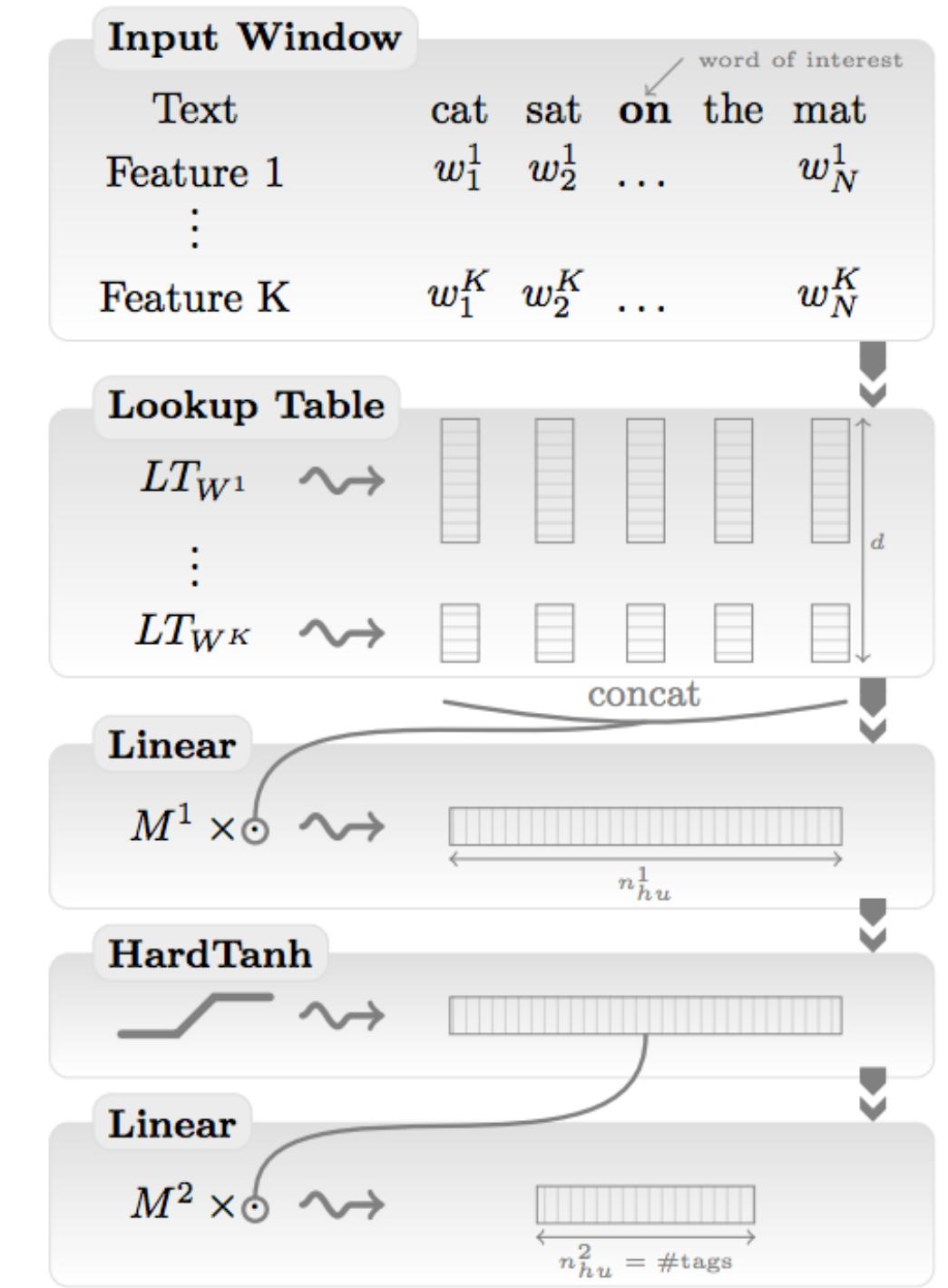
2008-2013: A glimmer of light...

- ▶ 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



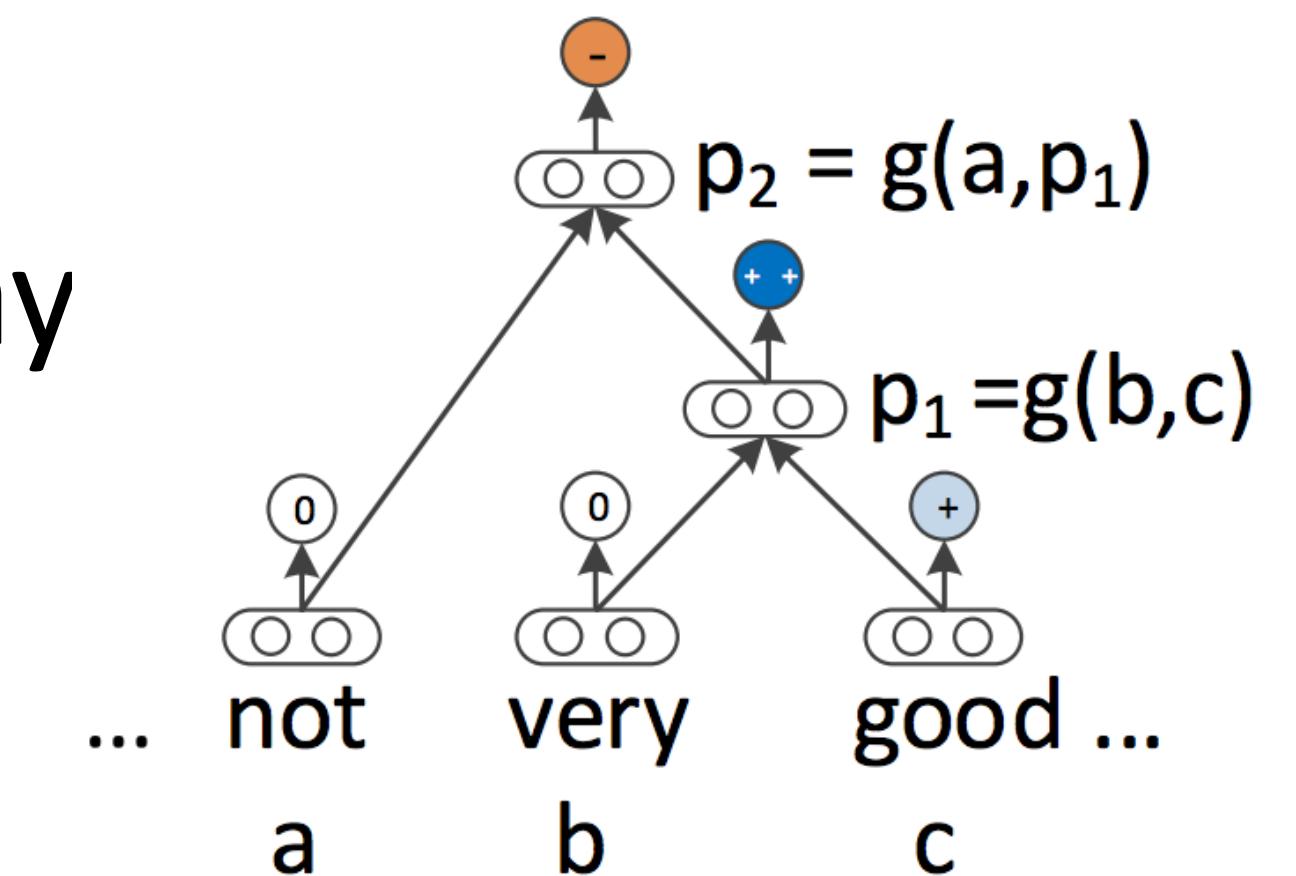
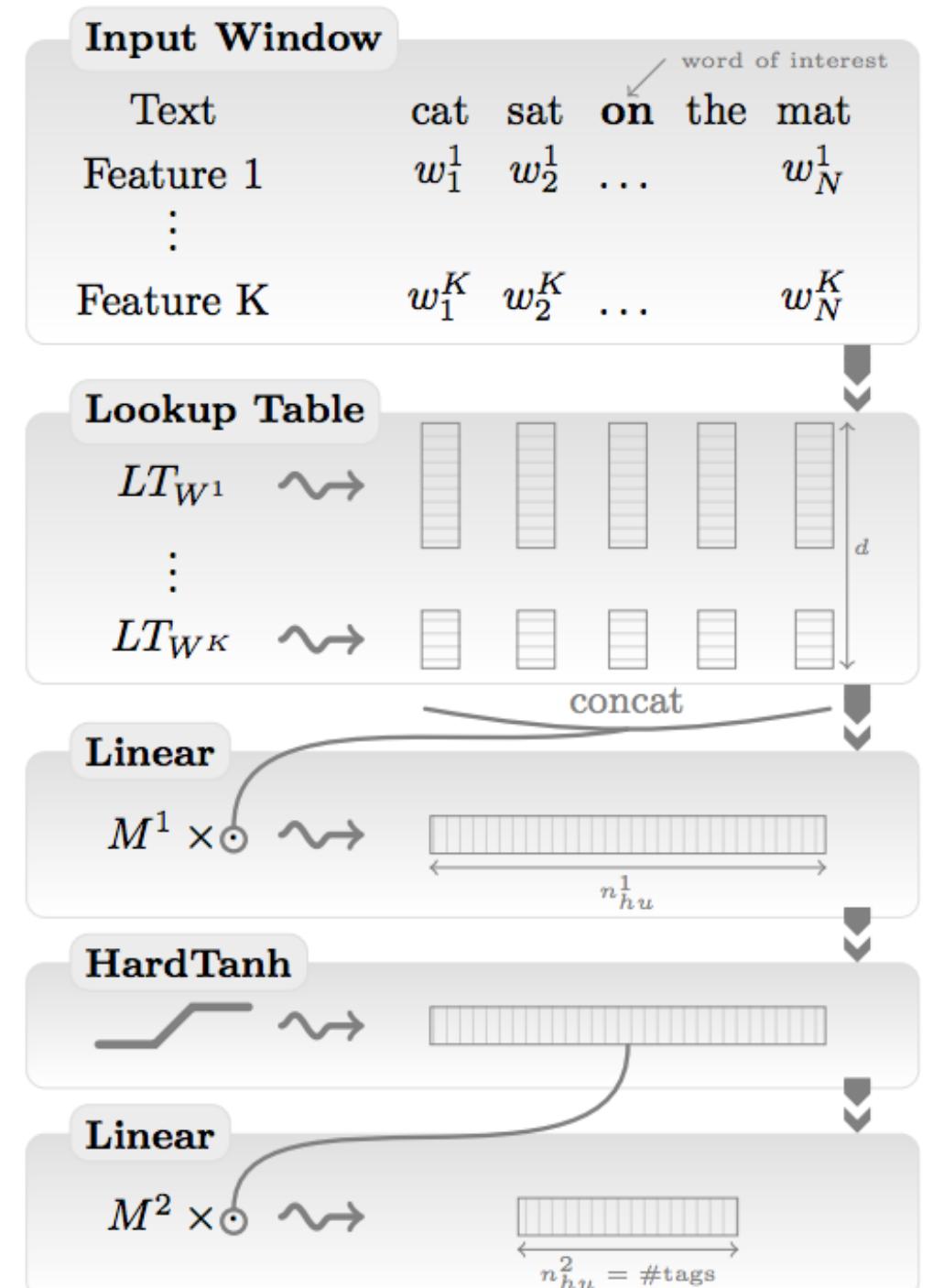
2008-2013: A glimmer of light...

- ▶ 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



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

2014: Stuff starts working

- ▶ Kim (2014) + Kalchbrenner et al. (2014): sentence classification / sentiment (convnets work for NLP?)

2014: Stuff starts working

- ▶ Kim (2014) + Kalchbrenner et al. (2014): sentence classification / sentiment (convnets work for NLP?)
- ▶ Sutskever et al. + Bahdanau et al.: seq2seq for neural MT (LSTMs work for NLP?)

2014: Stuff starts working

- ▶ Kim (2014) + Kalchbrenner et al. (2014): sentence classification / sentiment (convnets work for NLP?)
- ▶ Sutskever et al. + Bahdanau et al.: seq2seq for neural MT (LSTMs work for NLP?)
- ▶ Chen and Manning transition-based dependency parser (even feedforward networks work well for NLP?)

2014: Stuff starts working

- ▶ Kim (2014) + Kalchbrenner et al. (2014): sentence classification / sentiment (convnets work for NLP?)
- ▶ Sutskever et al. + Bahdanau et al.: seq2seq for neural MT (LSTMs work for NLP?)
- ▶ Chen and Manning 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?

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)

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

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

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

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

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

Neural Networks

- ▶ Linear classification: $\operatorname{argmax}_y w^\top f(x, y)$
- ▶ How can we do nonlinear classification? Kernels are too slow...

Neural Networks

- ▶ 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

Neural Networks

- ▶ 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*

Neural Networks

- ▶ 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
- ▶ Output

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

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

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

- ▶ Inputs x_1, x_2

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

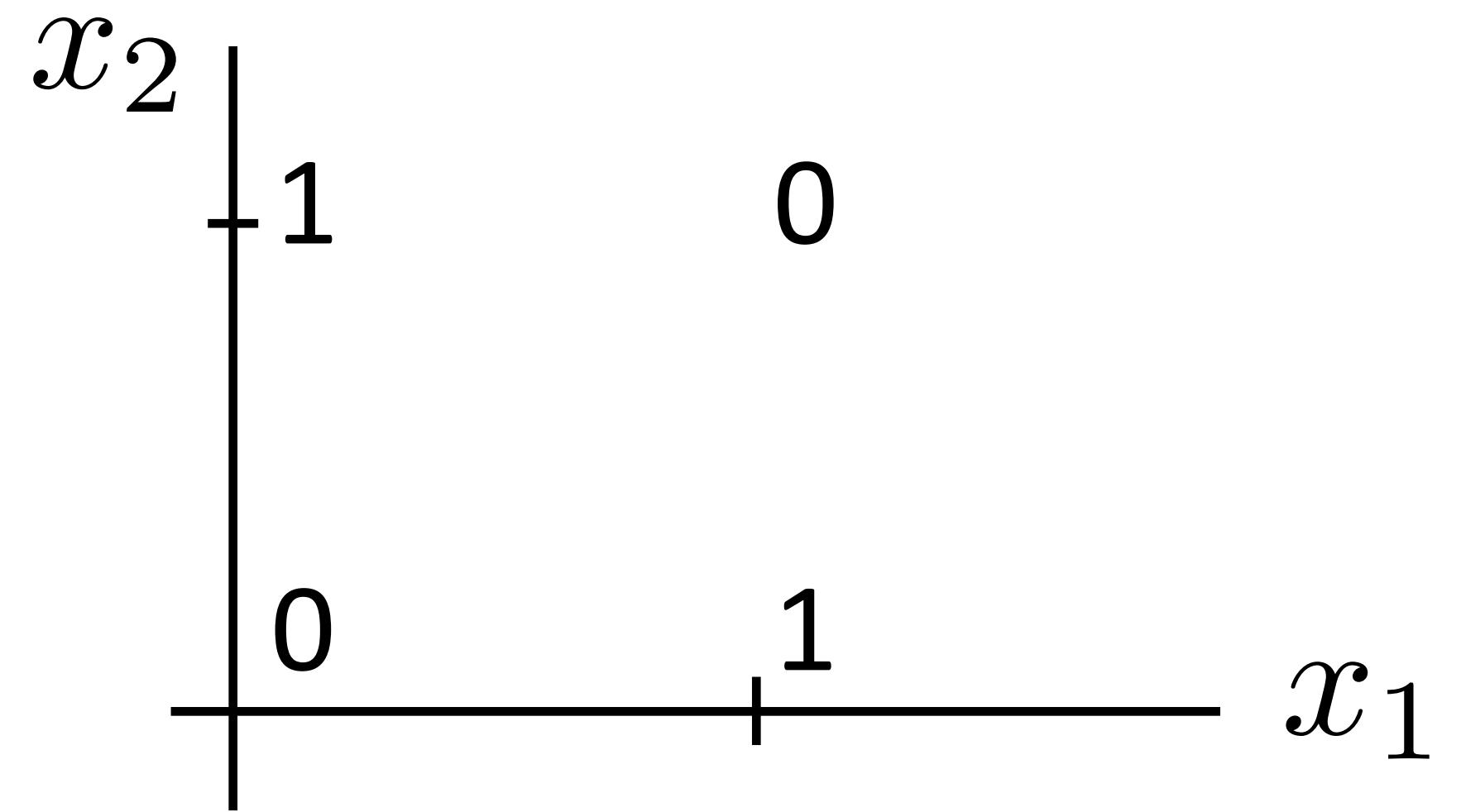
- ▶ Output y

(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

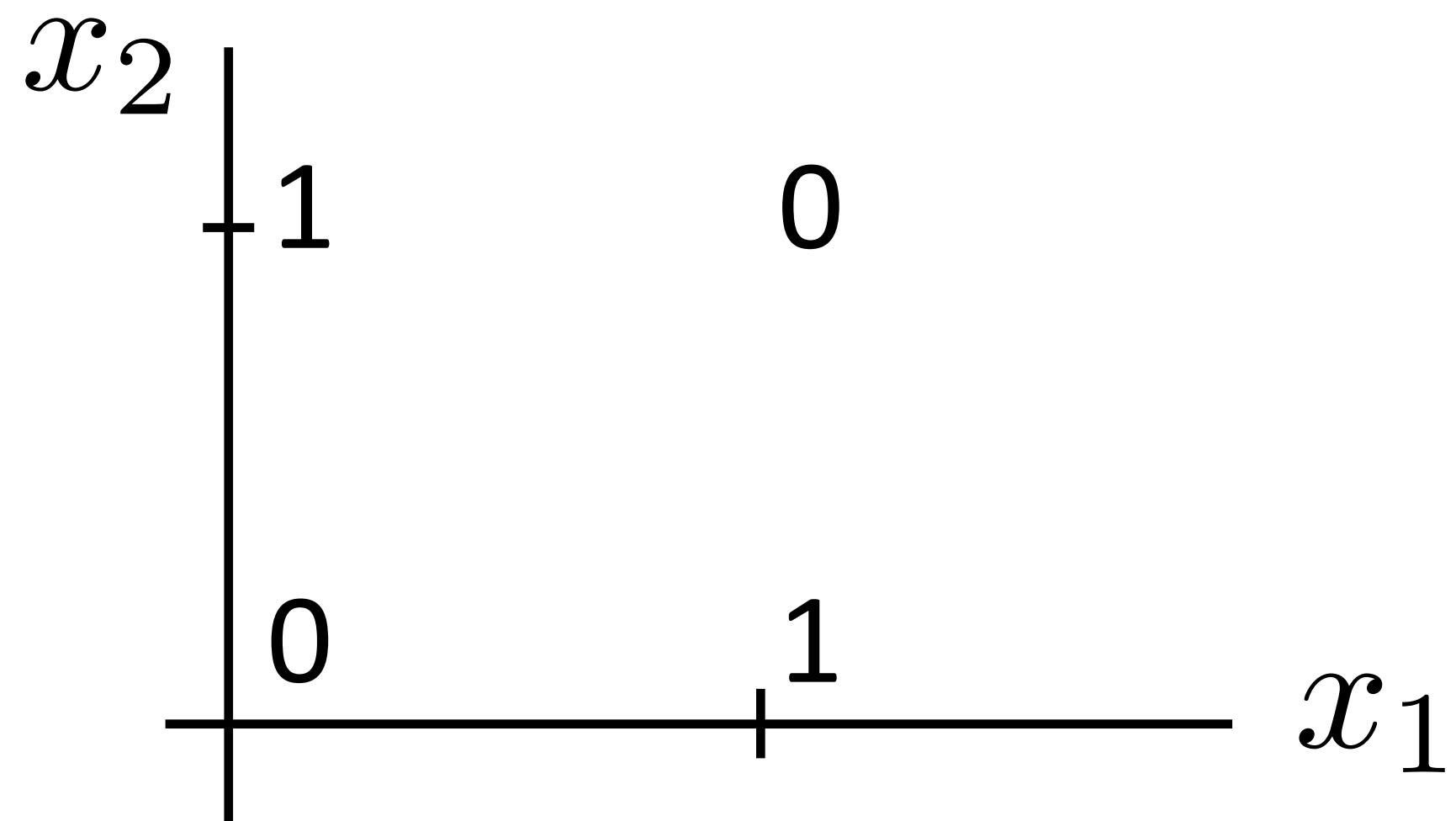
Neural Networks: XOR

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



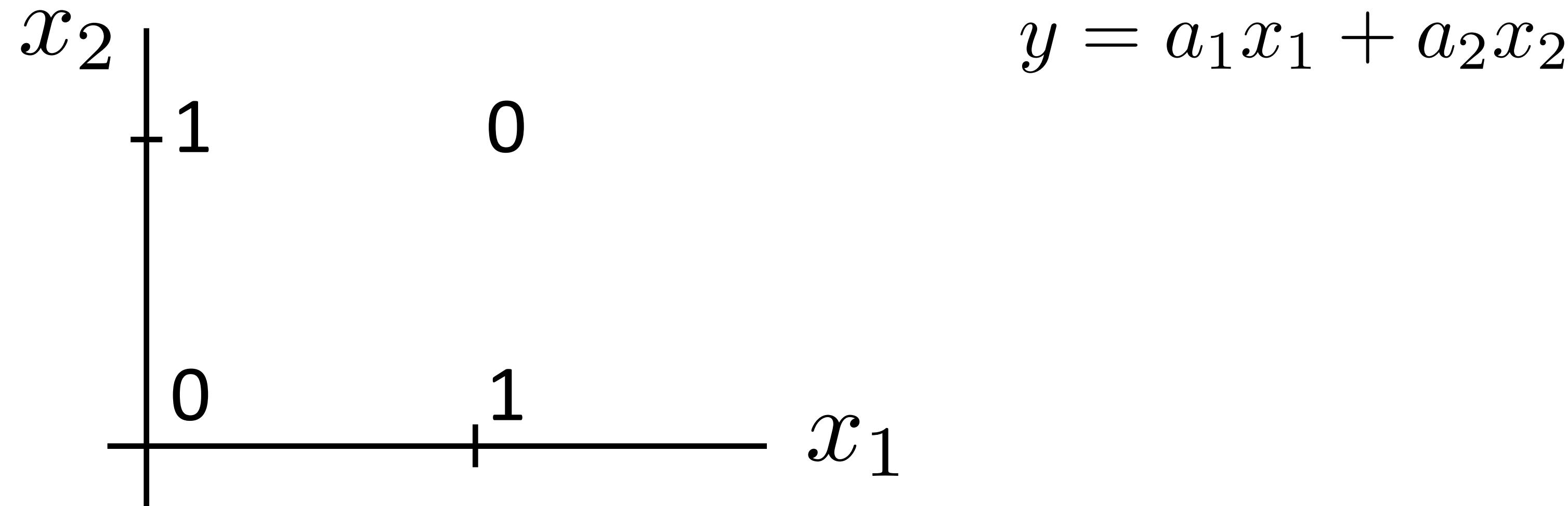
x_1	x_2	$y = x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Neural Networks: XOR



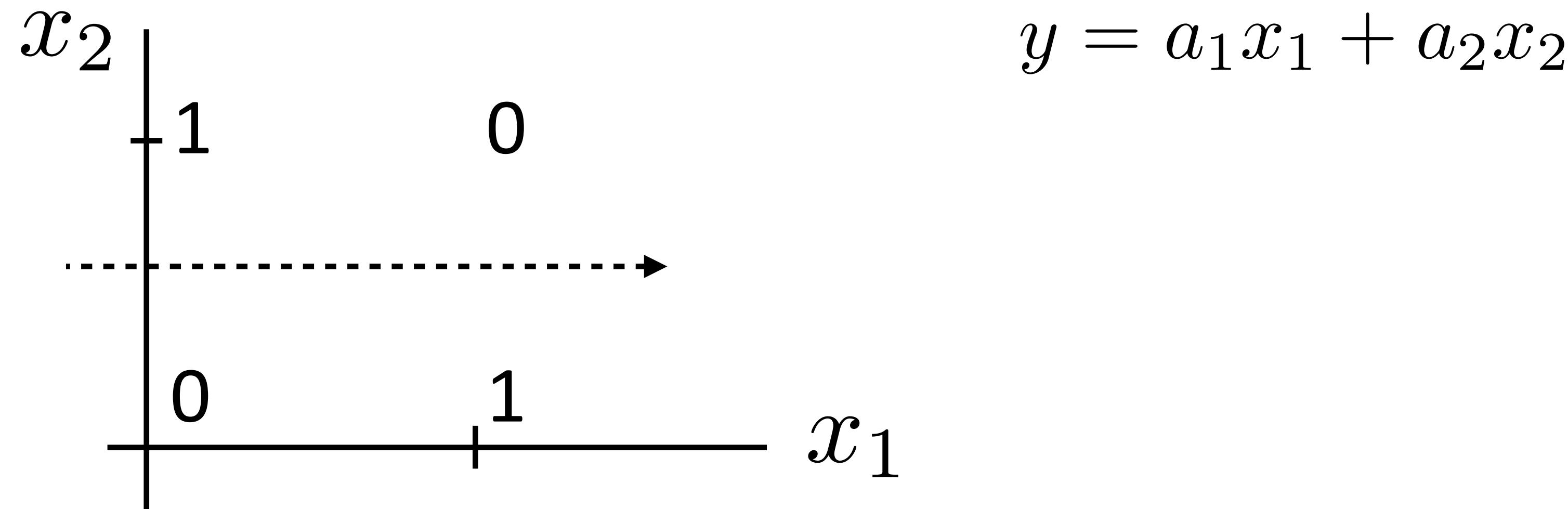
x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Neural Networks: XOR



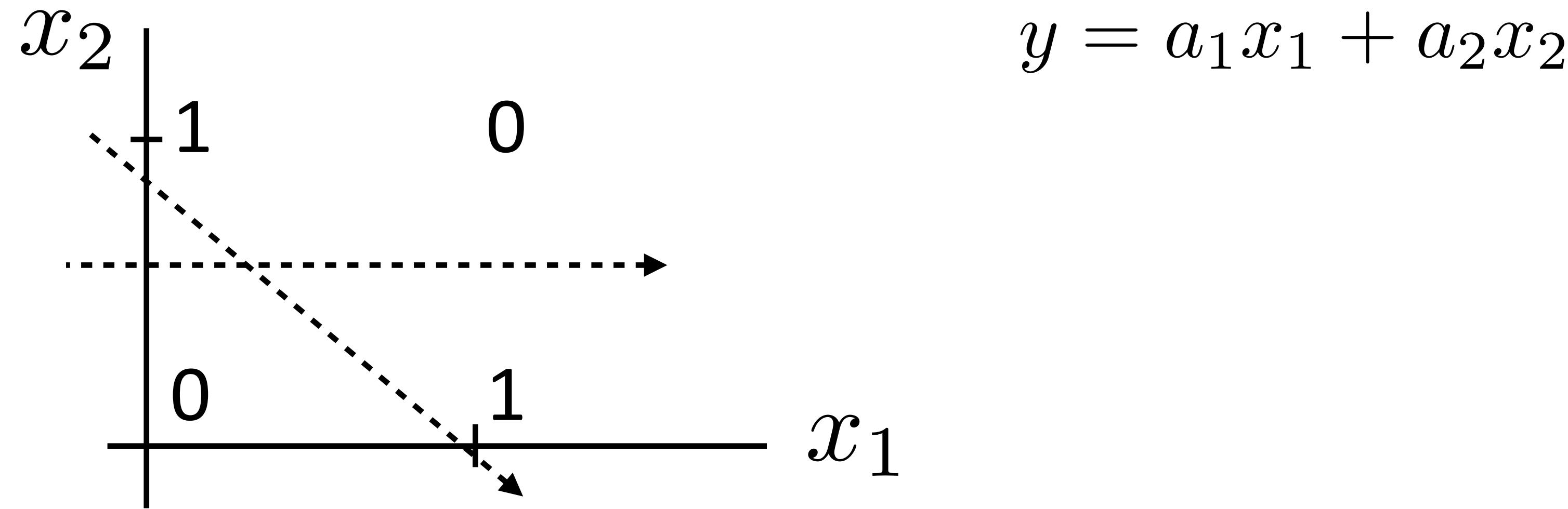
x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Neural Networks: XOR



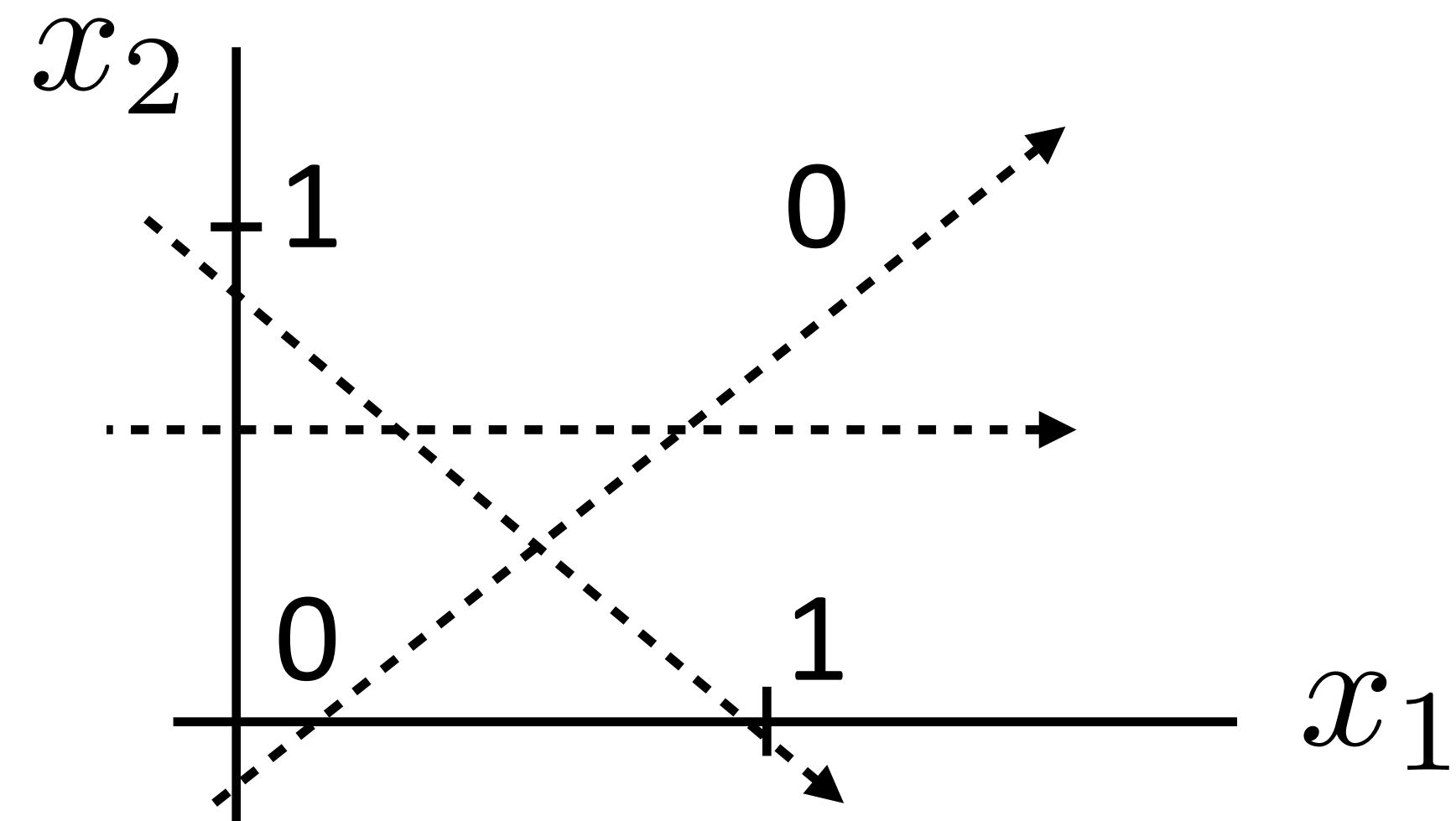
x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

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

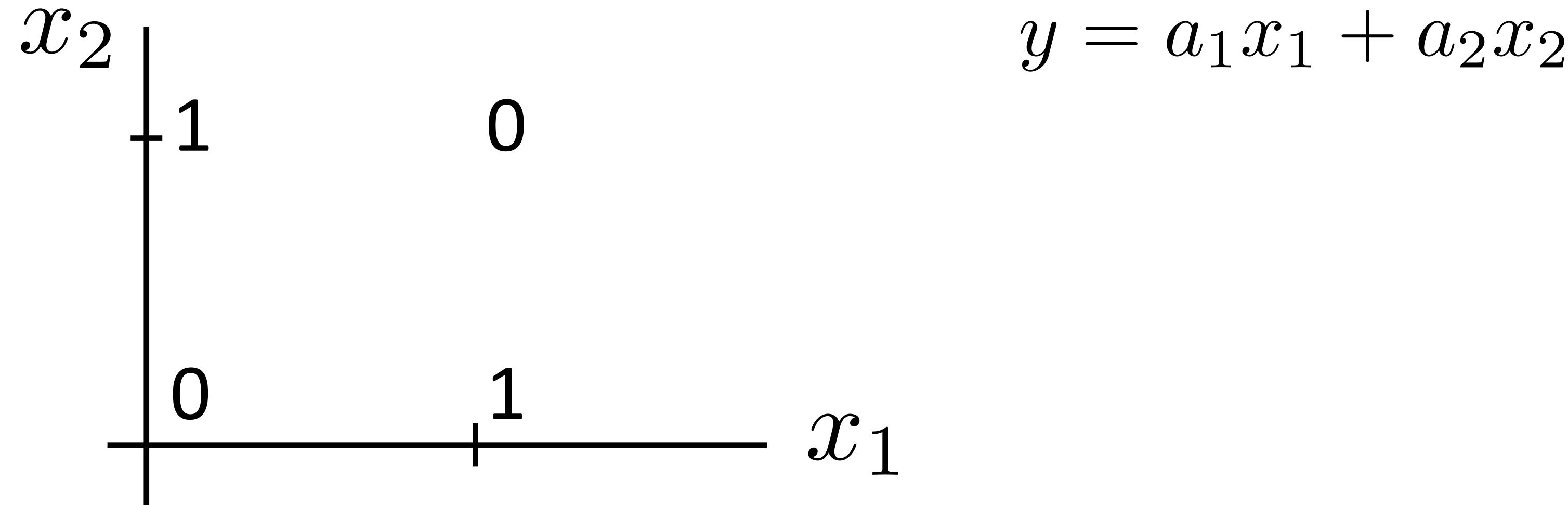
Neural Networks: XOR



$$y = a_1x_1 + a_2x_2$$

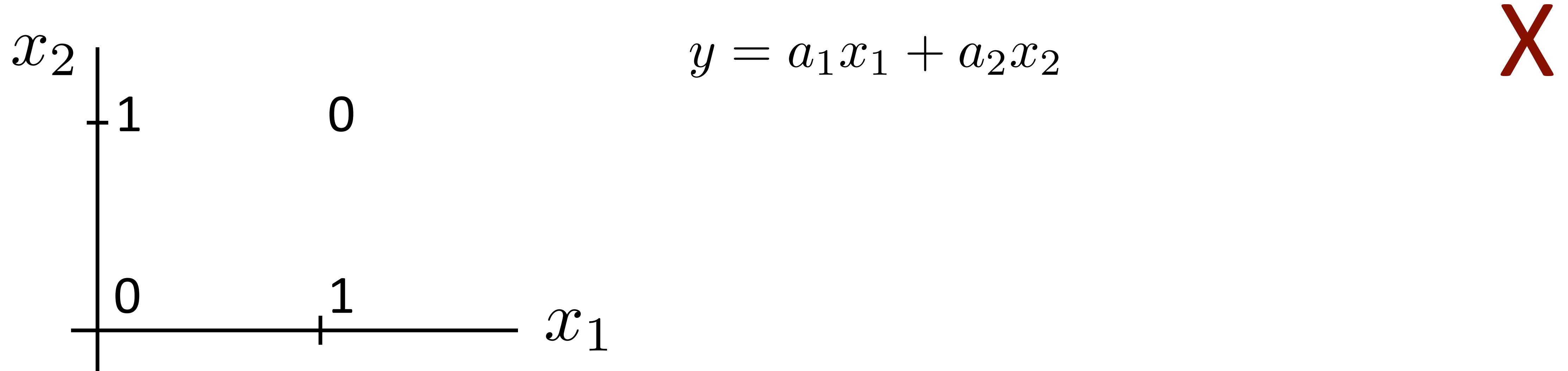
x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Neural Networks: XOR



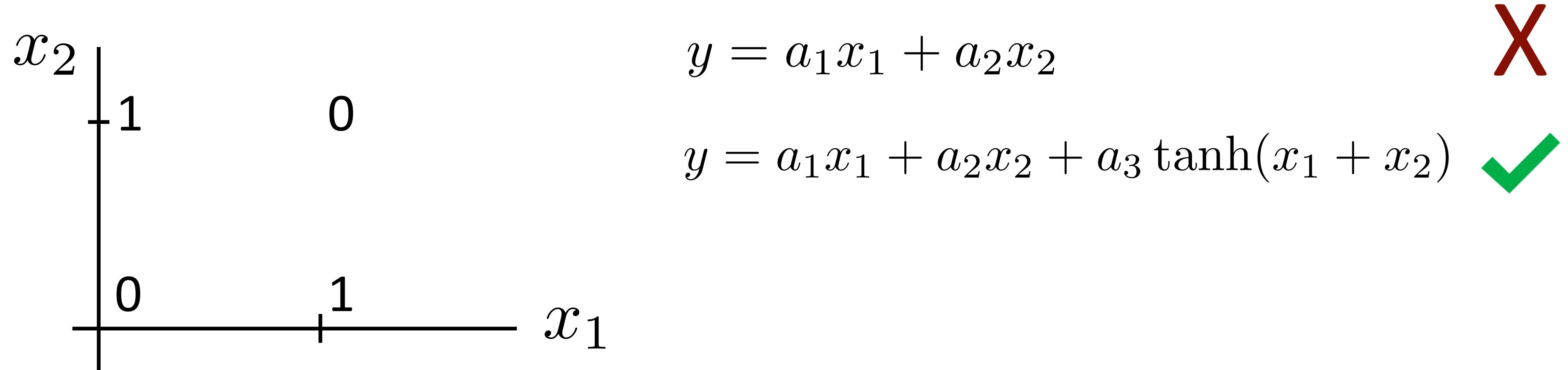
x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Neural Networks: XOR



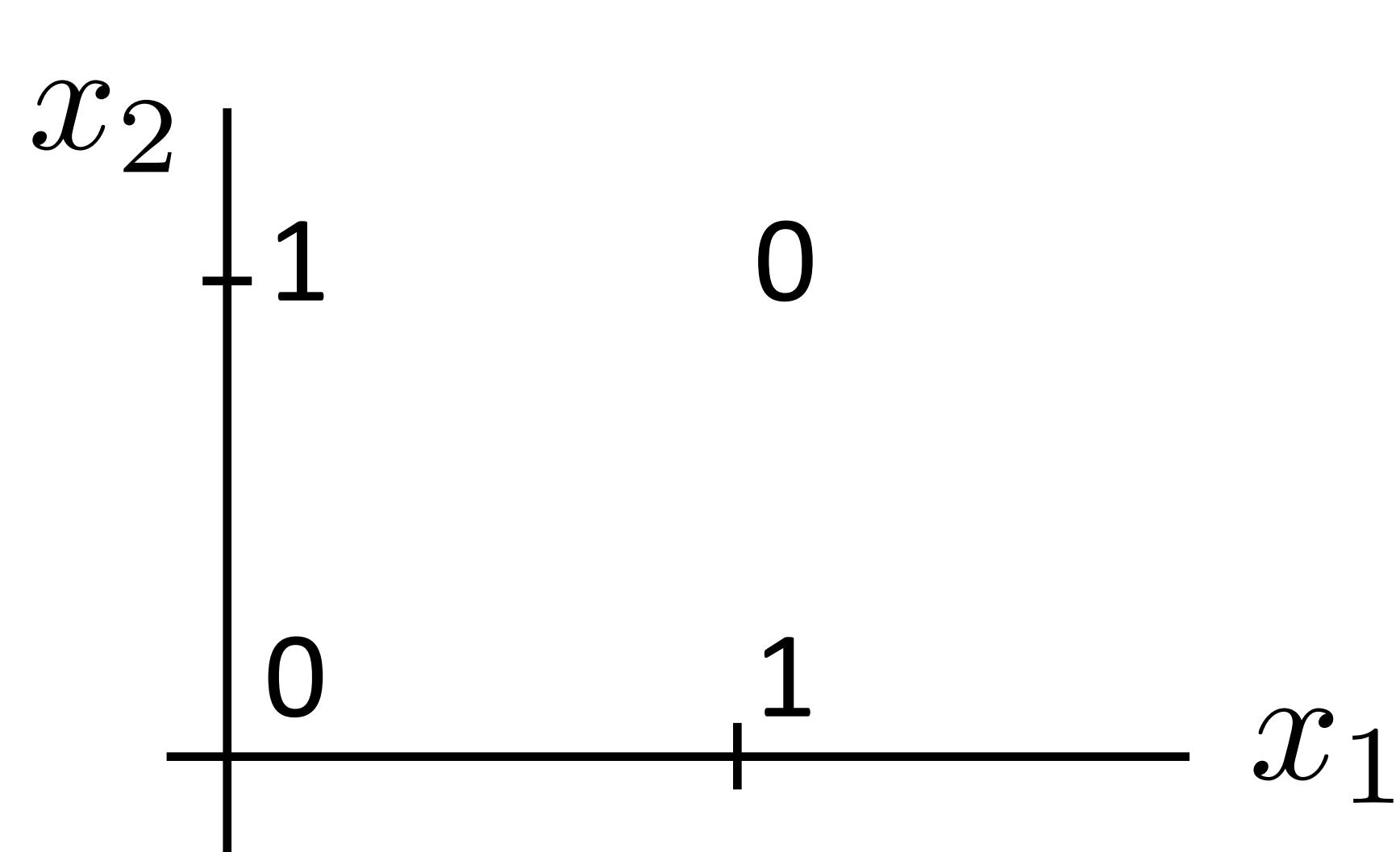
x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

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

Neural Networks: XOR



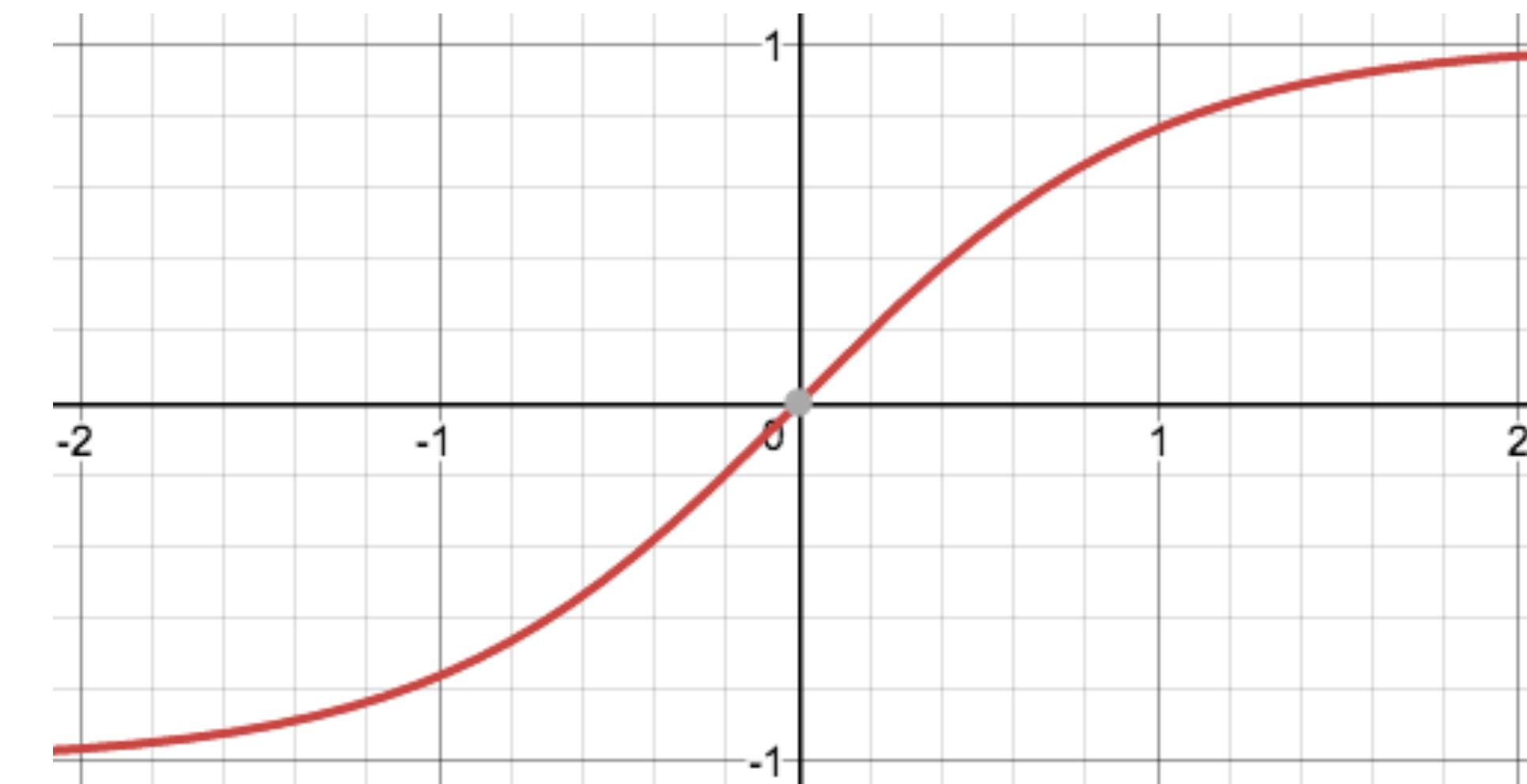
x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

$$y = a_1 x_1 + a_2 x_2$$

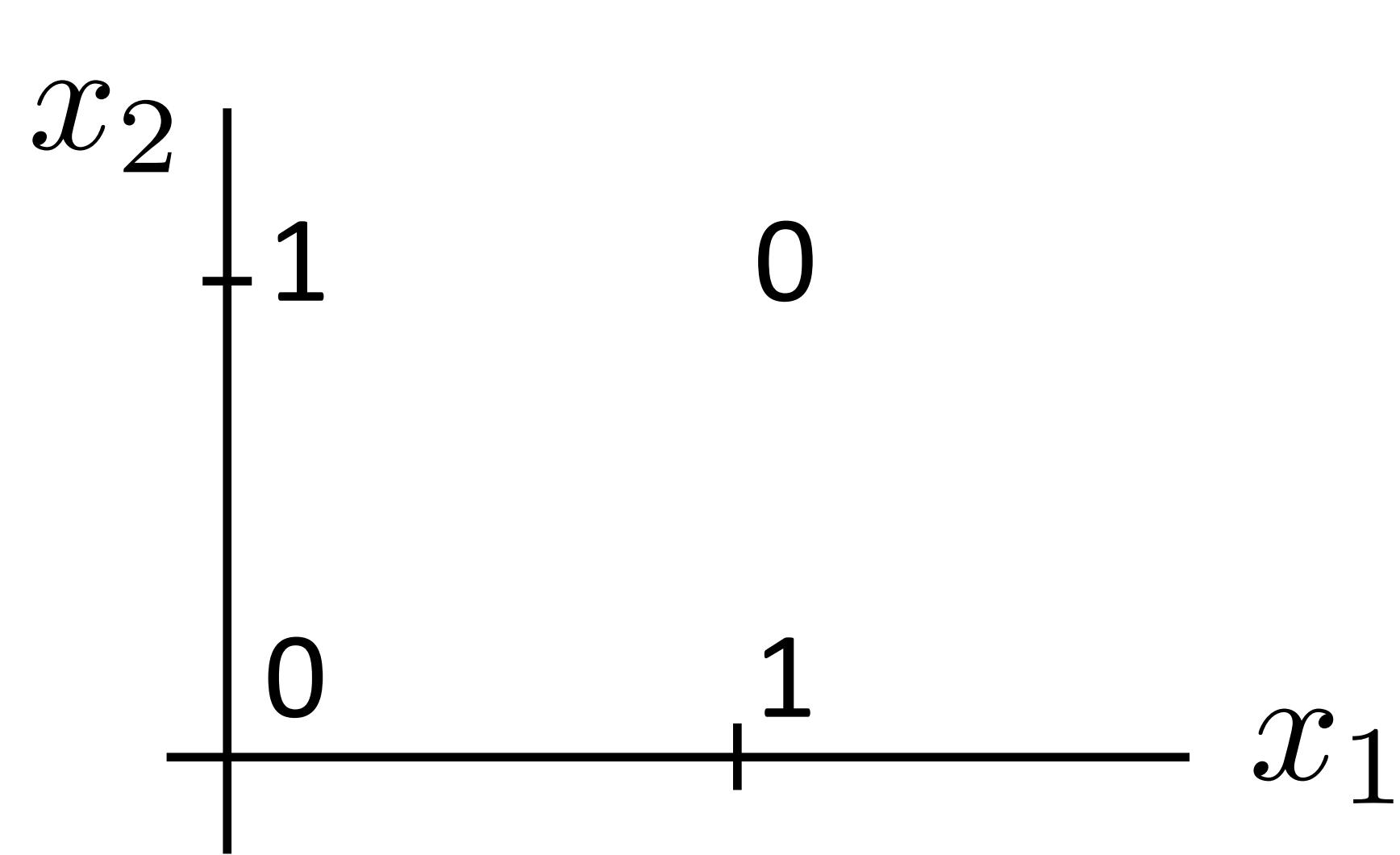
$$y = a_1 x_1 + a_2 x_2 + a_3 \tanh(x_1 + x_2)$$

“or”

X



Neural Networks: XOR



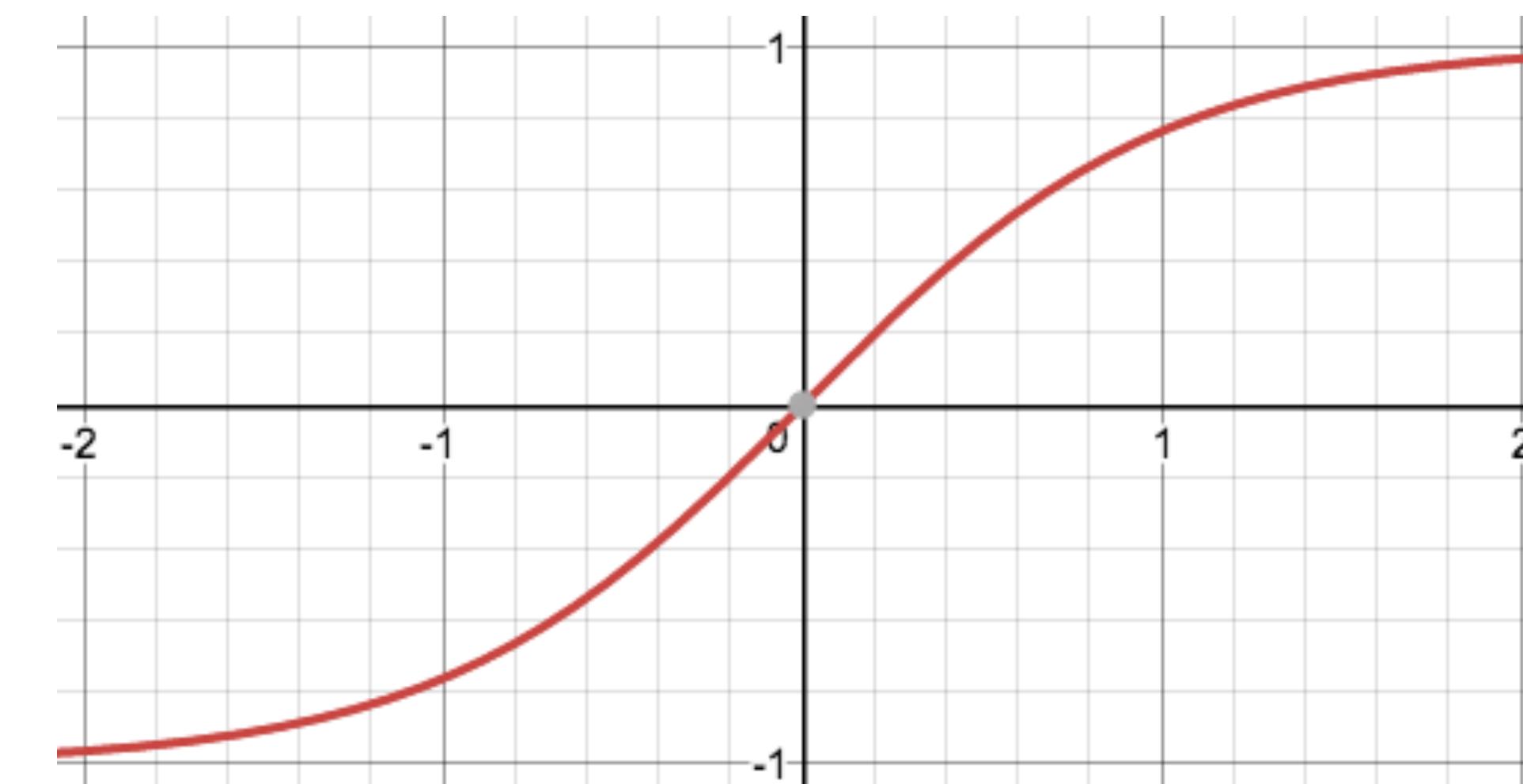
x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

$$y = a_1 x_1 + a_2 x_2$$

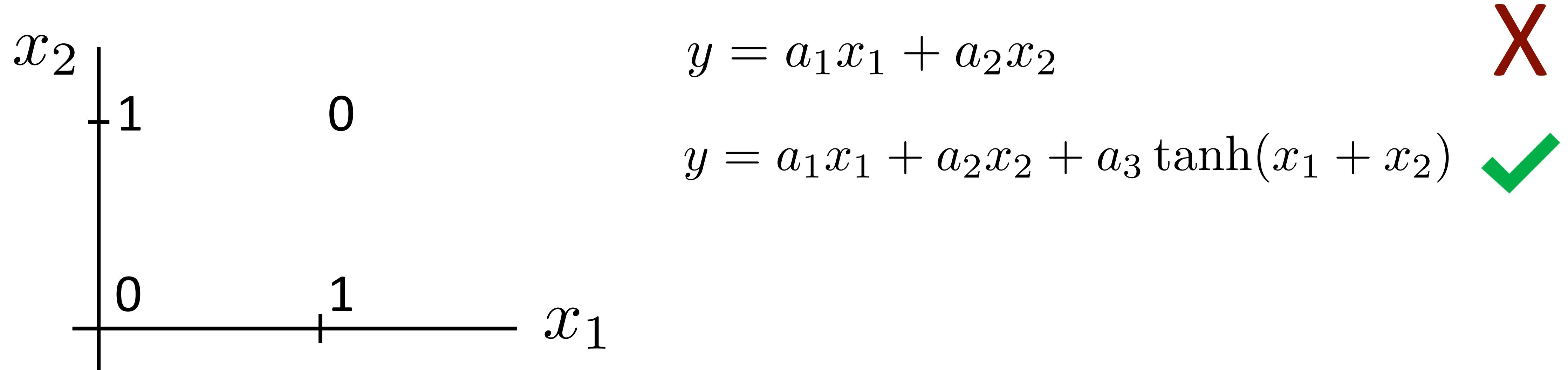
$$y = a_1 x_1 + a_2 x_2 + a_3 \tanh(x_1 + x_2)$$

“or”

(looks like action potential in neuron)

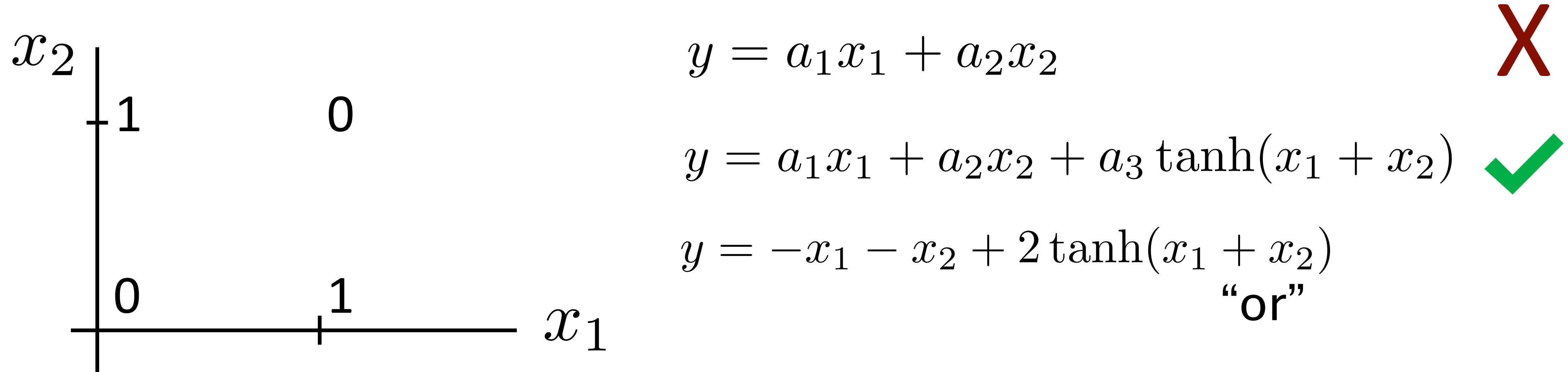


Neural Networks: XOR



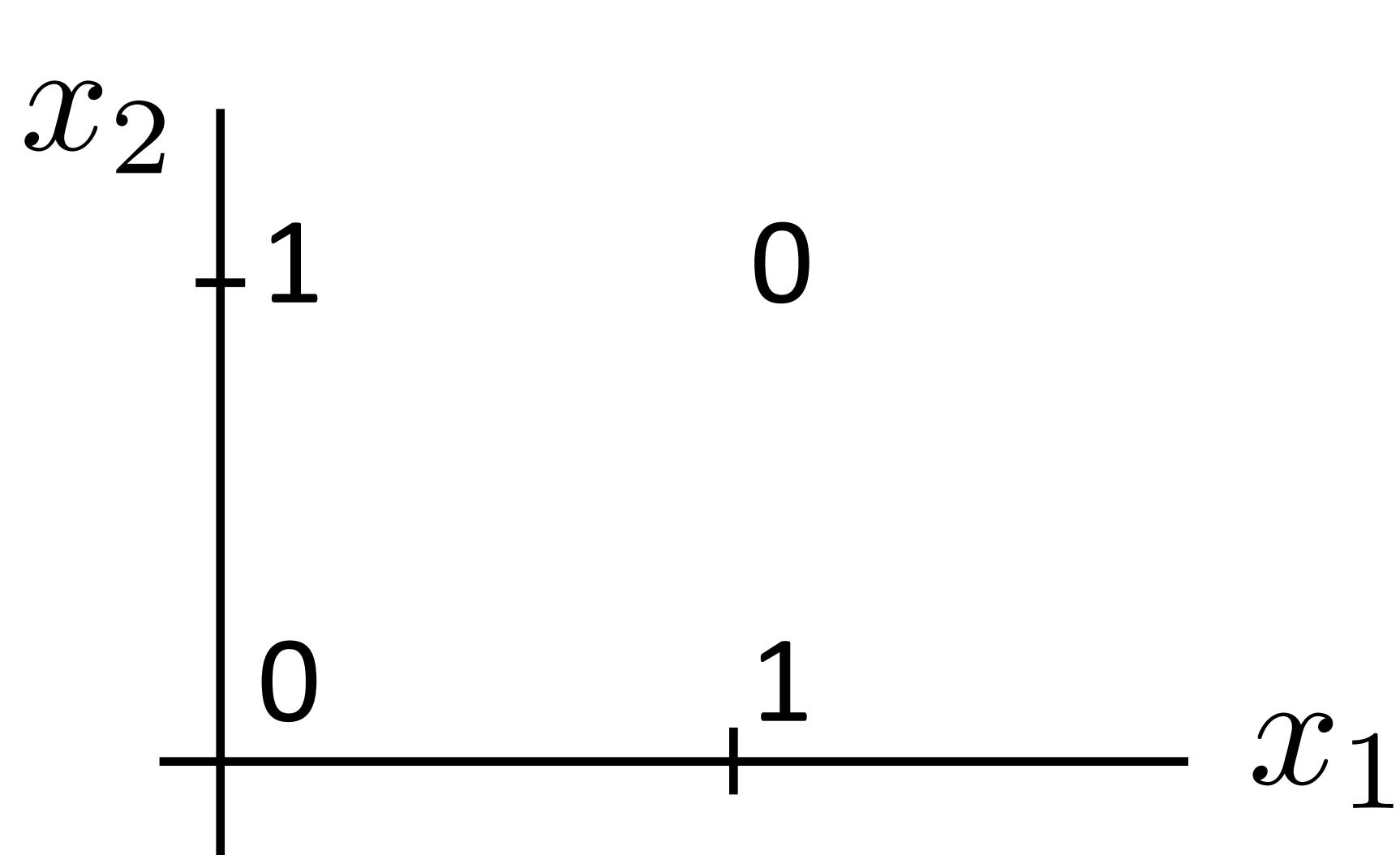
x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Neural Networks: XOR



x_1	x_2	x_1 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

$$y = a_1 x_1 + a_2 x_2$$

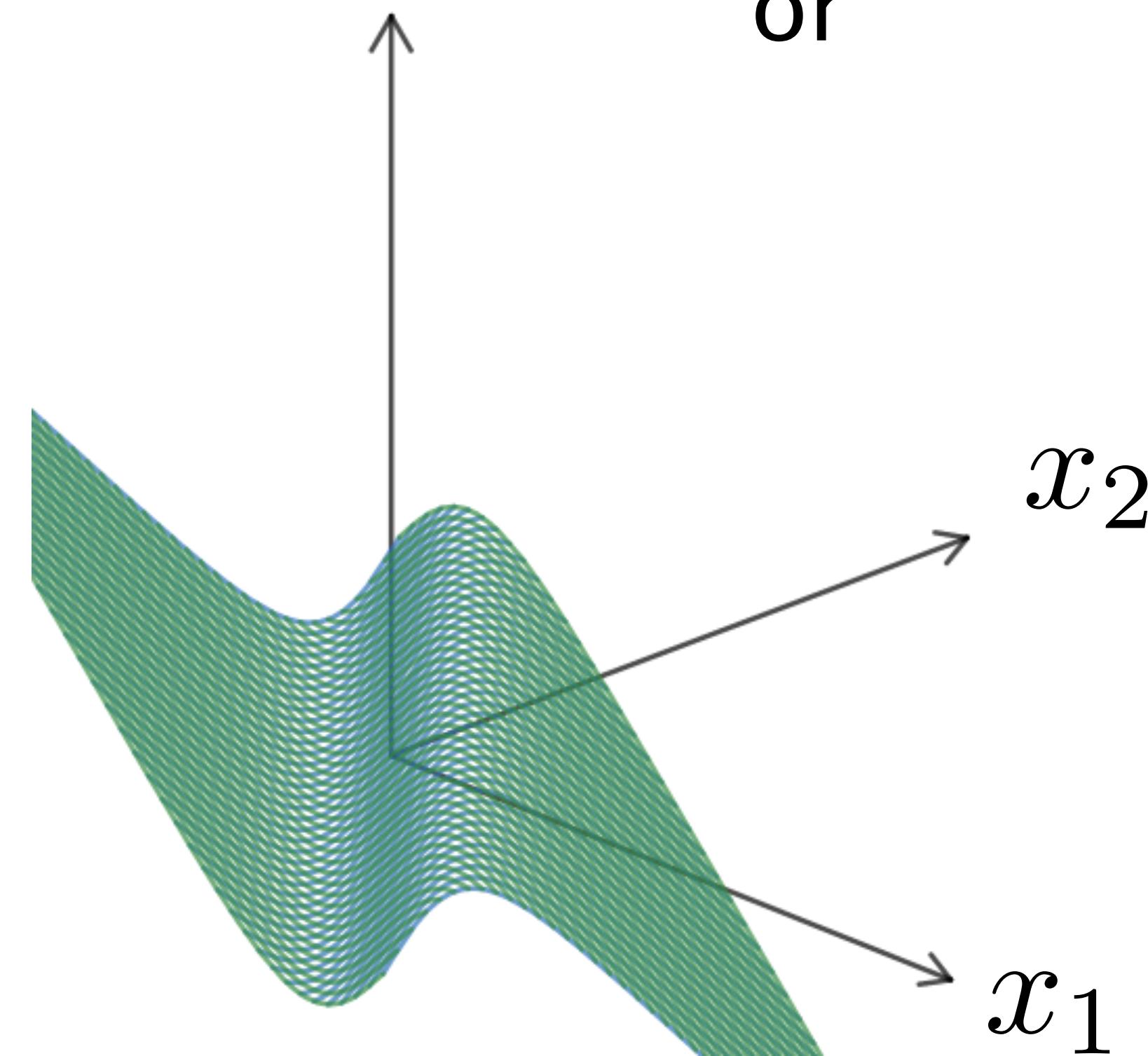
$$y = a_1 x_1 + a_2 x_2 + a_3 \tanh(x_1 + x_2)$$

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

“or”

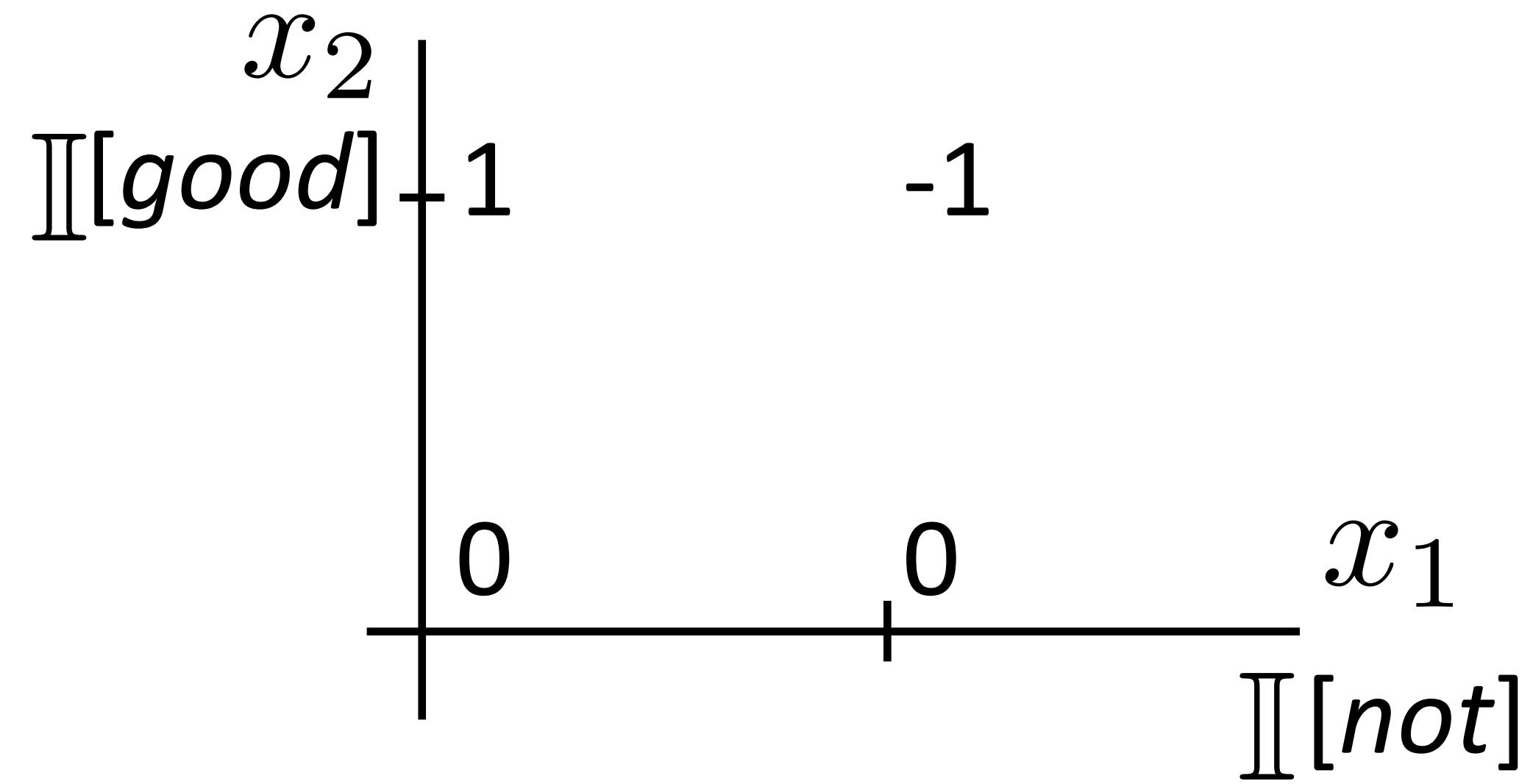
X

✓



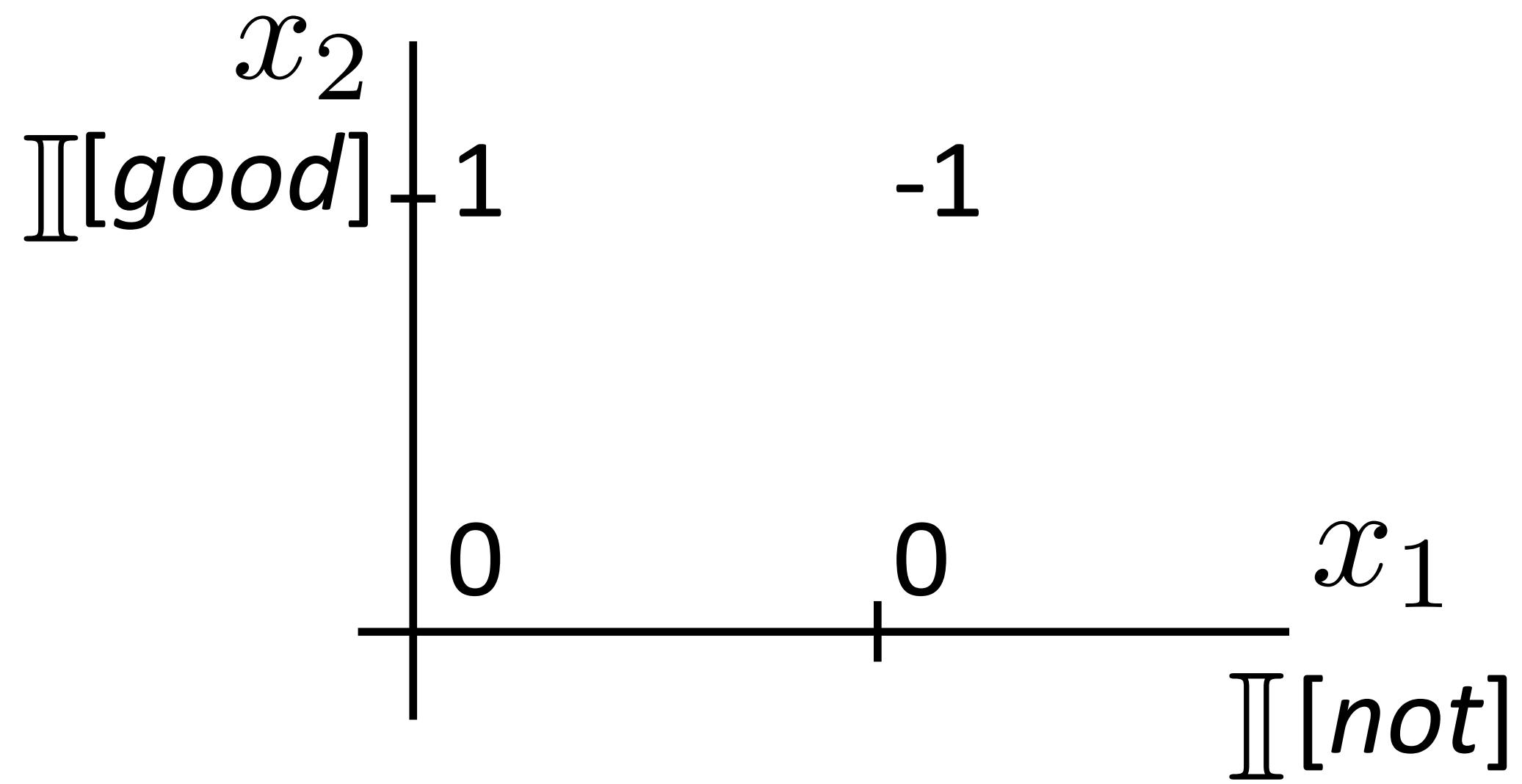
Neural Networks: XOR

Neural Networks: XOR



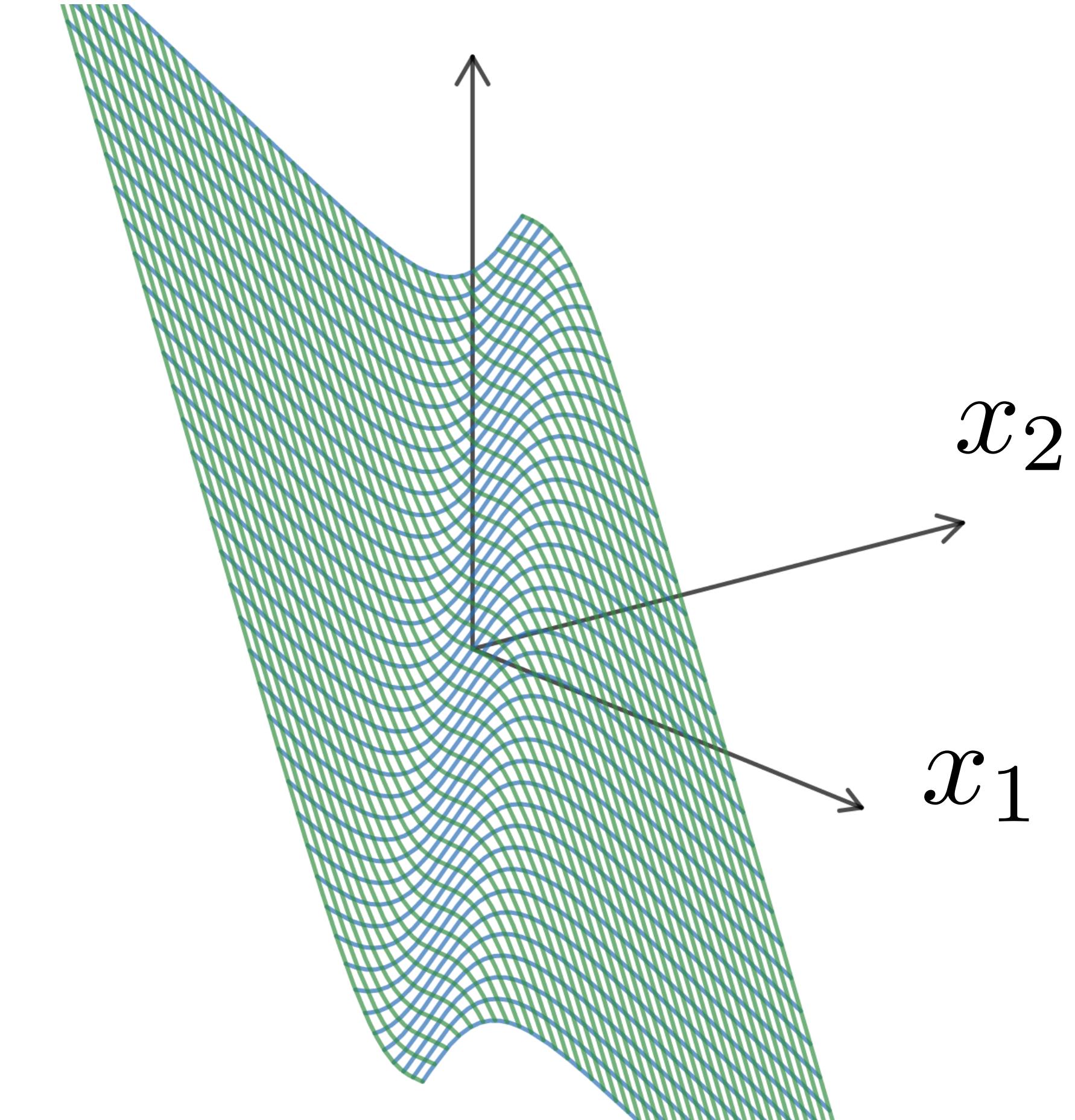
*the movie was **not** all that **good***

Neural Networks: XOR

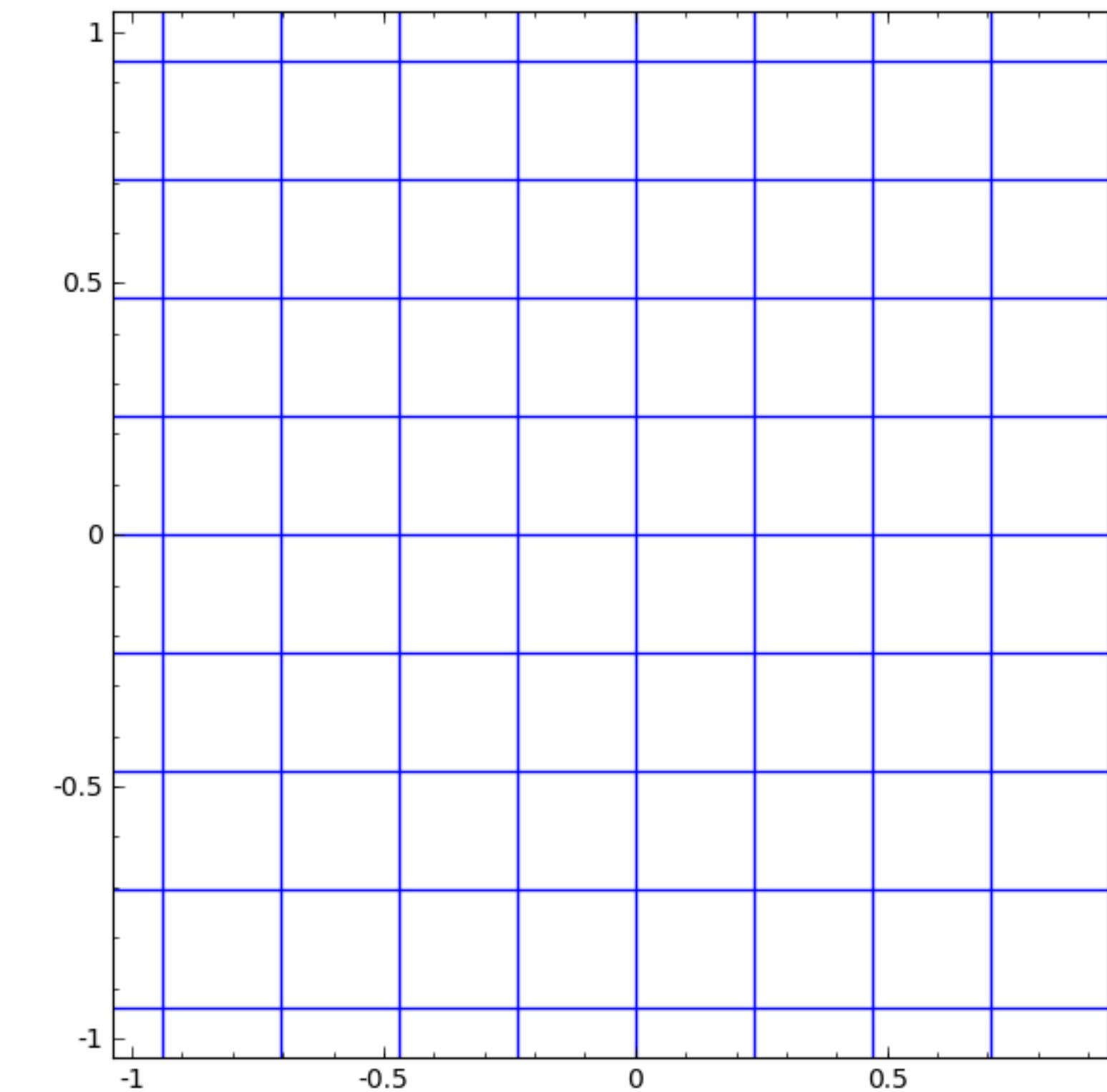


$$y = -2x_1 - x_2 + 2 \tanh(x_1 + x_2)$$

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

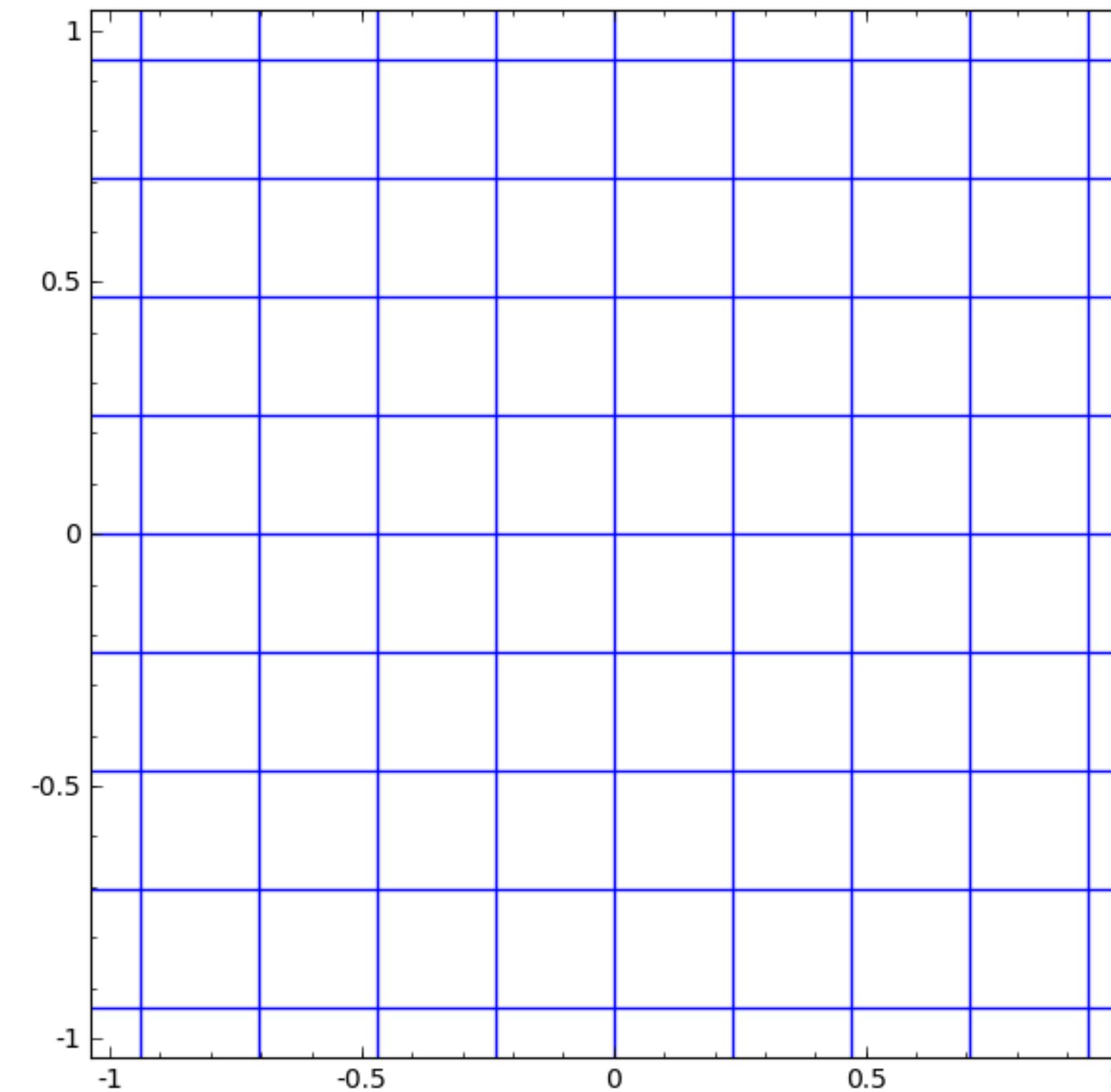


Neural Networks



Neural Networks

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



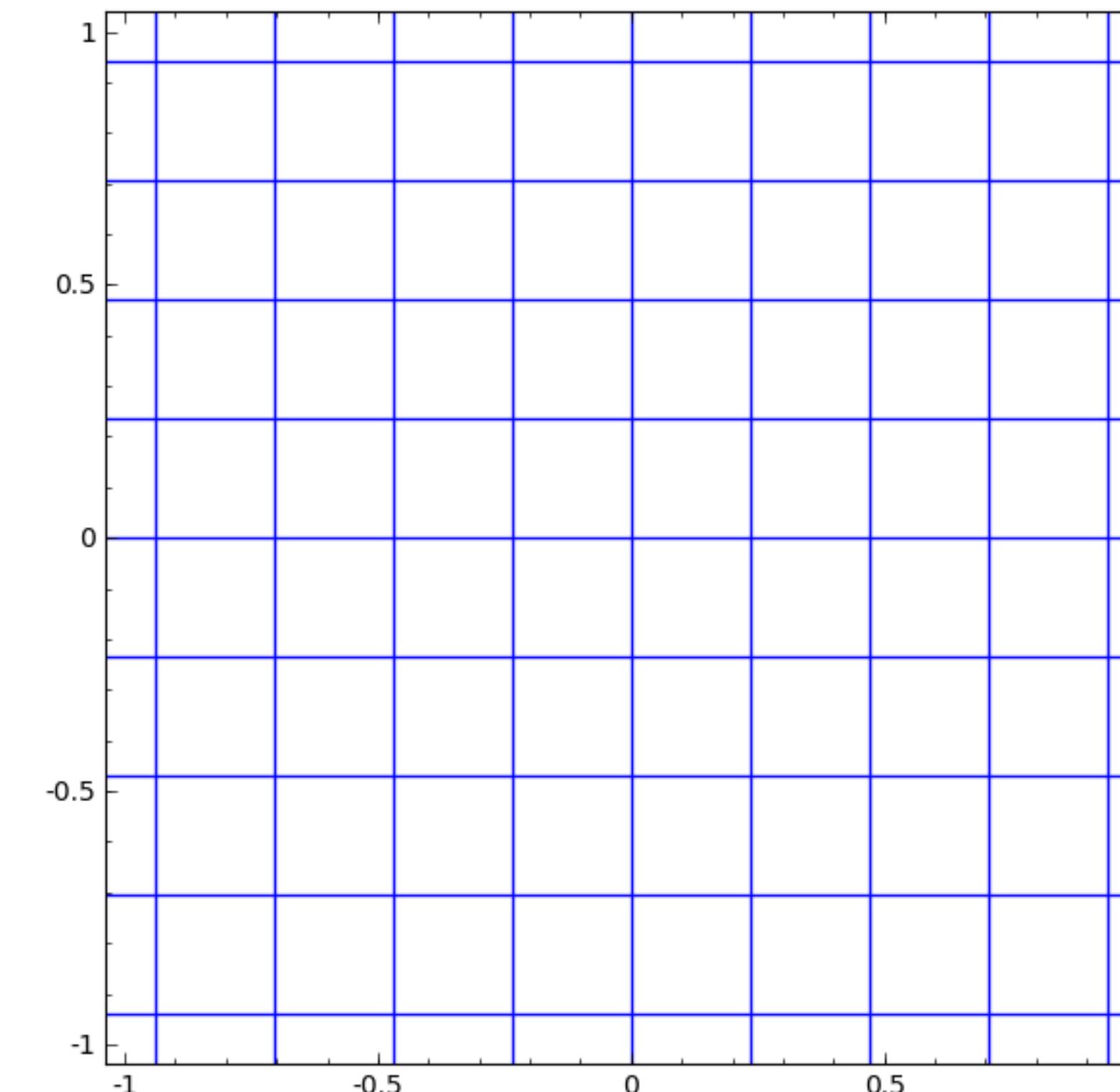
Neural Networks

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

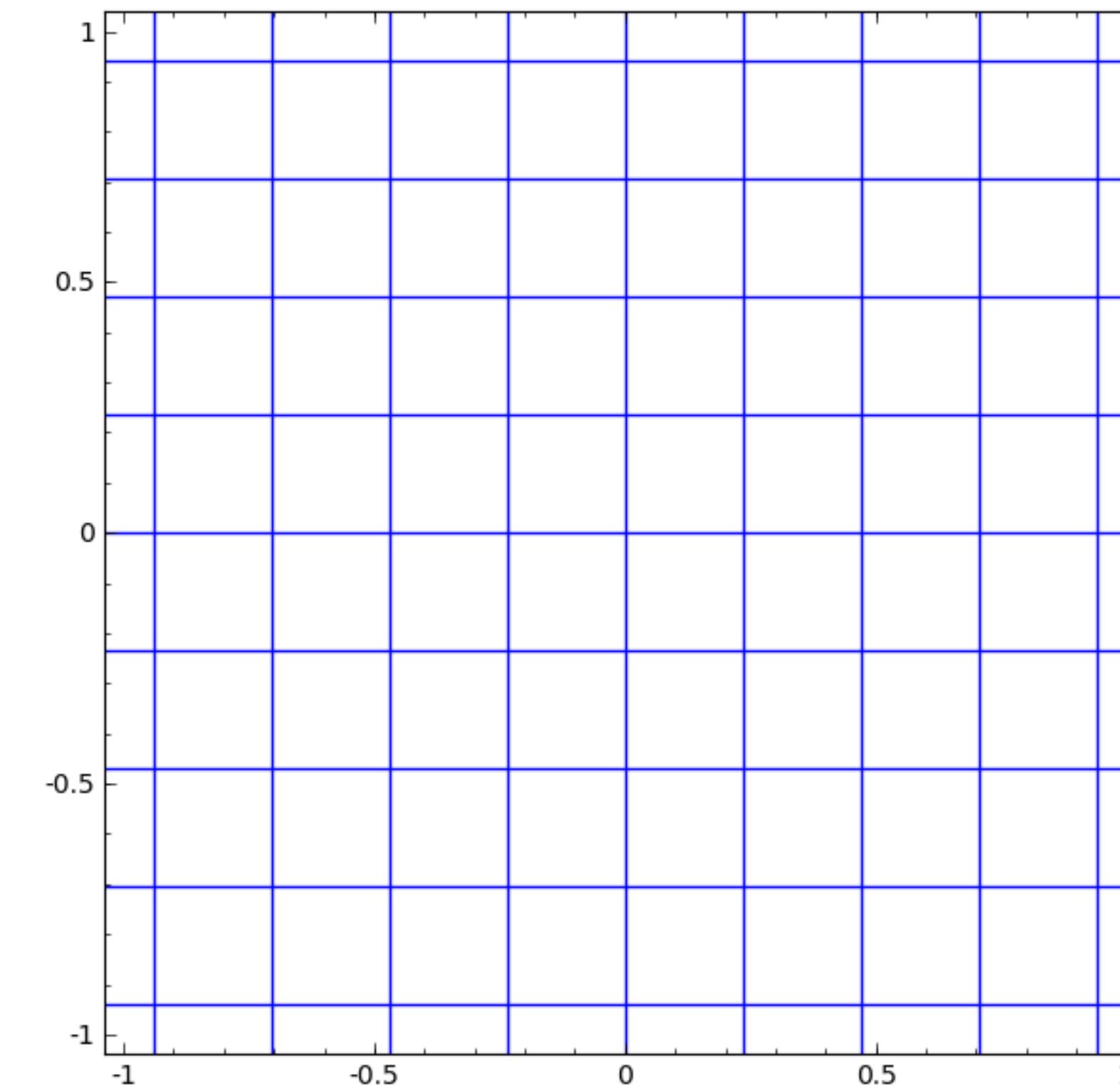
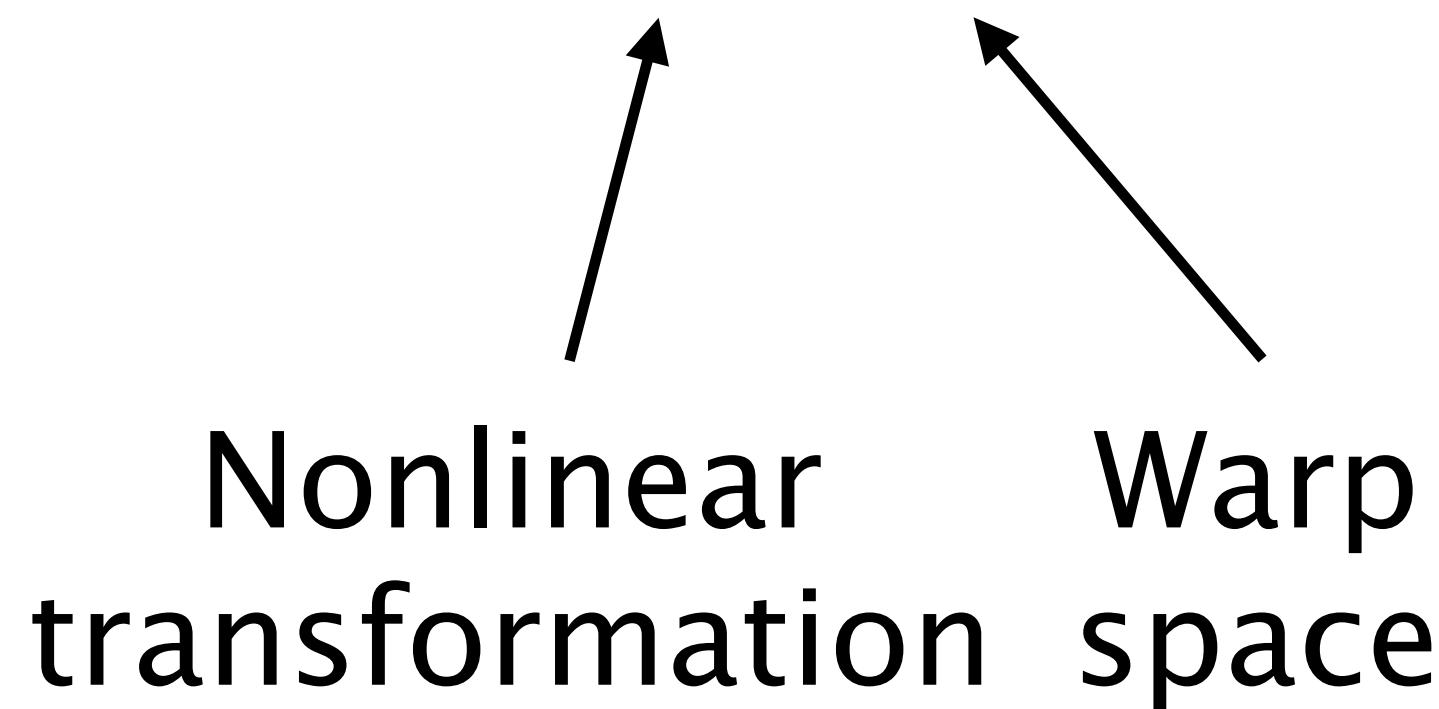


Neural Networks

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

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

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

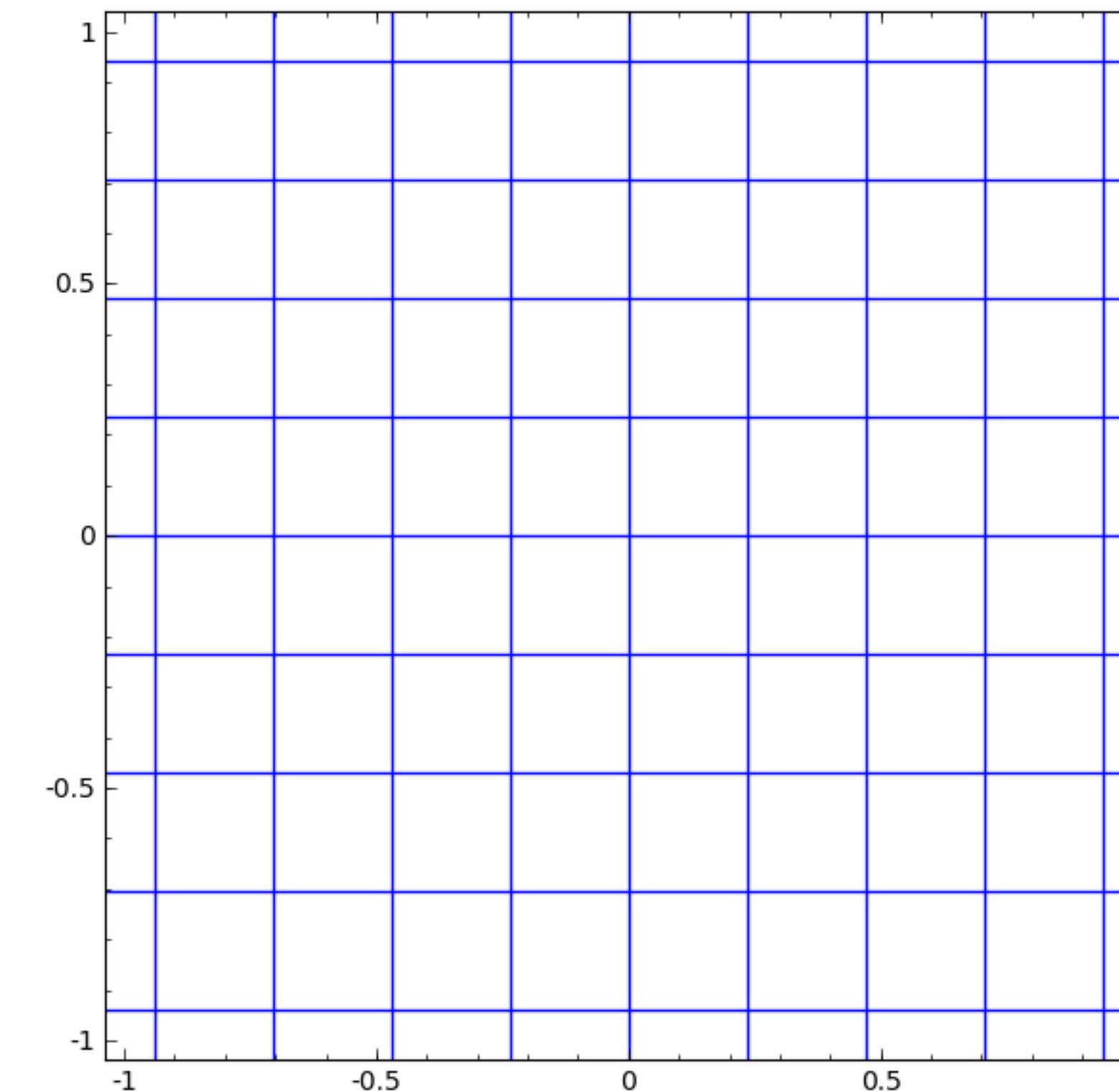
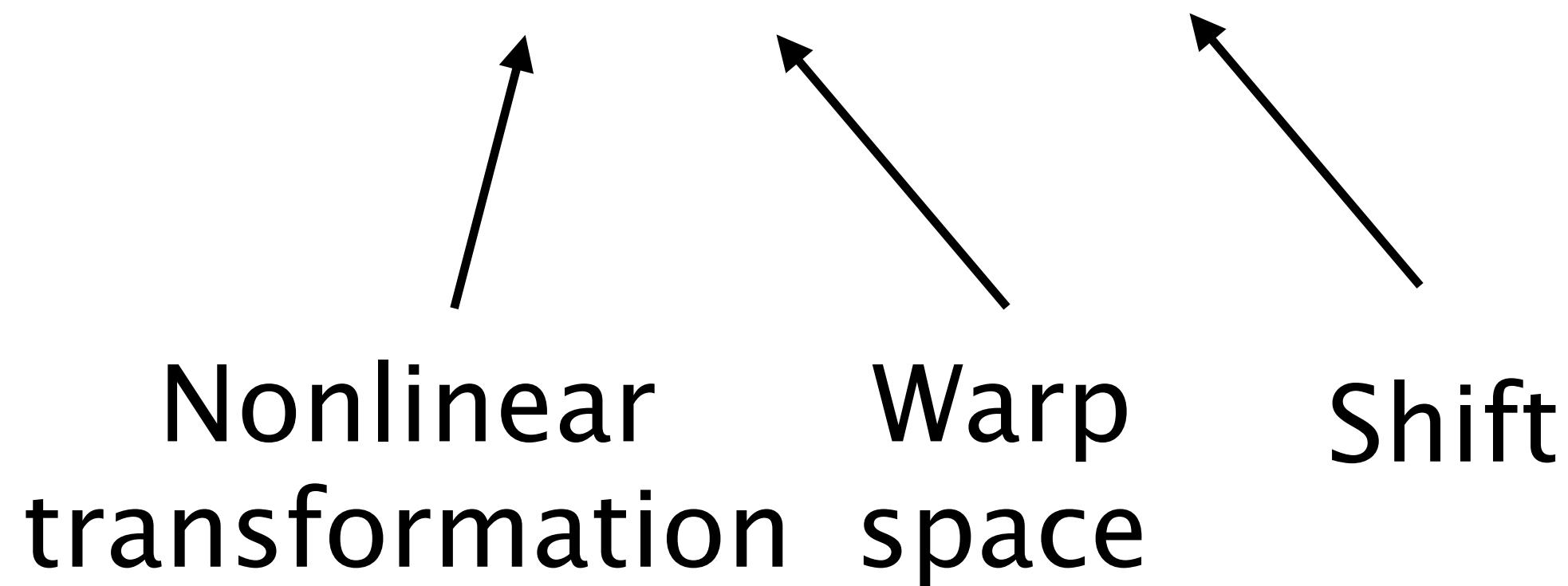


Neural Networks

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

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

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

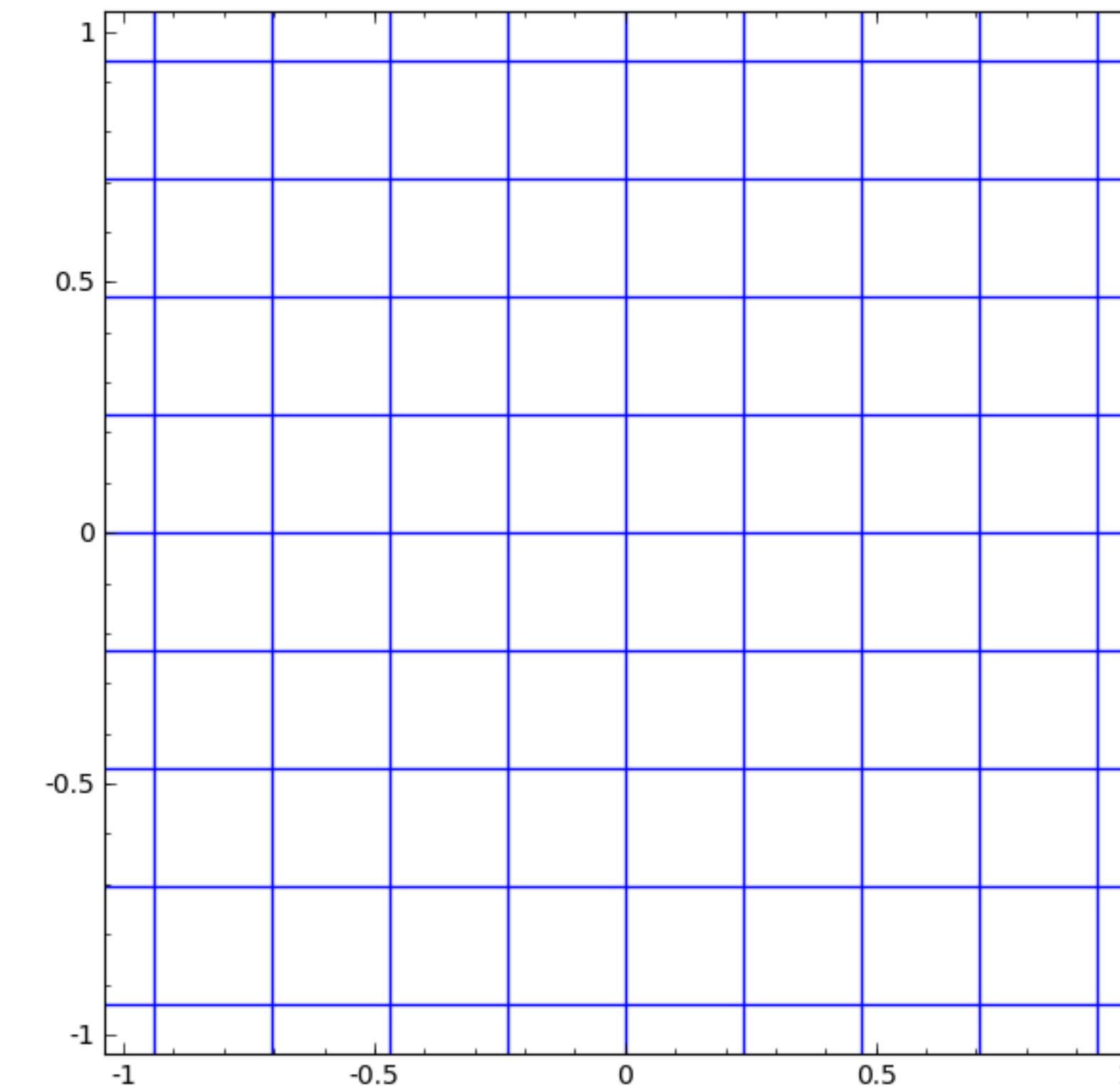
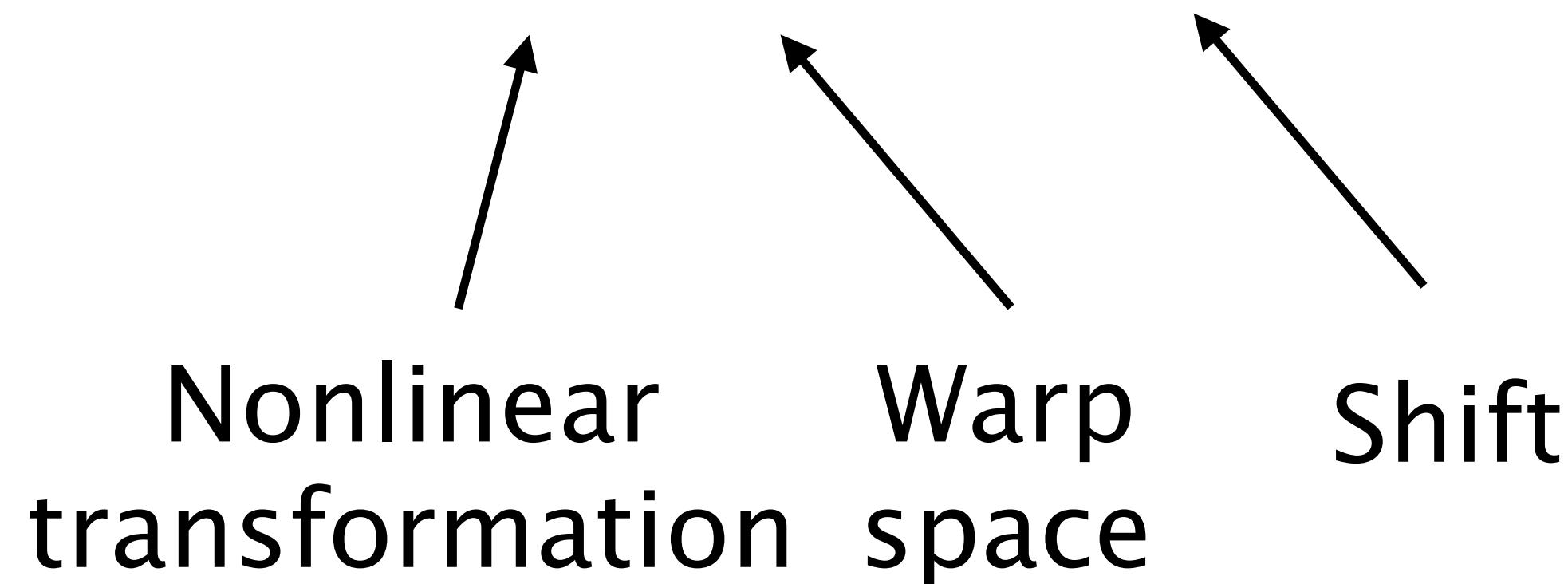


Neural Networks

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

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

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



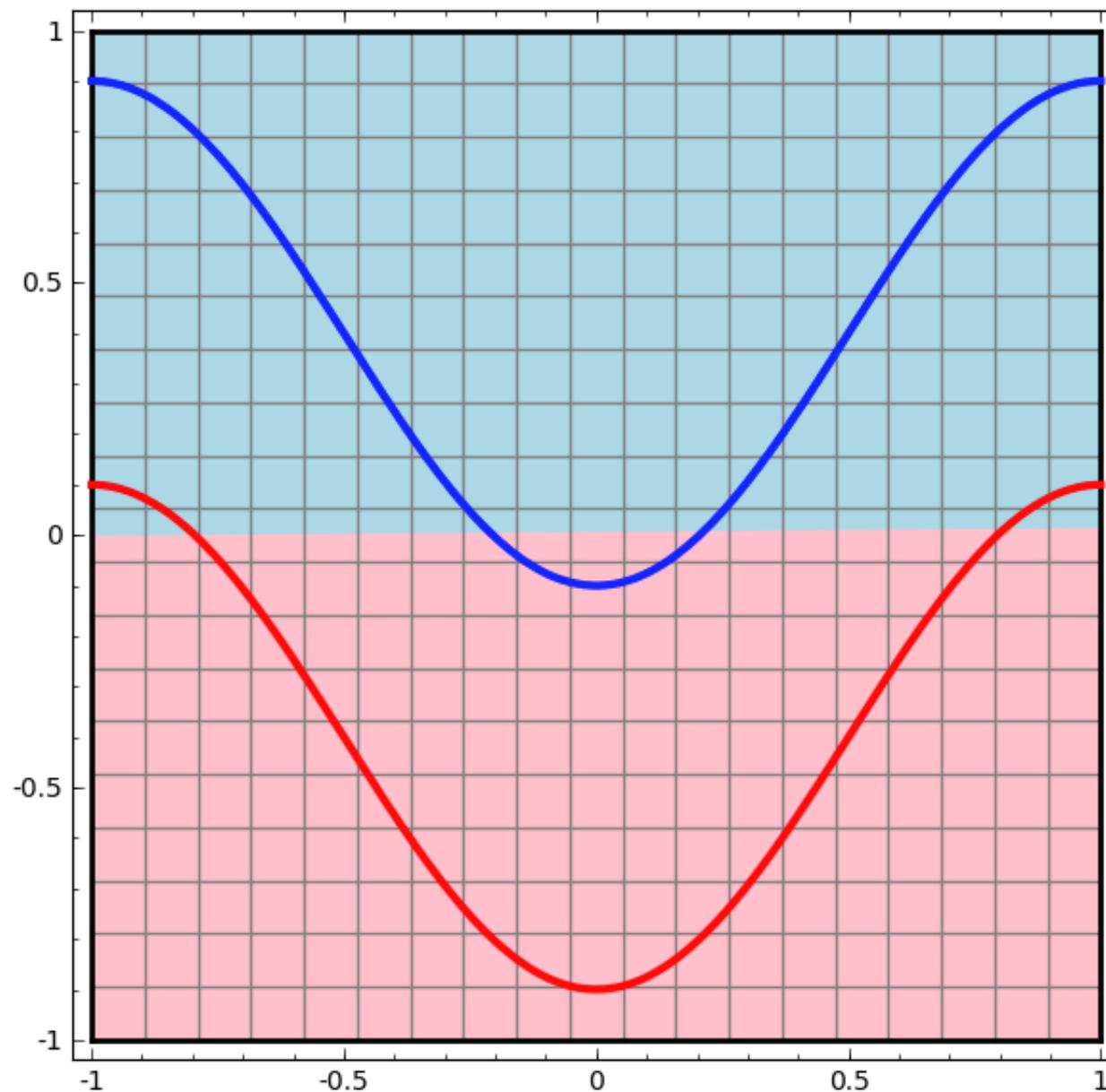
Neural Networks

Neural Networks

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

Neural Networks

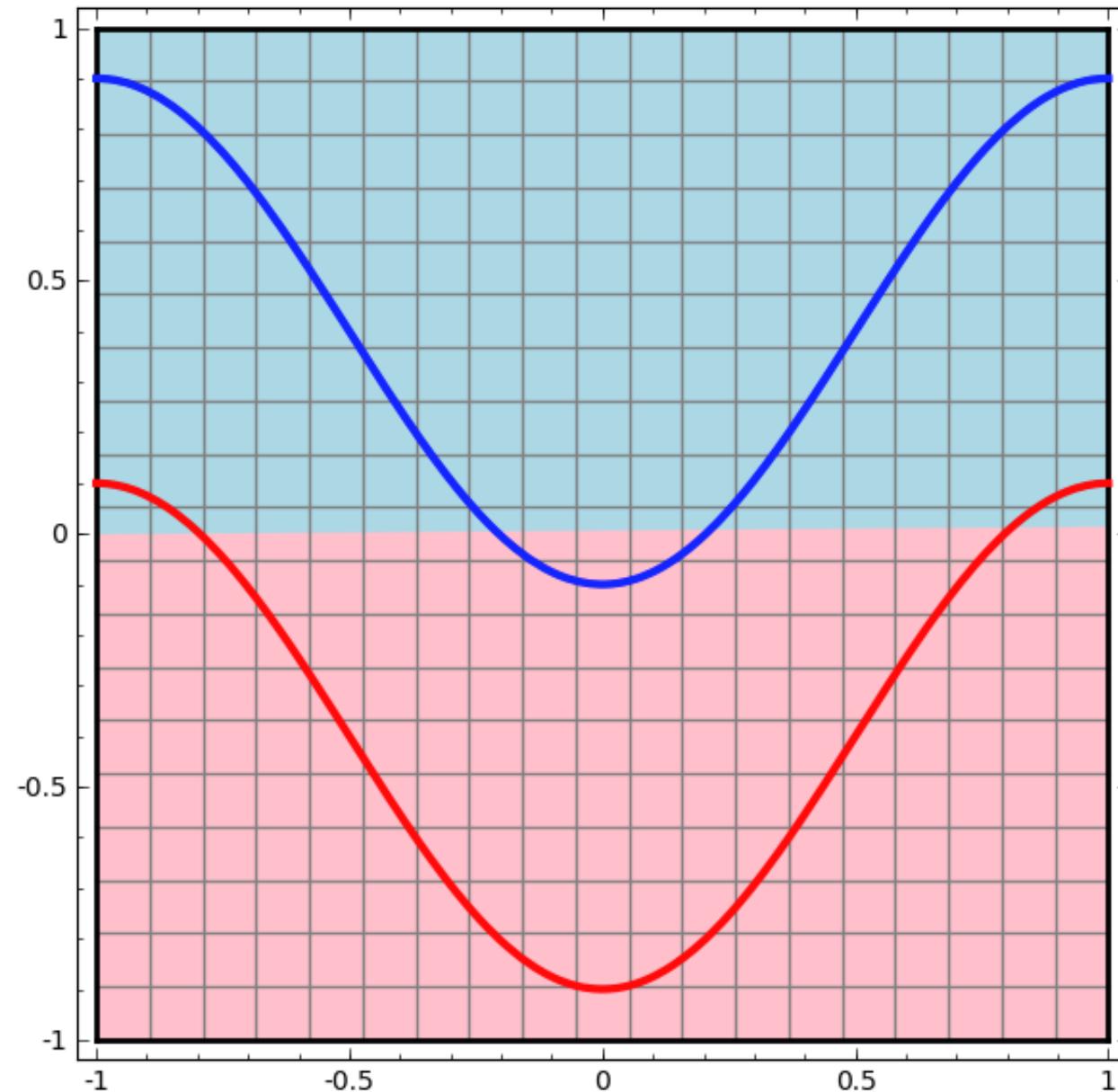
Linear classifier



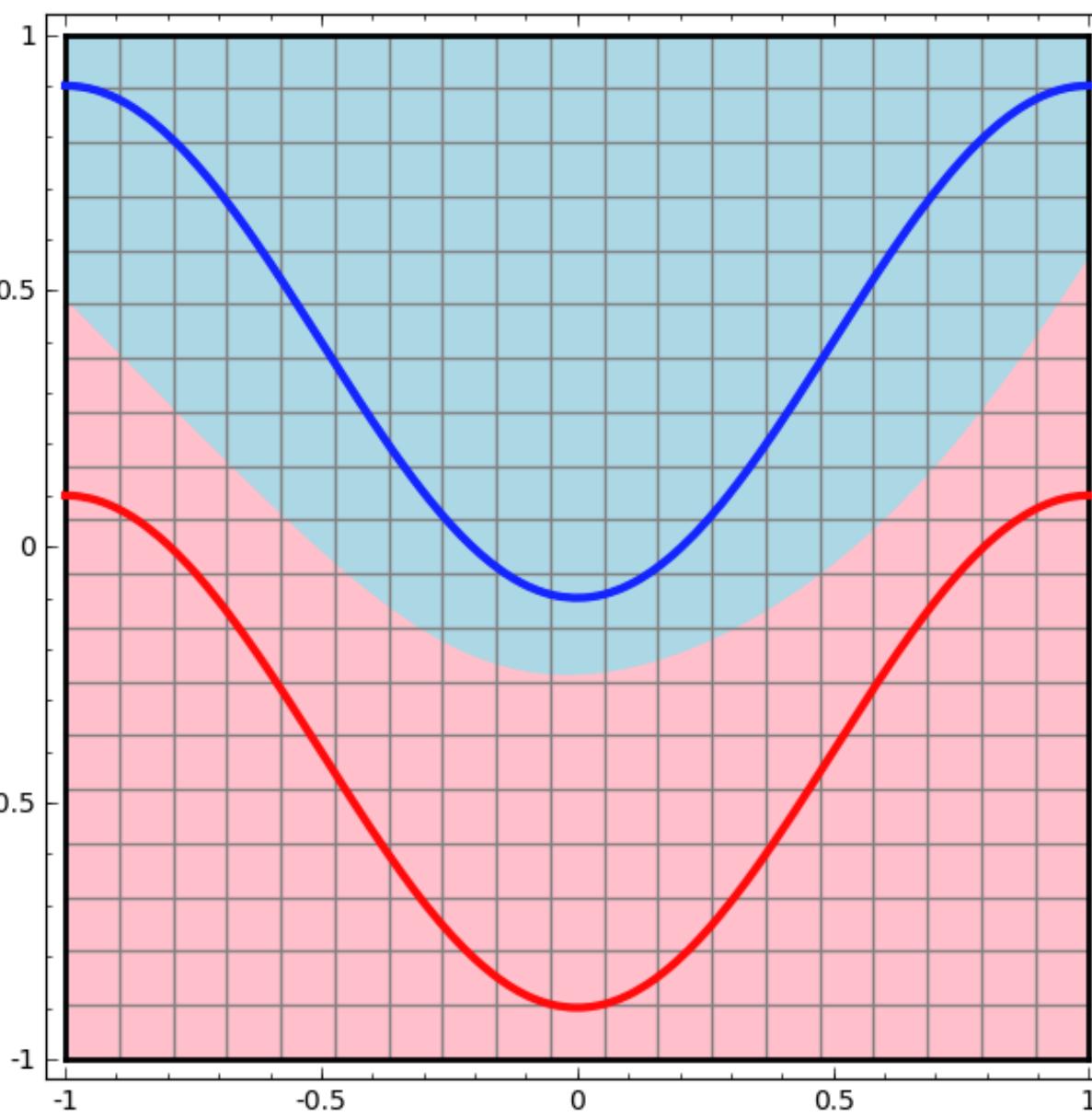
Taken from <http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/>

Neural Networks

Linear classifier

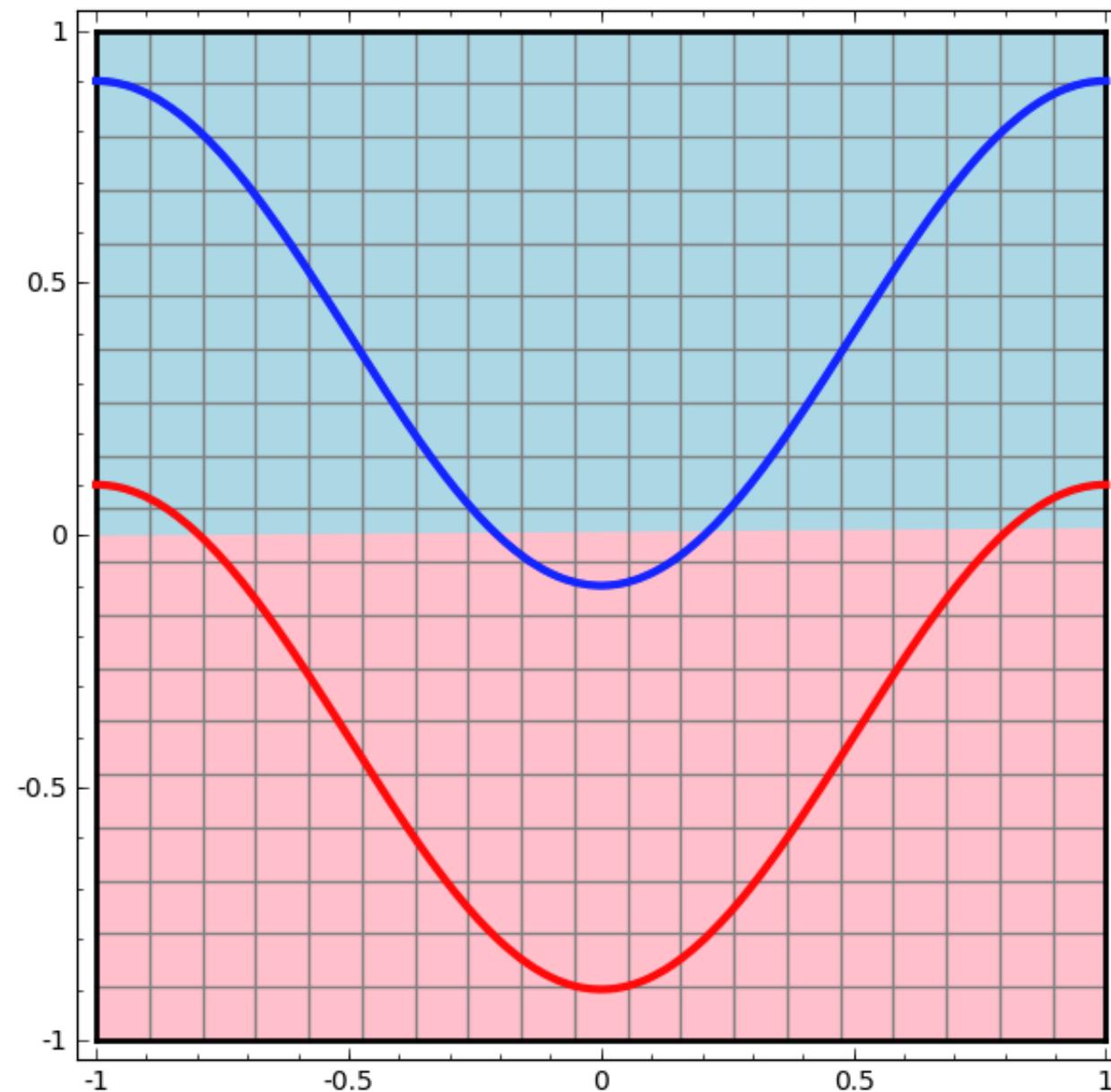


Neural network

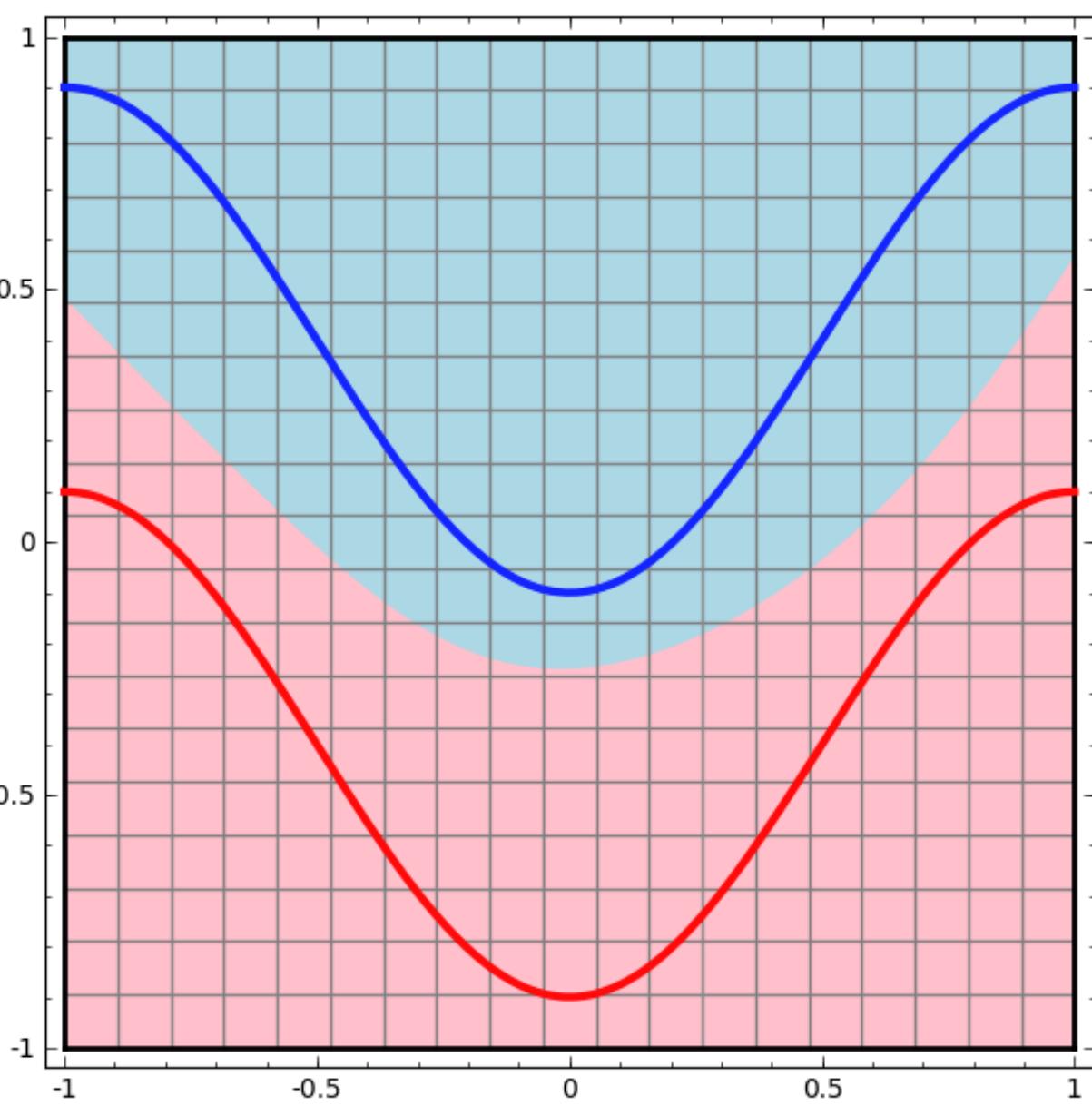


Neural Networks

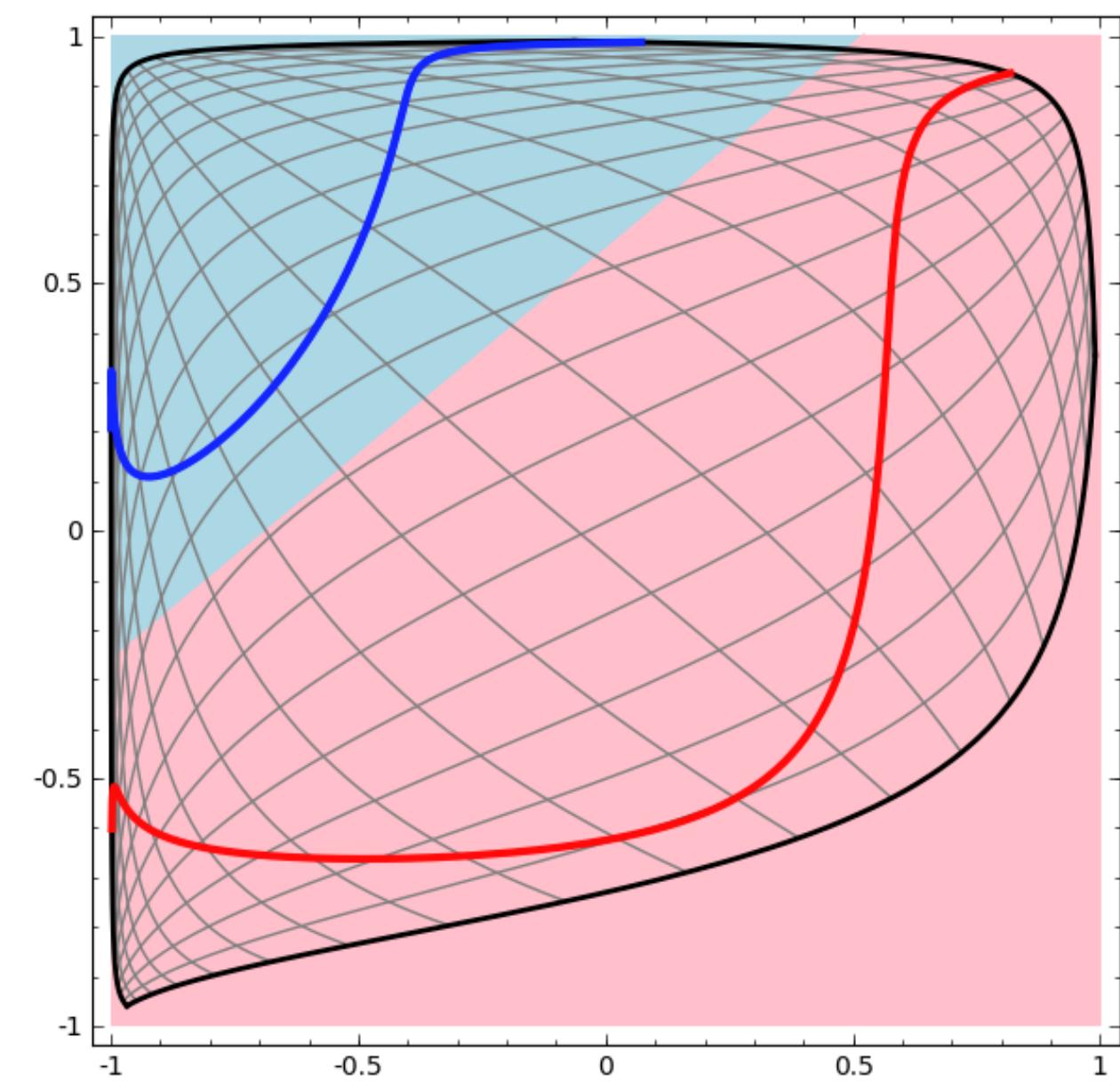
Linear classifier



Neural network

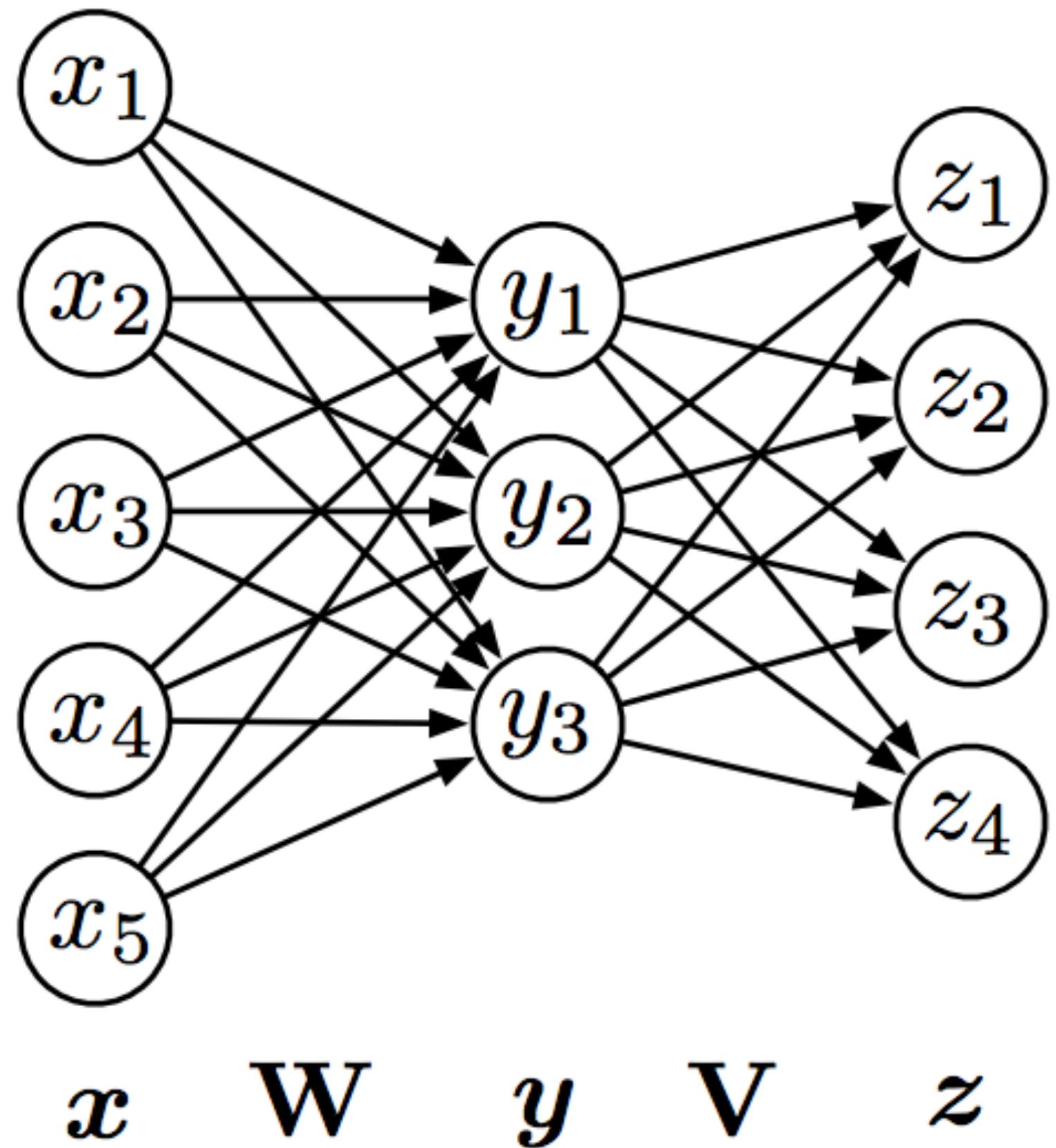


...possible because
we transformed the
space!

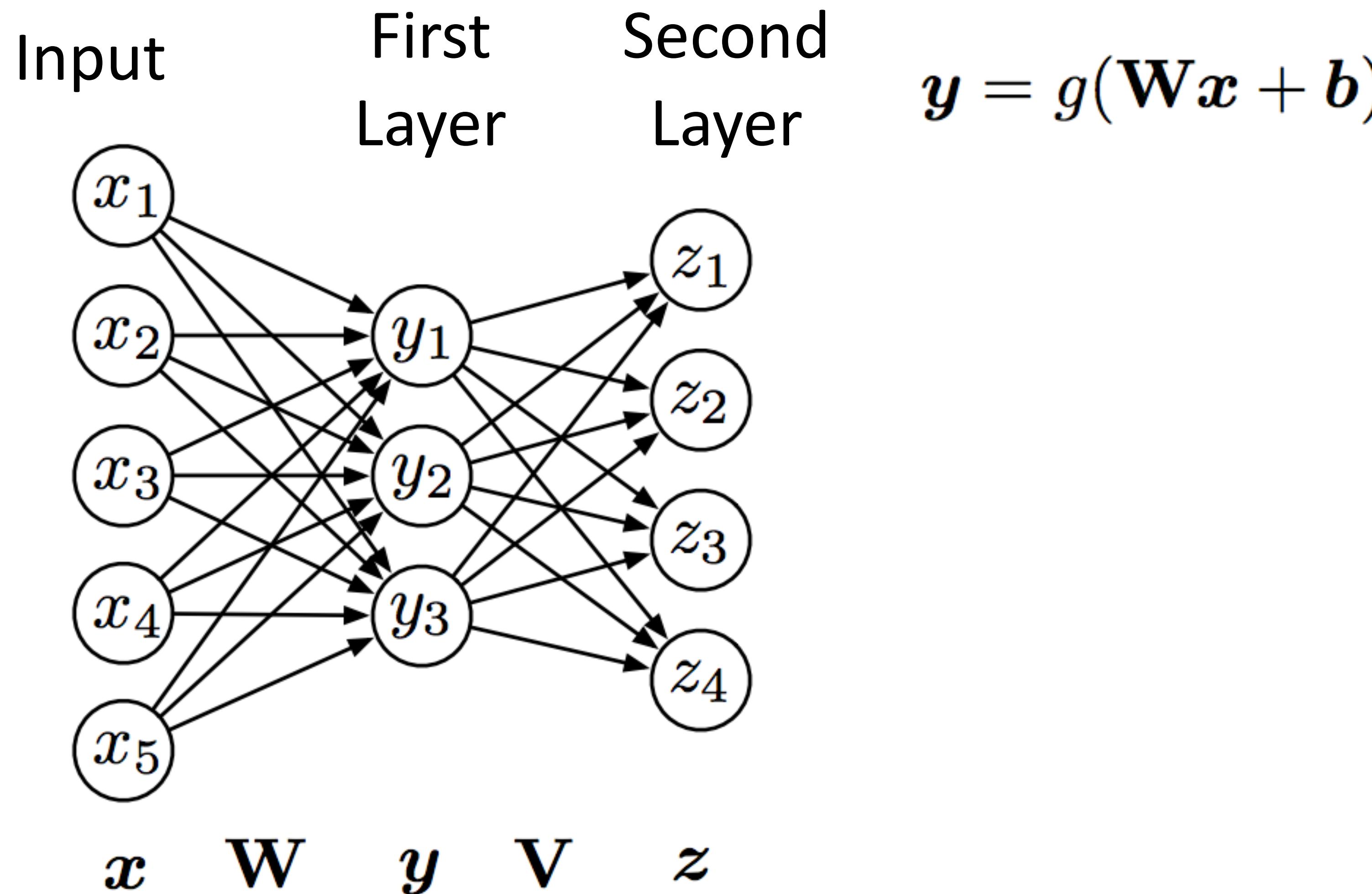


Deep Neural Networks

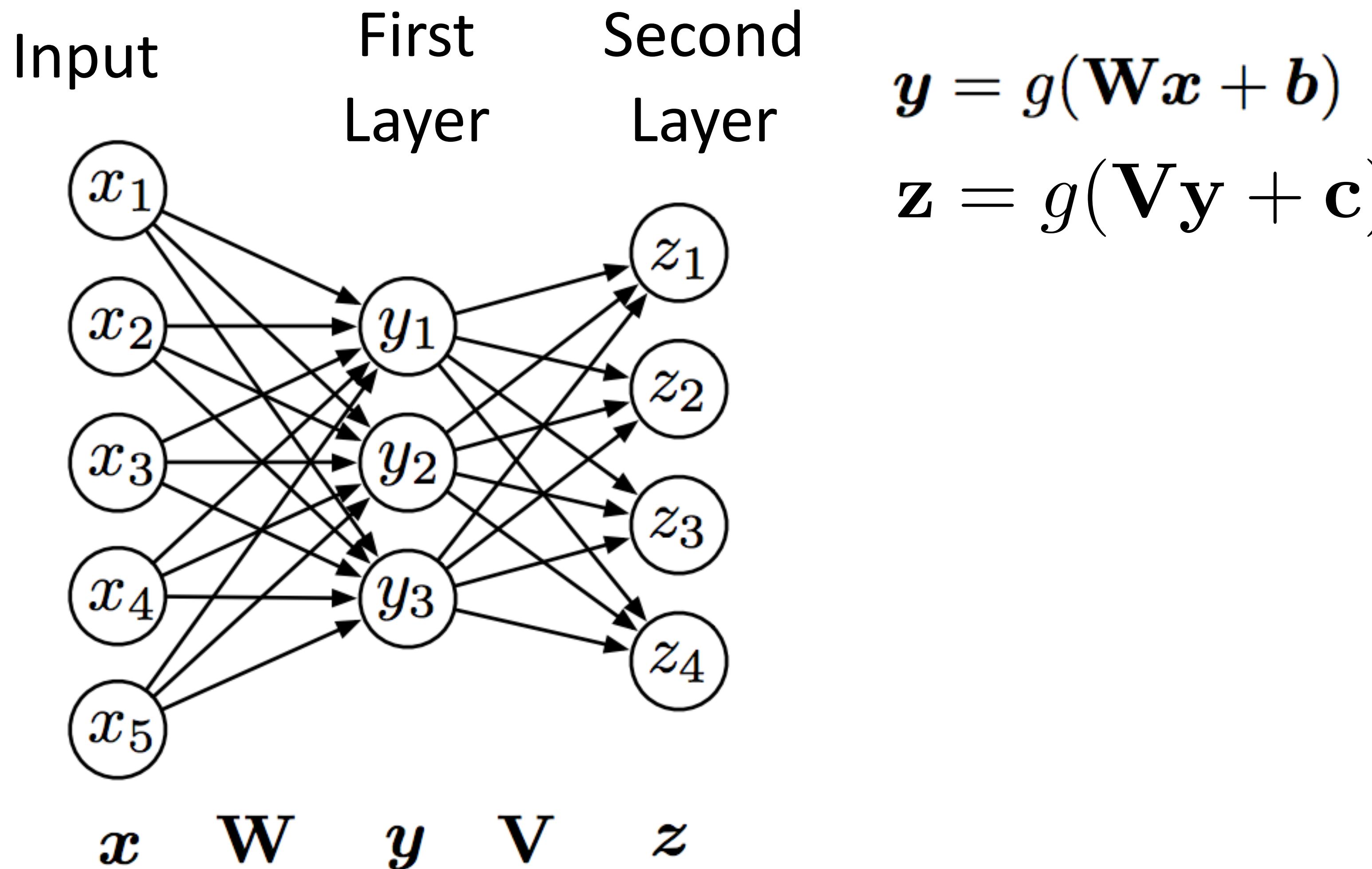
$$\mathbf{y} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$



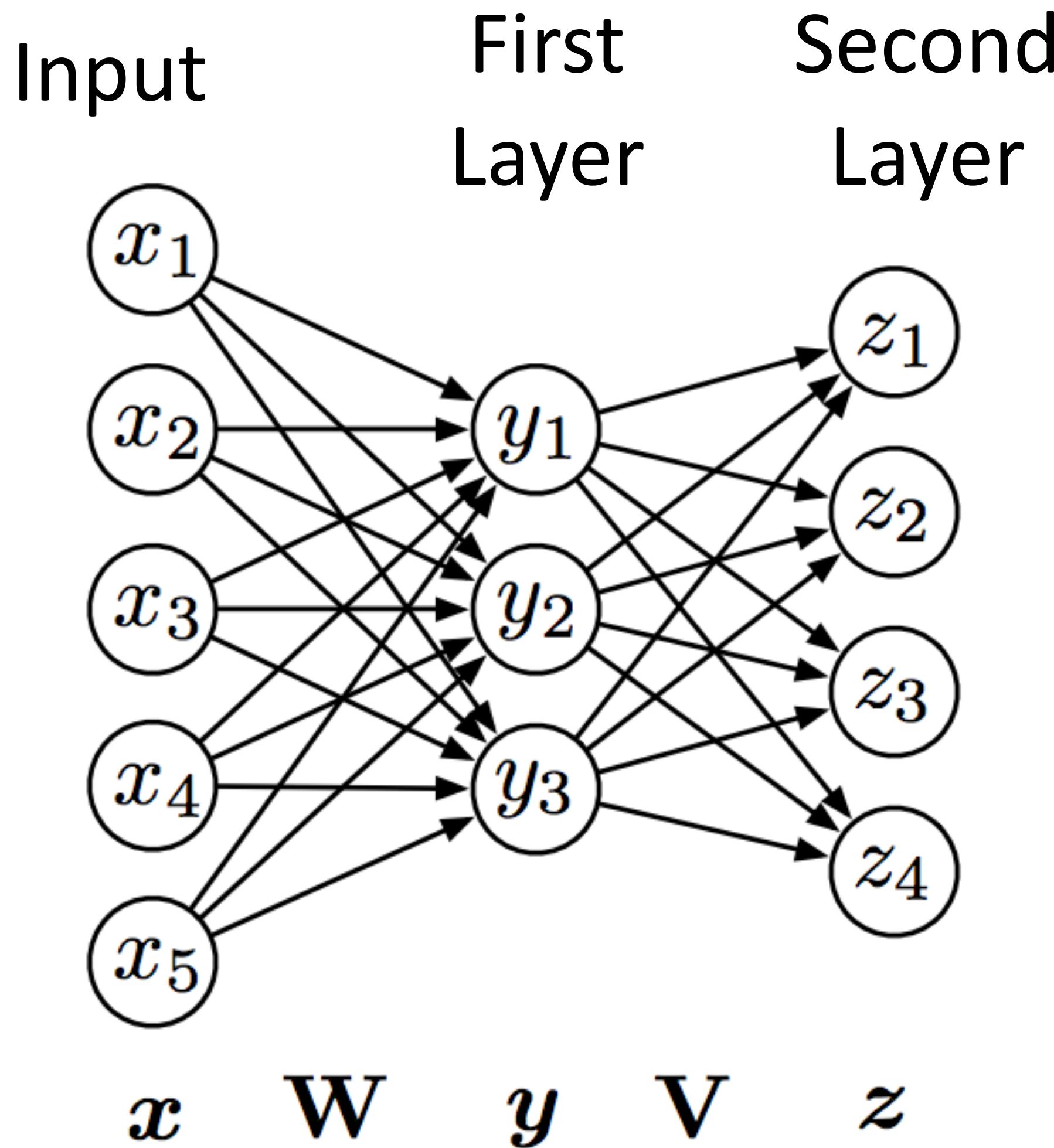
Deep Neural Networks



Deep Neural Networks



Deep Neural Networks



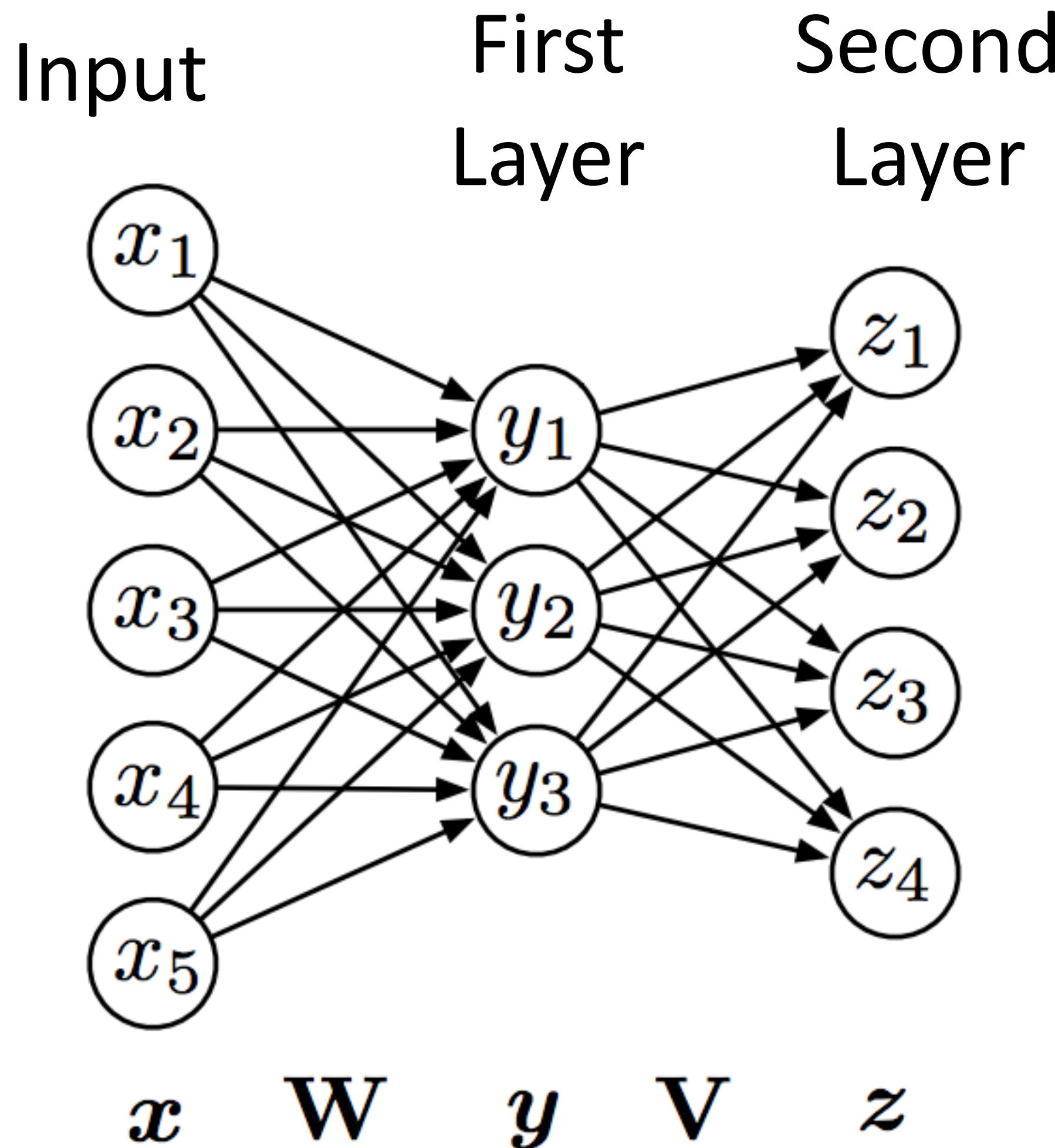
$$\mathbf{y} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$$

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

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

output of first layer

Deep Neural Networks



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

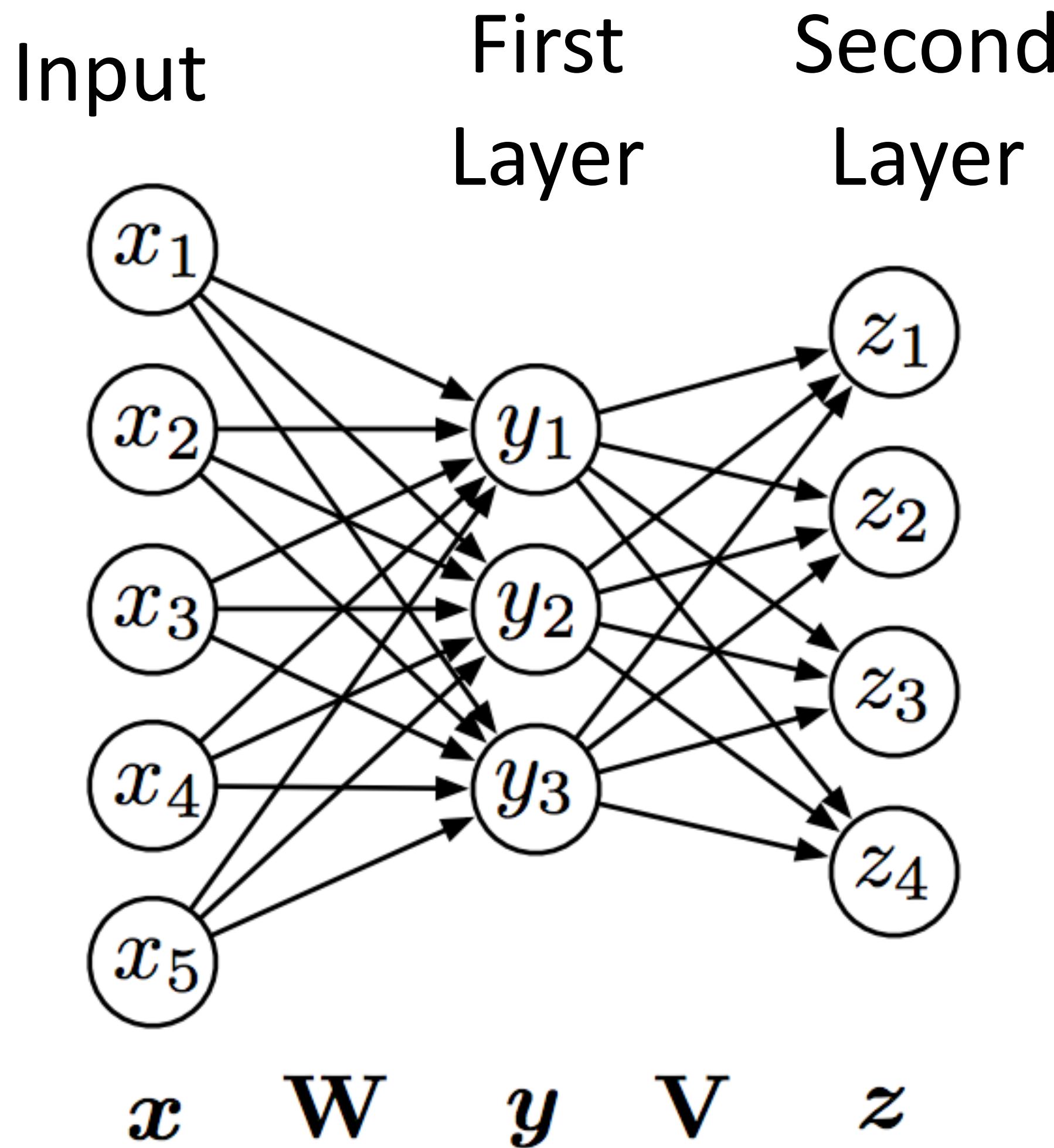
$$\mathbf{z} = g(\mathbf{V}\mathbf{y} + \mathbf{c})$$

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

output of first layer

“Feedforward” computation (not recurrent)

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

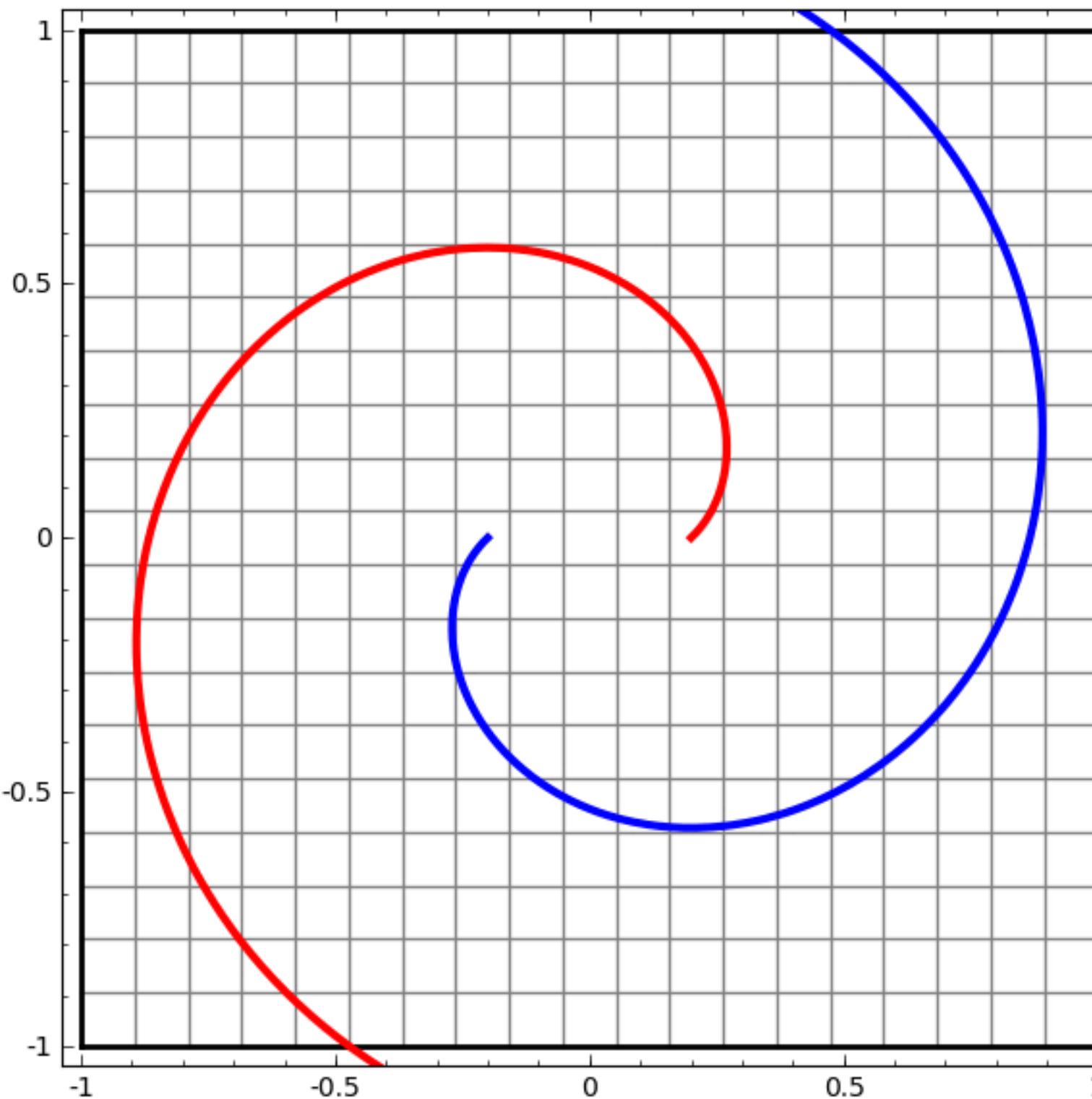
“Feedforward” computation (not recurrent)

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

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

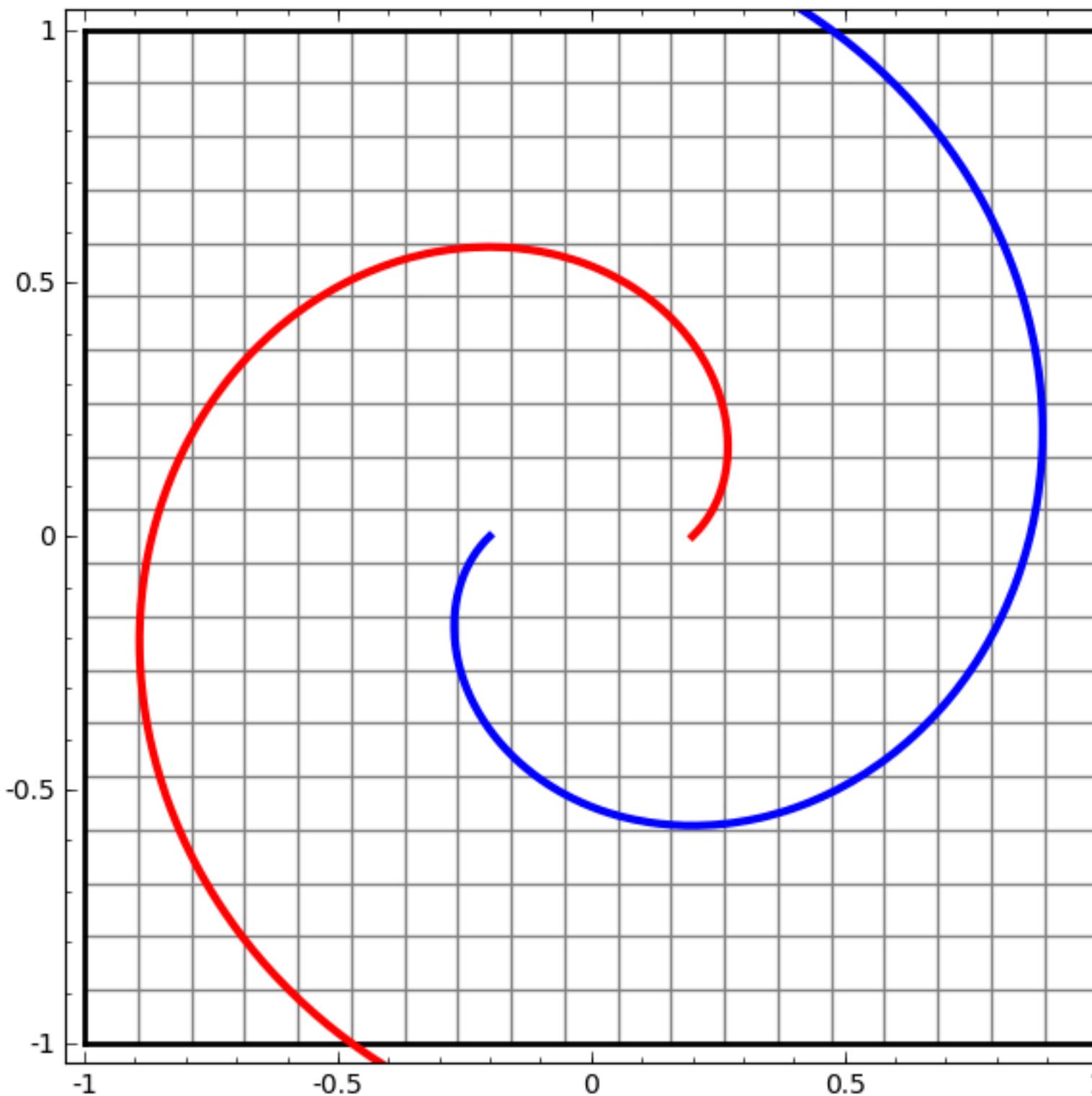
Adopted from Chris Dyer

Deep Neural Networks



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

Deep Neural Networks



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

Feedforward Networks, Backpropagation

Logistic Regression with NNs

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

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

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

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

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

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

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

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

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

Logistic Regression with NNs

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

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

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

- ▶ Single scalar probability
- ▶ Compute scores for all possible labels at once (returns vector)
- ▶ softmax: exps and normalizes a given vector

Logistic Regression with NNs

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

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

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

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

- ▶ 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]

Logistic Regression with NNs

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

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

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

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

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

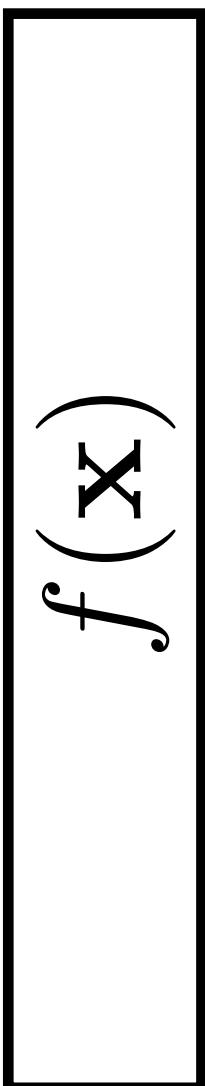
- ▶ Single scalar probability
- ▶ Compute scores for all possible labels at once (returns vector)
- ▶ softmax: exps and normalizes a given vector
- ▶ Weight vector per class; W is [num classes x num feats]
- ▶ Now one hidden layer

Neural Networks for Classification

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

Neural Networks for Classification

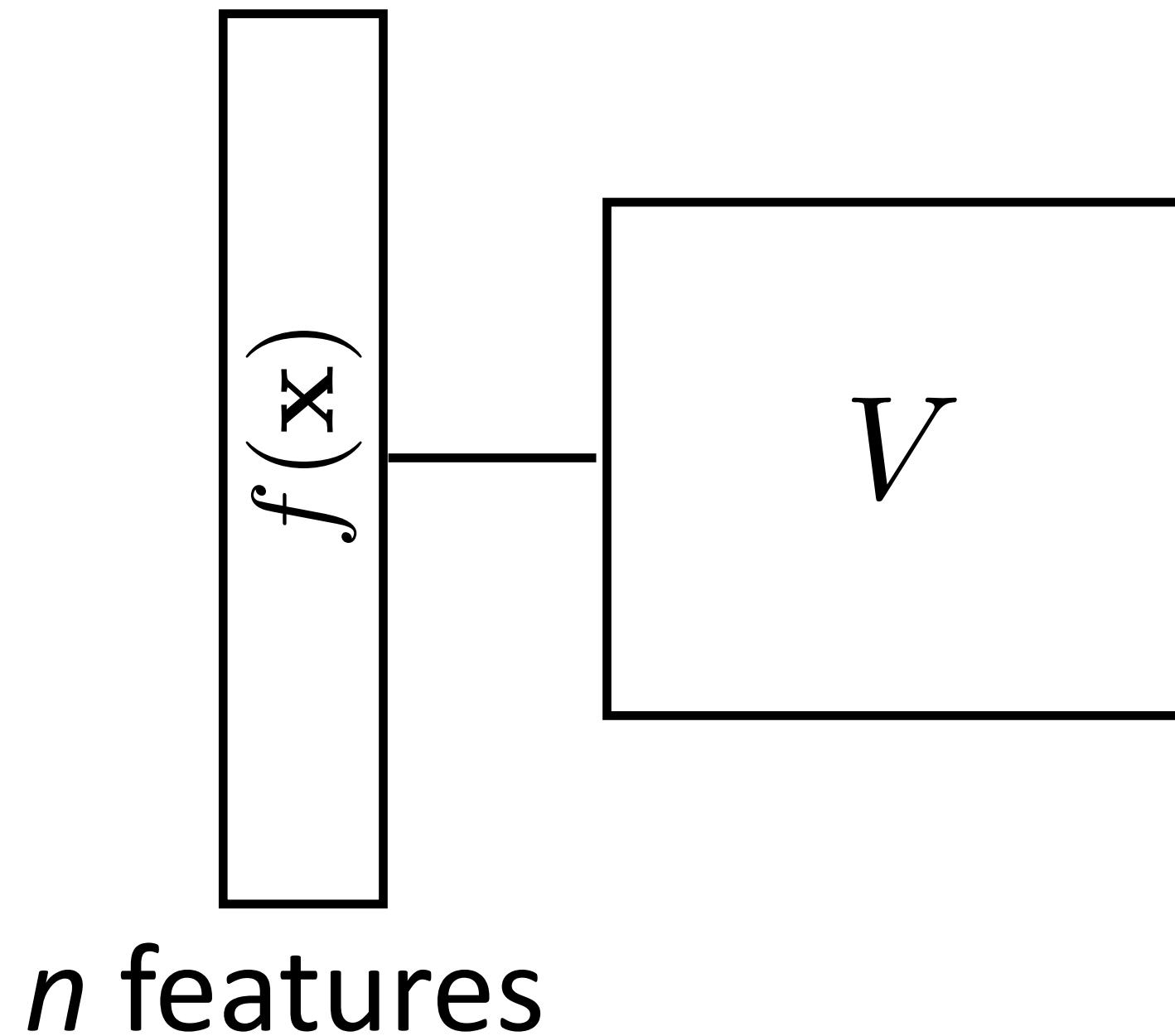
$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wg(Vf(\mathbf{x})))$$



n features

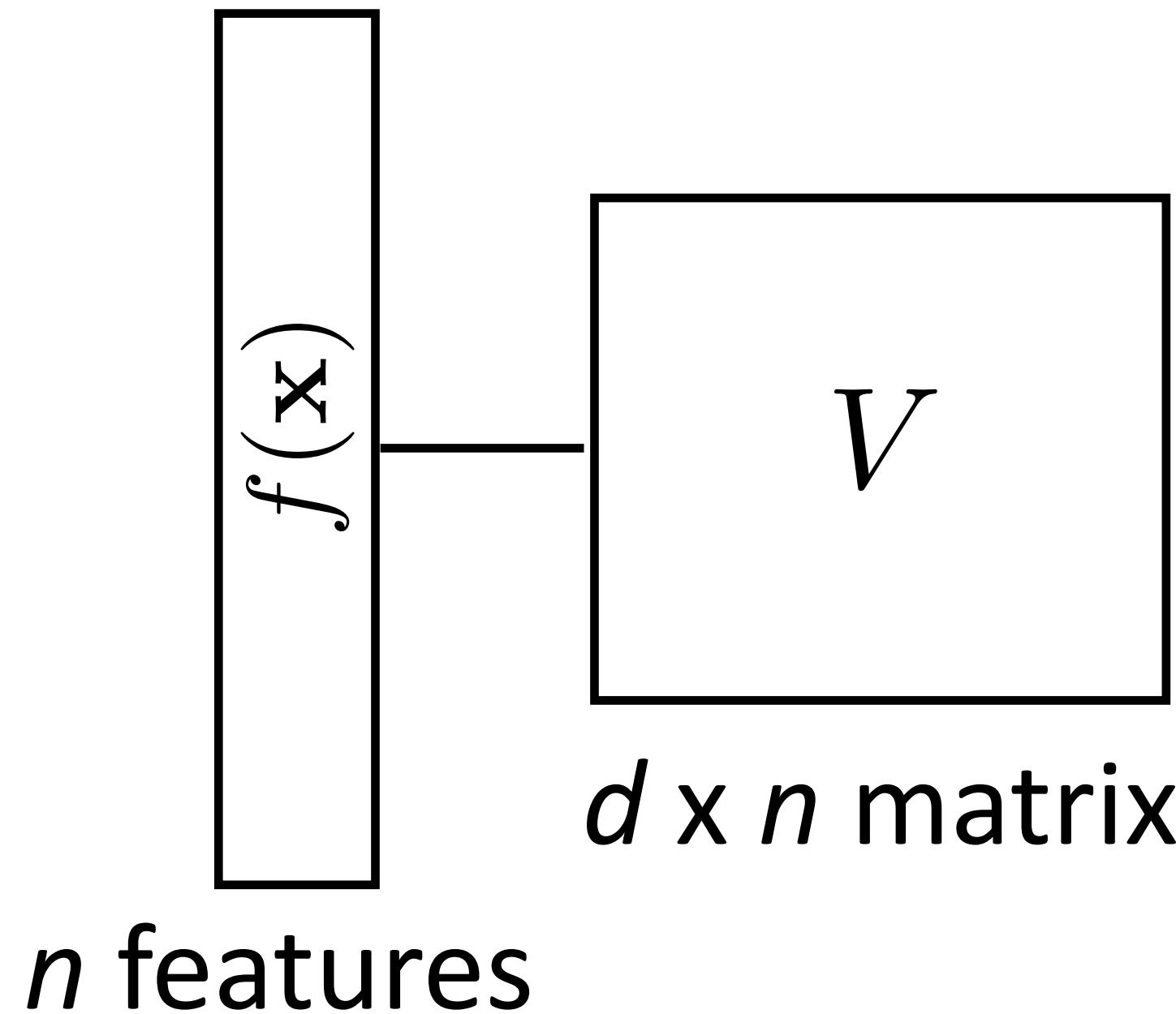
Neural Networks for Classification

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



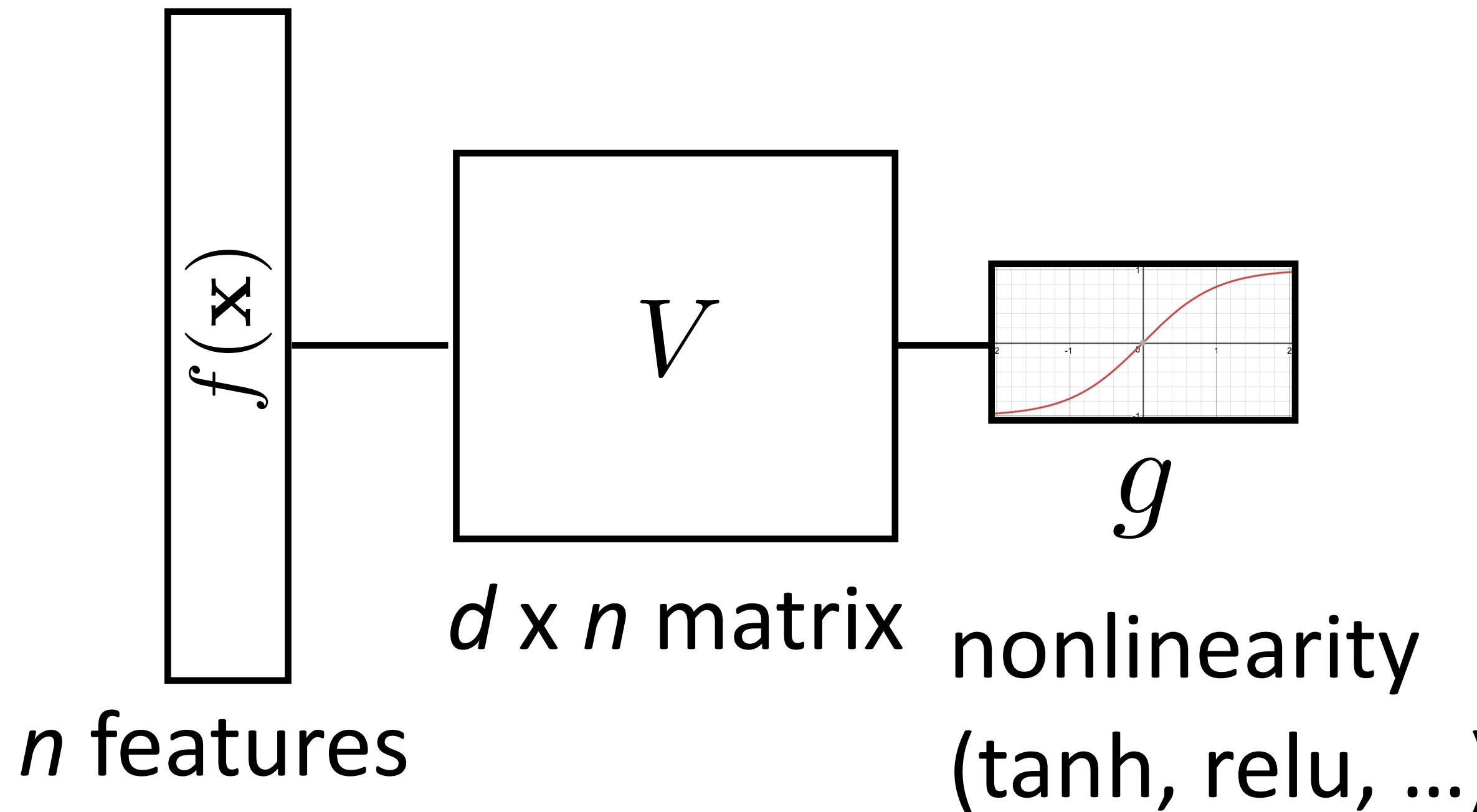
Neural Networks for Classification

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



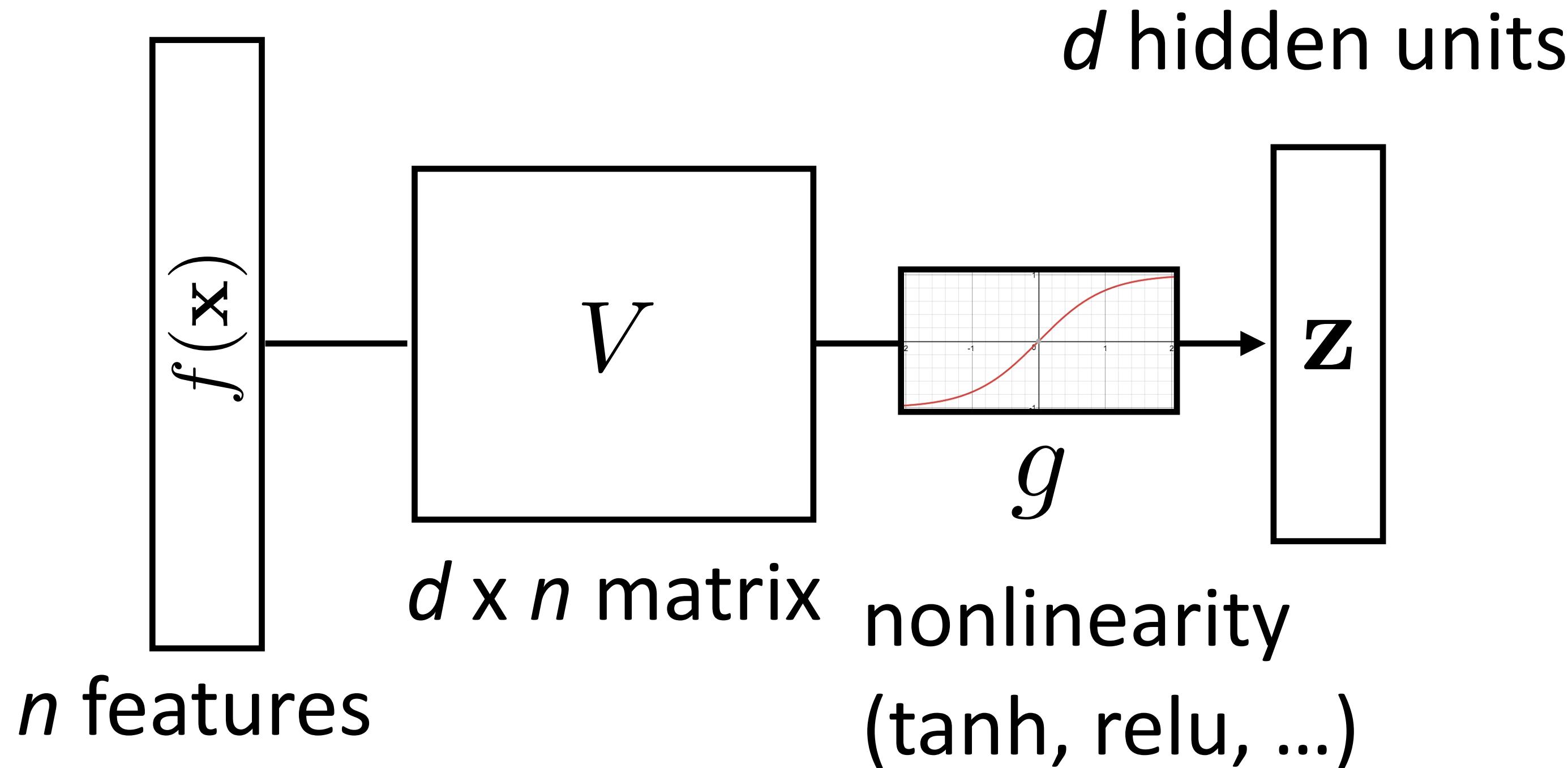
Neural Networks for Classification

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



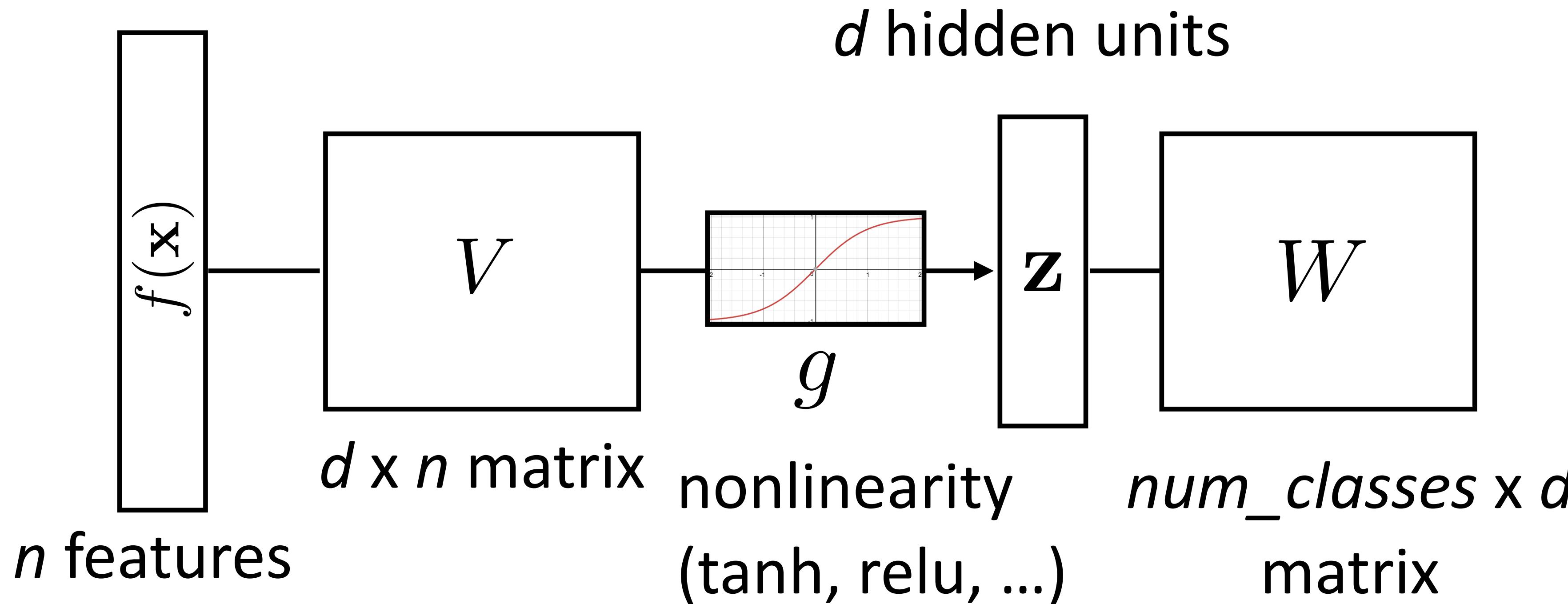
Neural Networks for Classification

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



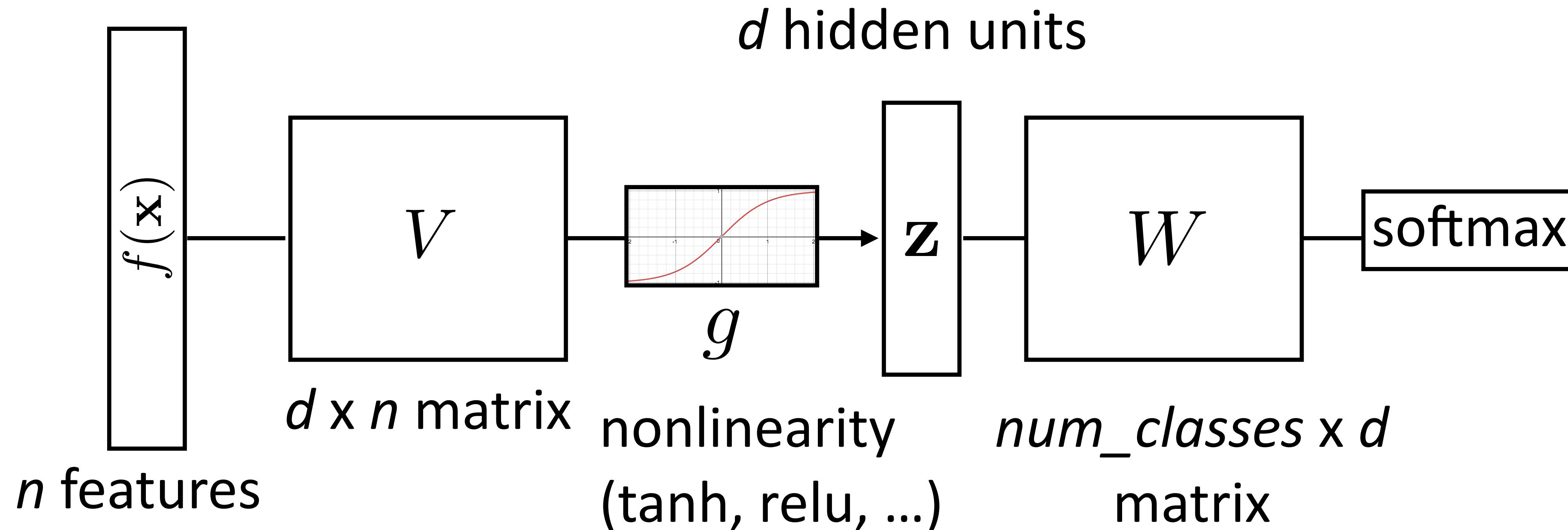
Neural Networks for Classification

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



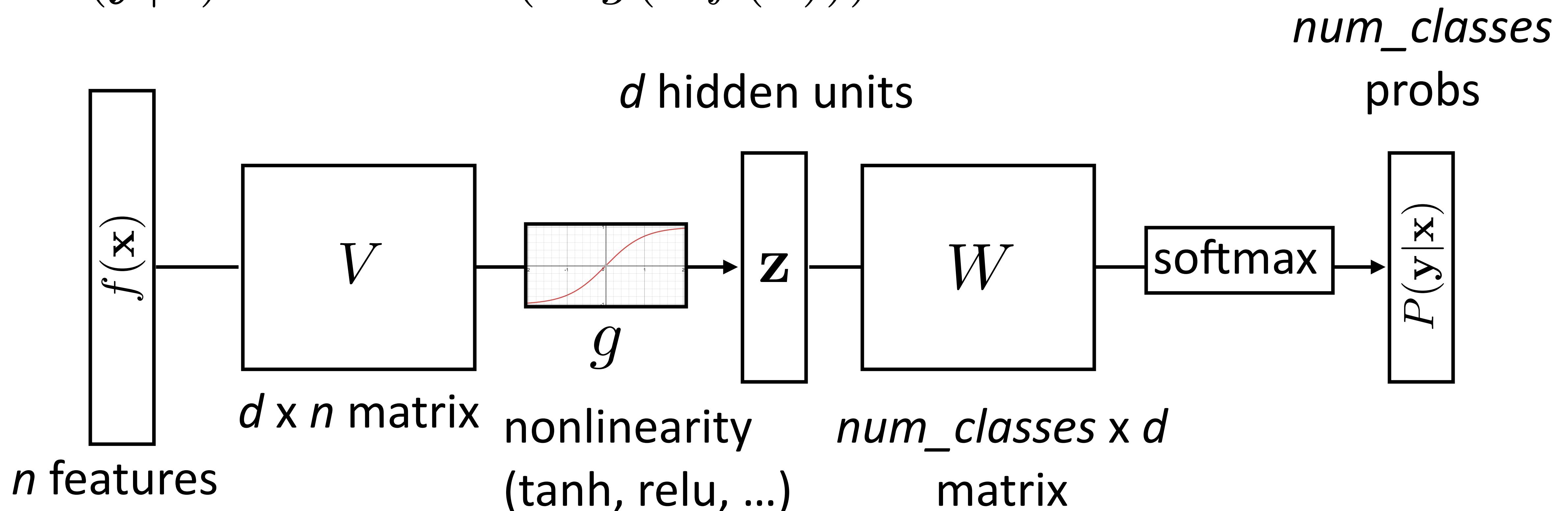
Neural Networks for Classification

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



Neural Networks for Classification

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



Training Neural Networks

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

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

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

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

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

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

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

Computing Gradients

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

- ▶ Gradient with respect to \mathbf{W}

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

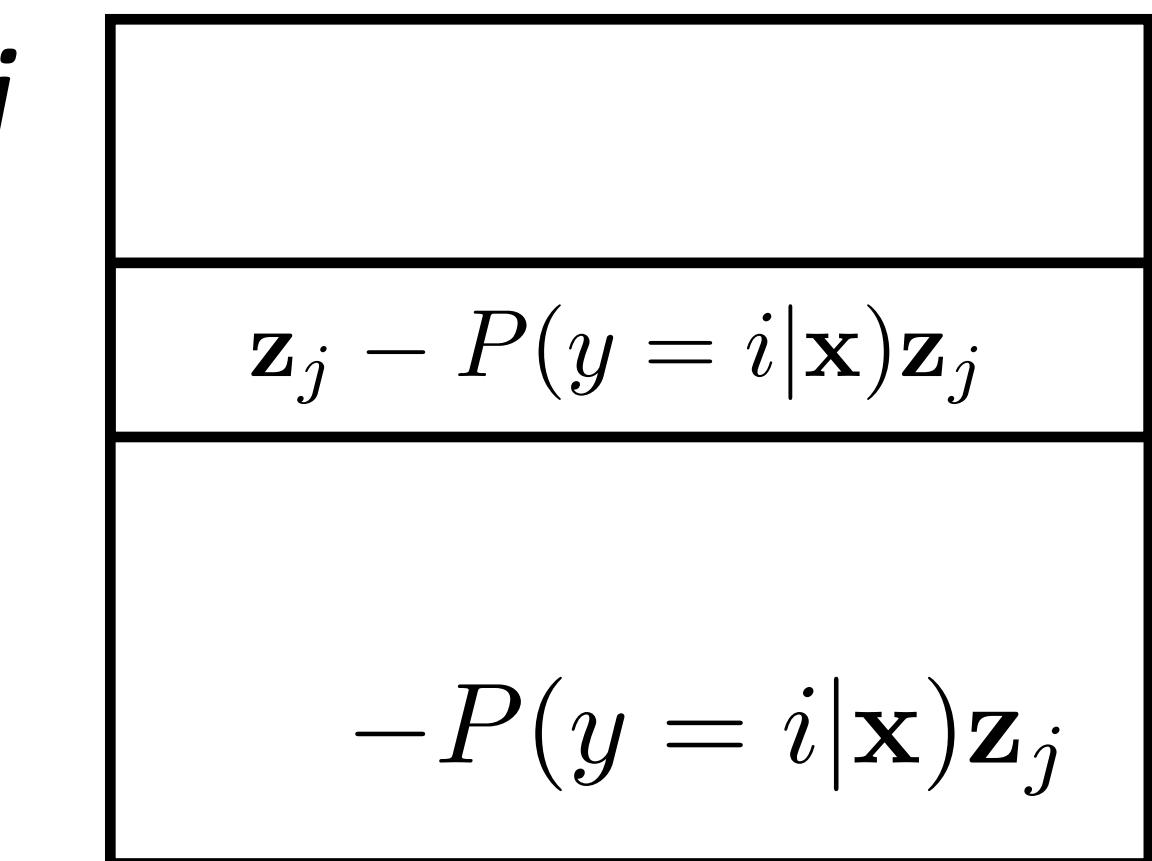
Computing Gradients

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

- Gradient with respect to W

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

W_j



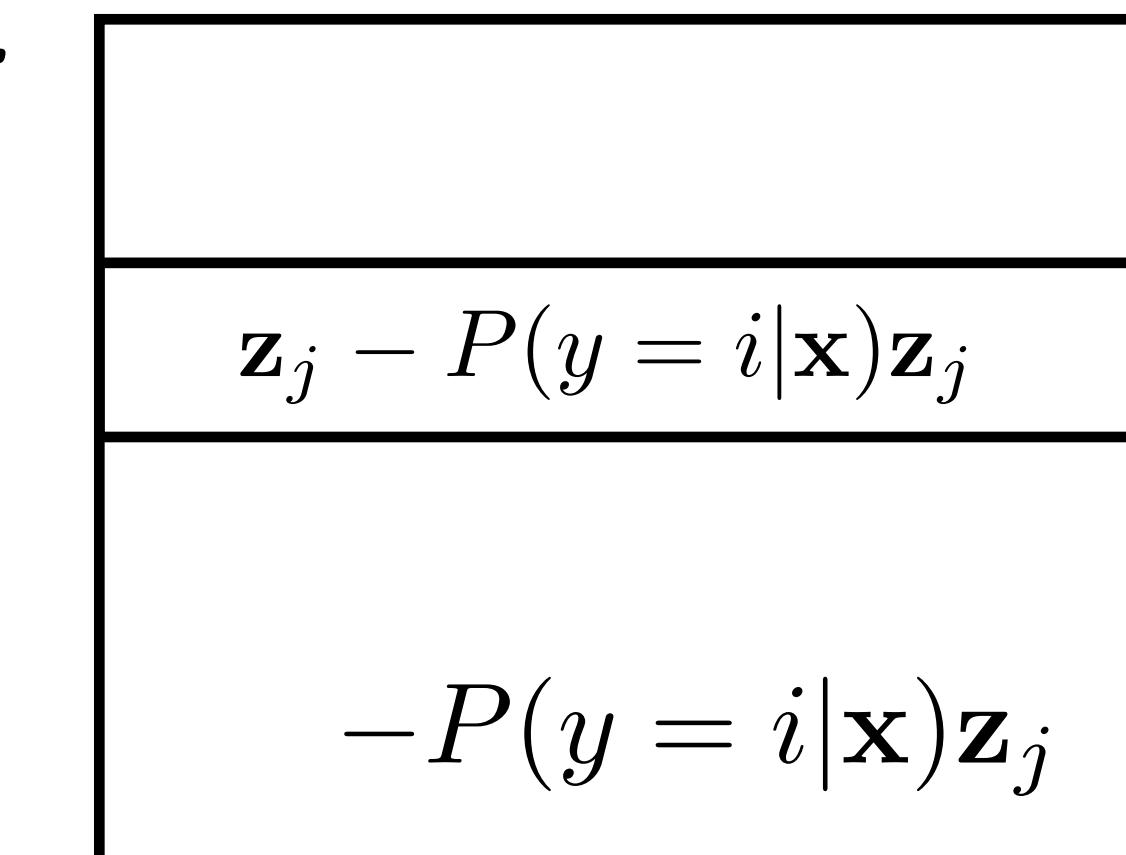
Computing Gradients

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

- Gradient with respect to W

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

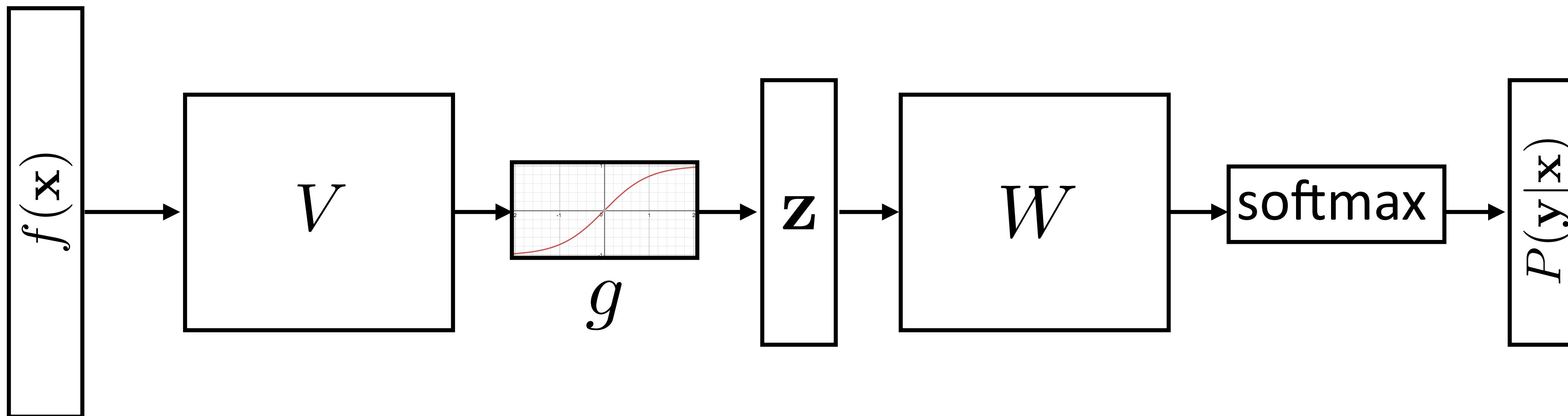
W_j



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

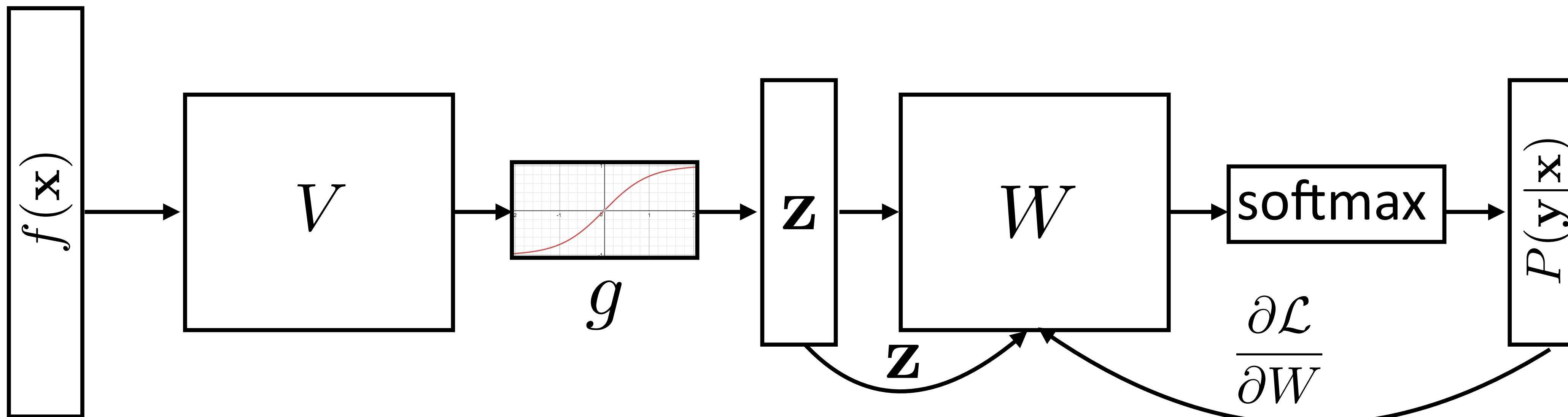
Neural Networks for Classification

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



Neural Networks for Classification

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



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

Computing Gradients: Backpropagation

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

Activations at
hidden layer

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

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

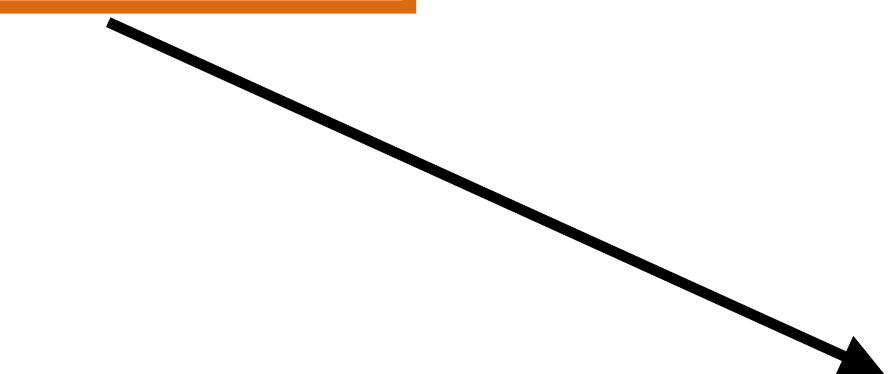
Computing Gradients: Backpropagation

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

Activations at
hidden layer

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

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



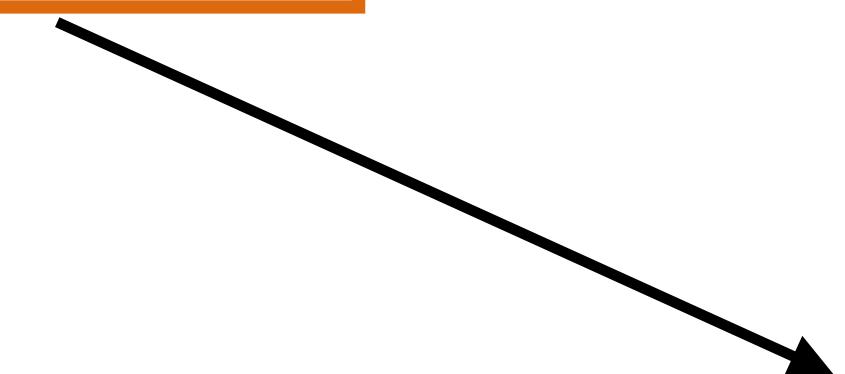
Computing Gradients: Backpropagation

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

Activations at
hidden layer

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

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



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

dim = num_classes

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...]

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

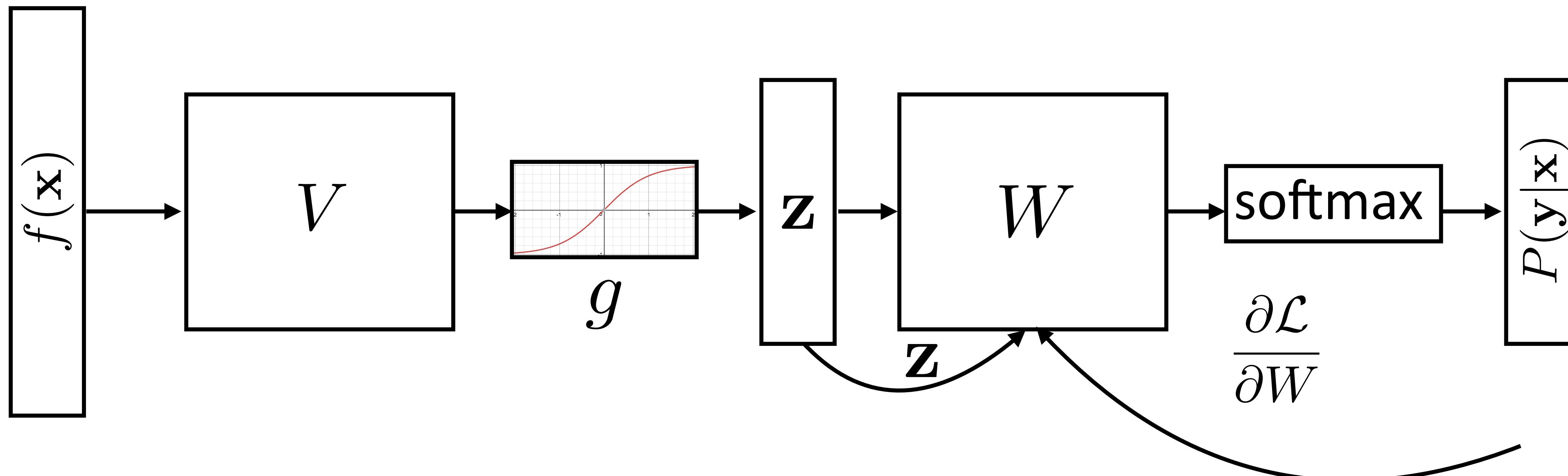
dim = num_classes

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

dim = d

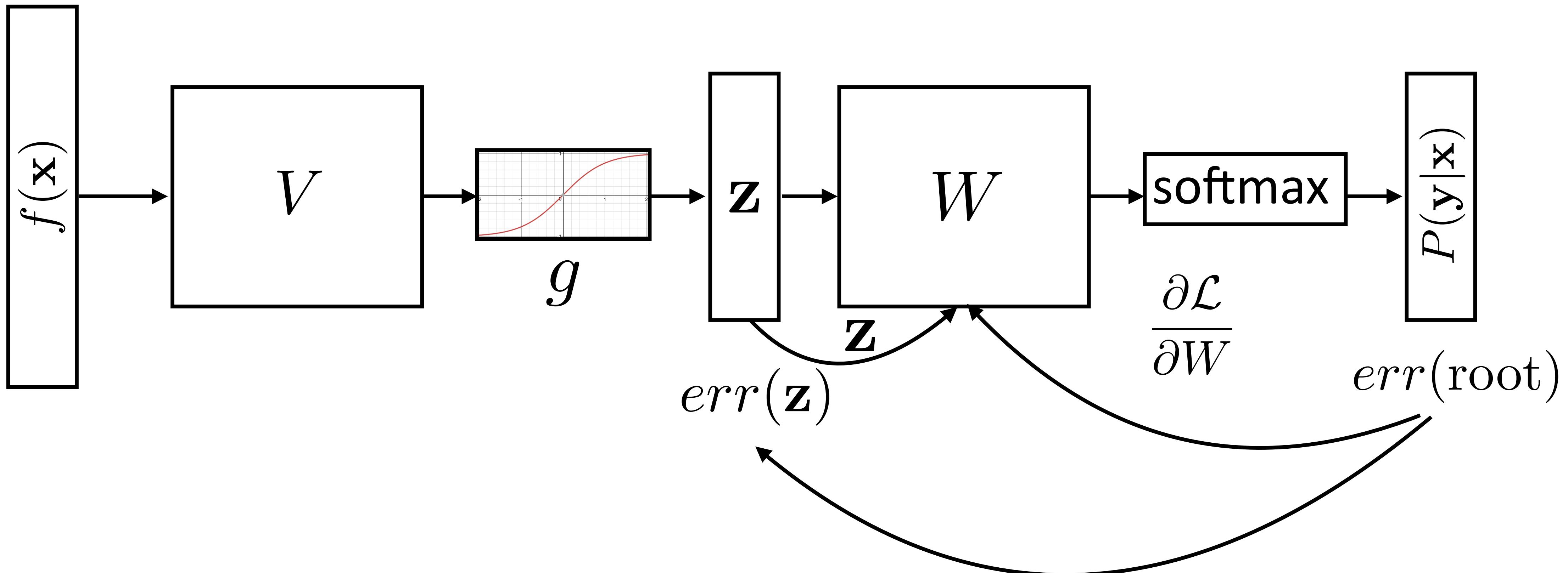
Backpropagation: Picture

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



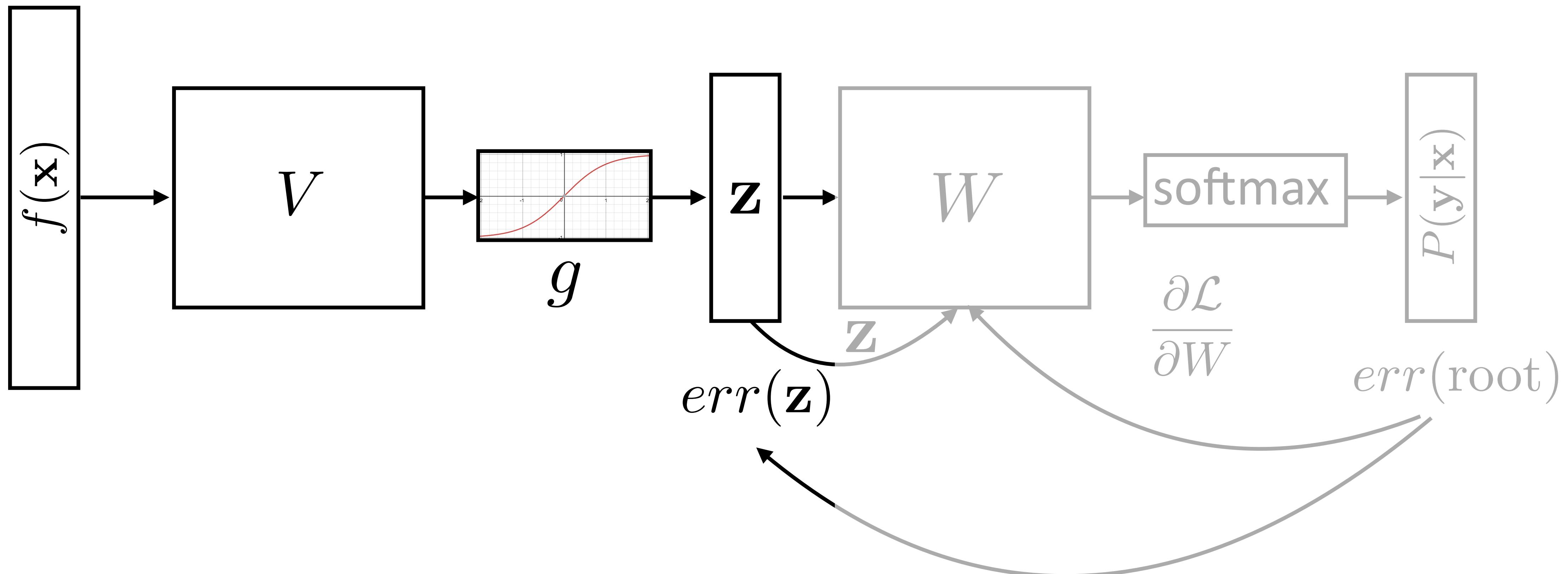
Backpropagation: Picture

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



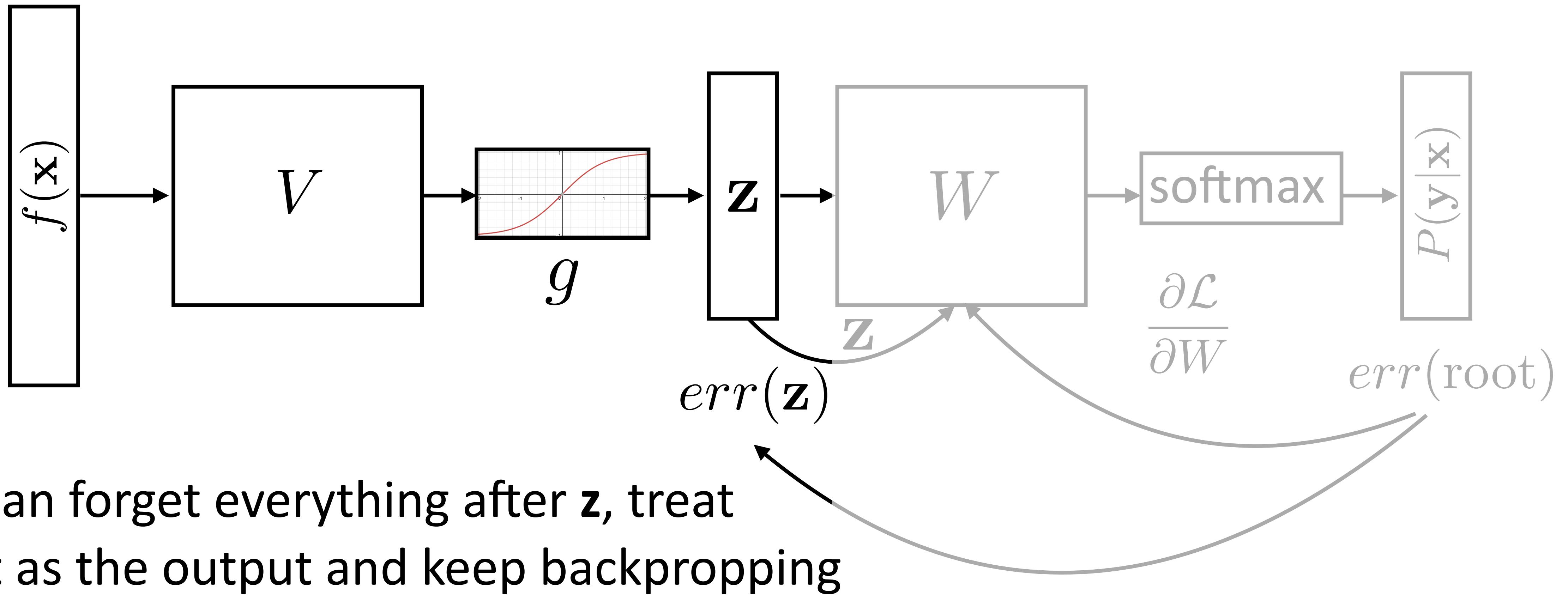
Backpropagation: Picture

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



Backpropagation: Picture

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



Computing Gradients: Backpropagation

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

$\mathbf{z} = g(Vf(\mathbf{x}))$
Activations at
hidden layer

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

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

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

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

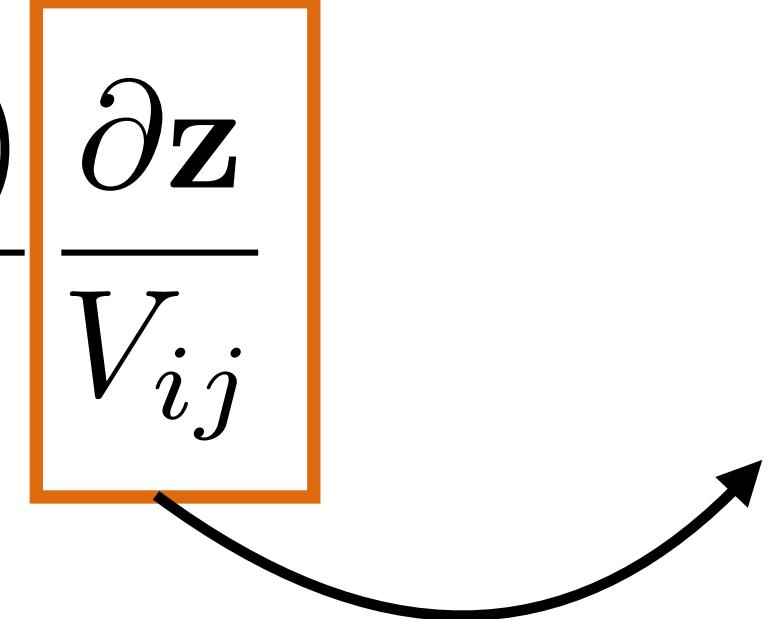

Computing Gradients: Backpropagation

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

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

Activations at
hidden layer

- Gradient with respect to V : apply the chain rule

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


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

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

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

The diagram illustrates the chain rule for backpropagation. It shows two equations side-by-side. The first equation is $\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial V_{ij}} = \frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial V_{ij}}$. An orange box highlights the term $\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}}$. The second equation is $\frac{\partial \mathbf{z}}{\partial V_{ij}} = \frac{\partial g(\mathbf{a})}{\partial \mathbf{a}} \frac{\partial \mathbf{a}}{\partial V_{ij}}$. A red box highlights the term $\frac{\partial g(\mathbf{a})}{\partial \mathbf{a}}$. A curved arrow points from the highlighted term in the first equation to the highlighted term in the second equation, indicating they are multiplied together.

Computing Gradients: Backpropagation

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

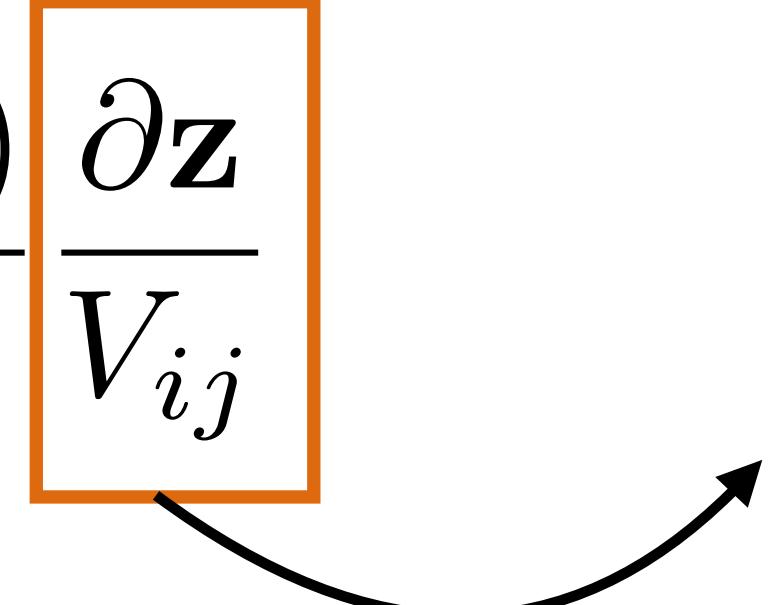
$$\mathbf{z} = g(Vf(\mathbf{x}))$$

Activations at hidden layer

- Gradient with respect to V : apply the chain rule

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

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



- First term: gradient of nonlinear activation function at \mathbf{a} (depends on current value)

Computing Gradients: Backpropagation

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

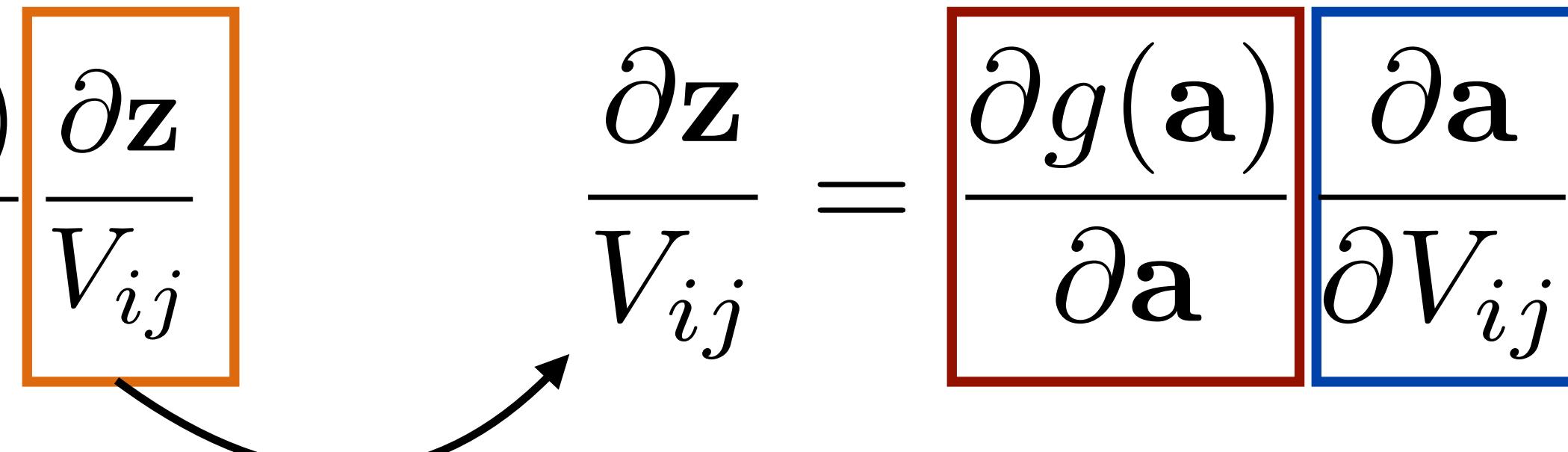
$$\mathbf{z} = g(Vf(\mathbf{x}))$$

Activations at hidden layer

- Gradient with respect to V : apply the chain rule

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

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



- First term: gradient of nonlinear activation function at \mathbf{a} (depends on current value)
- Second term: gradient of linear function

Computing Gradients: Backpropagation

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

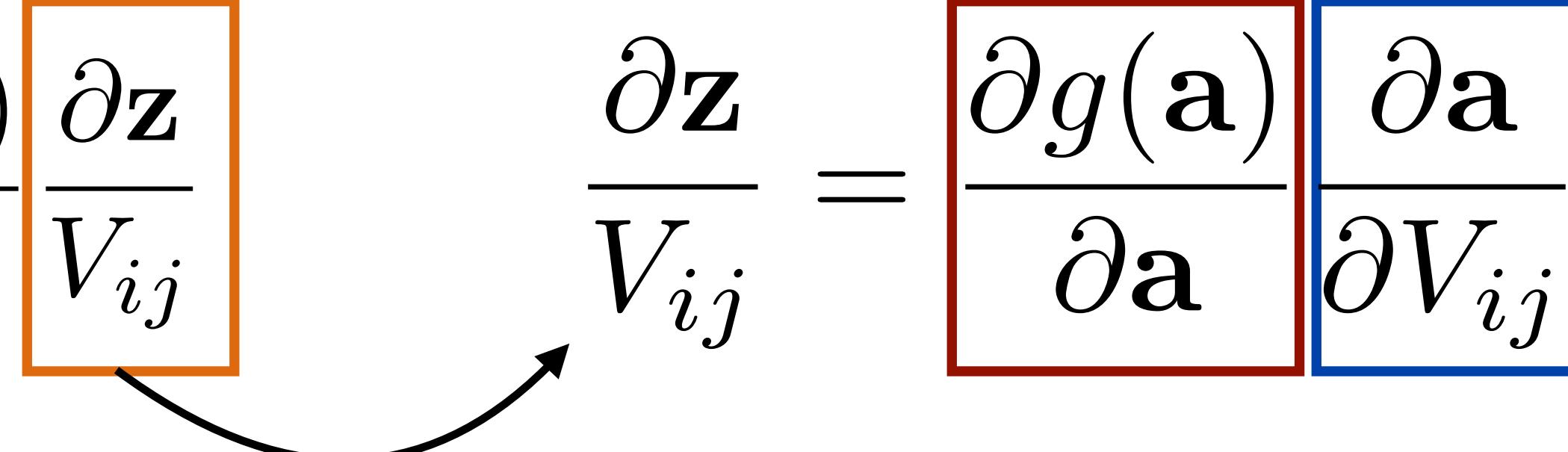
$$\mathbf{z} = g(Vf(\mathbf{x}))$$

Activations at hidden layer

- Gradient with respect to V : apply the chain rule

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

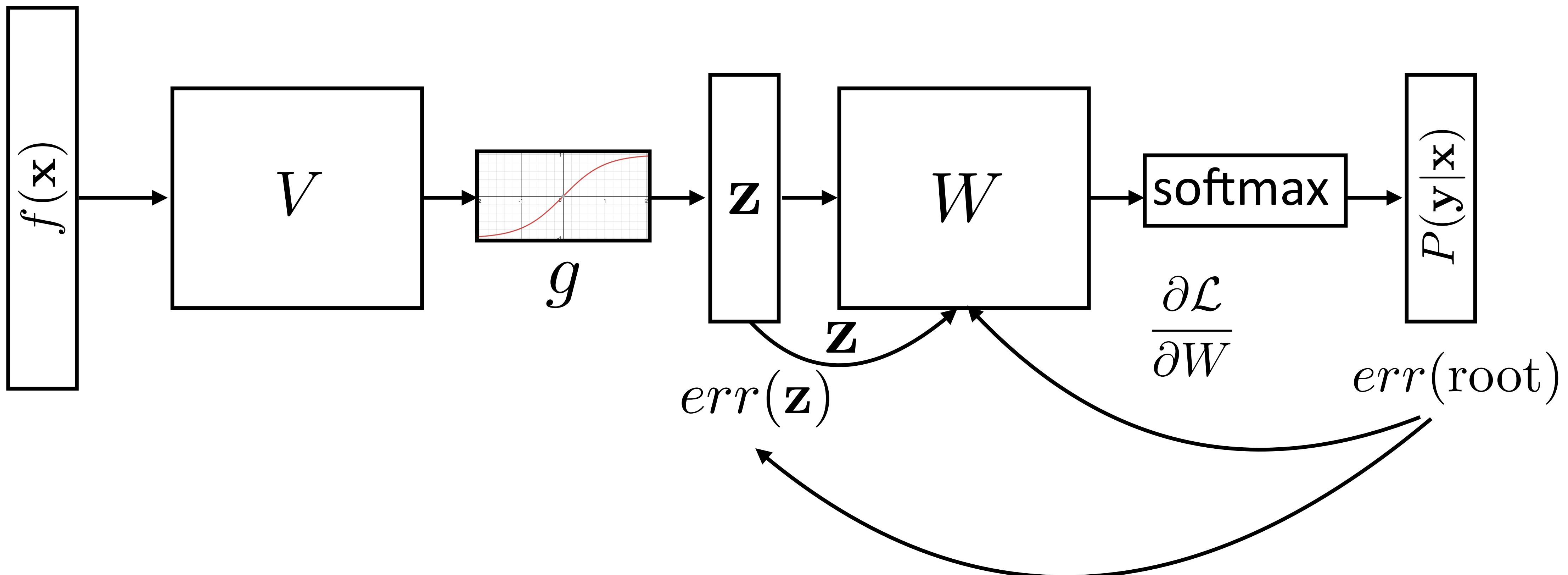
$\mathbf{a} = Vf(\mathbf{x})$



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

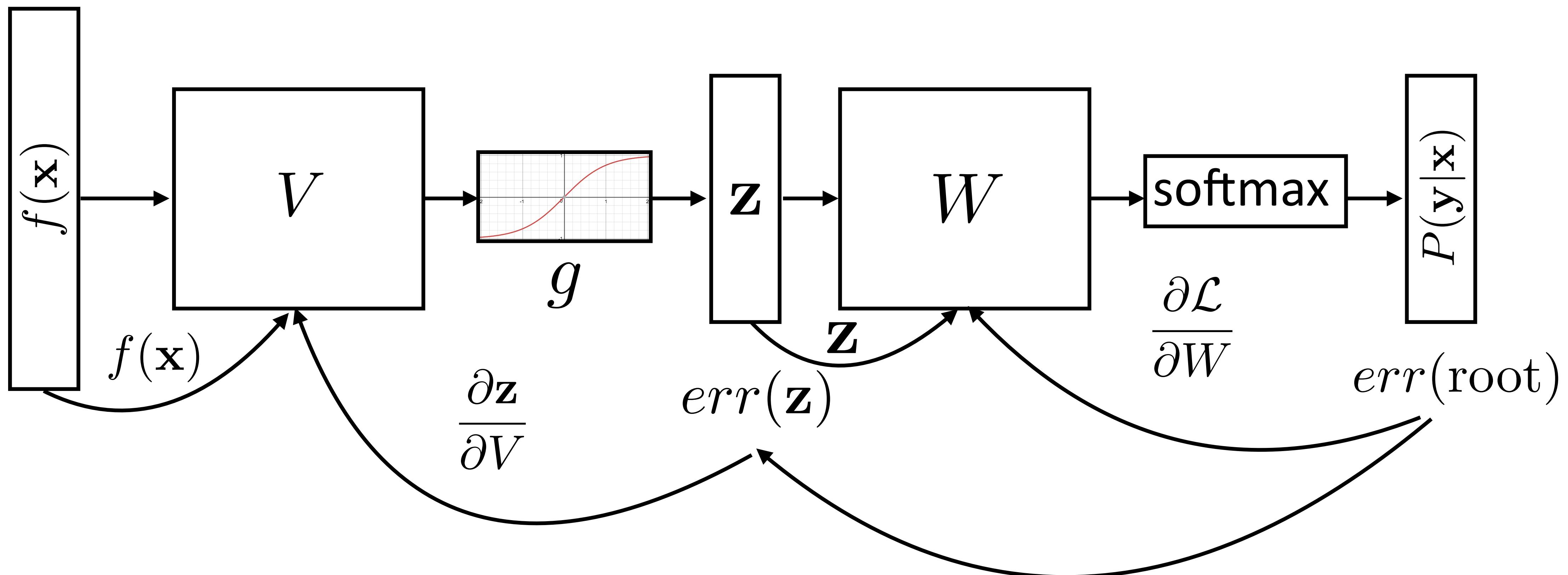
Backpropagation: Picture

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



Backpropagation: Picture

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



Backpropagation

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

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)

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)

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)

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)

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

Backpropagation: Takeaways

- ▶ Gradients of output weights W are easy to compute – looks like logistic regression with hidden layer \mathbf{z} as feature vector

Backpropagation: Takeaways

- ▶ Gradients of output weights W are easy to compute – looks like logistic regression with hidden layer \mathbf{z} as feature vector
- ▶ Can compute derivative of loss with respect to \mathbf{z} to form an “error signal” for backpropagation

Backpropagation: Takeaways

- ▶ Gradients of output weights W are easy to compute – looks like logistic regression with hidden layer \mathbf{z} as feature vector
- ▶ Can compute derivative of loss with respect to \mathbf{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

Backpropagation: Takeaways

- ▶ Gradients of output weights W are easy to compute – looks like logistic regression with hidden layer \mathbf{z} as feature vector
- ▶ Can compute derivative of loss with respect to \mathbf{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

NLP with Feedforward Networks

- ▶ Part-of-speech tagging with FFNNs

??

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

NLP with Feedforward Networks

- ▶ Part-of-speech tagging with FFNNs

??

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

NLP with Feedforward Networks

- ▶ Part-of-speech tagging with FFNNs

??

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

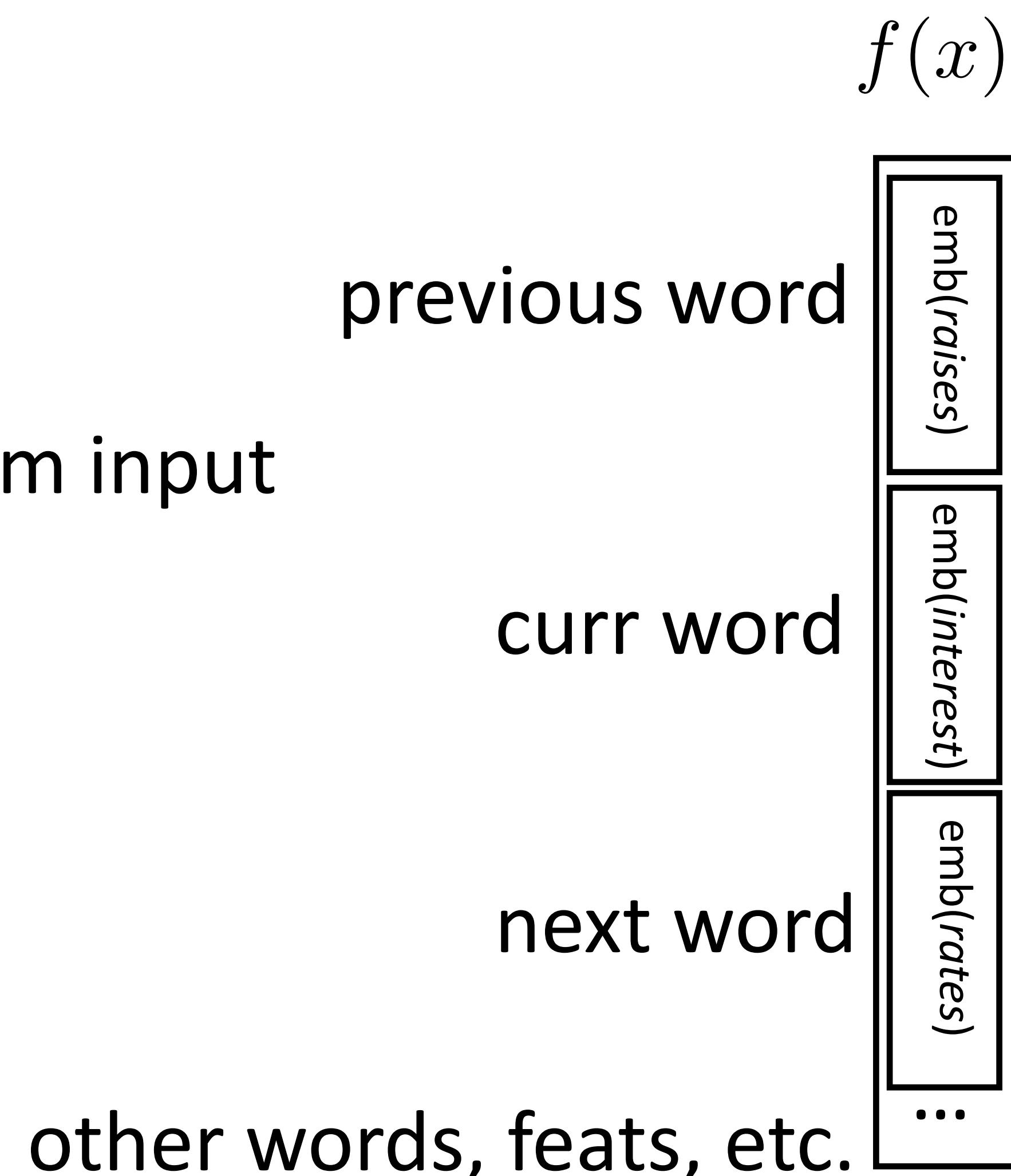
- ▶ Word embeddings for each word form input

NLP with Feedforward Networks

- ▶ Part-of-speech tagging with FFNNs

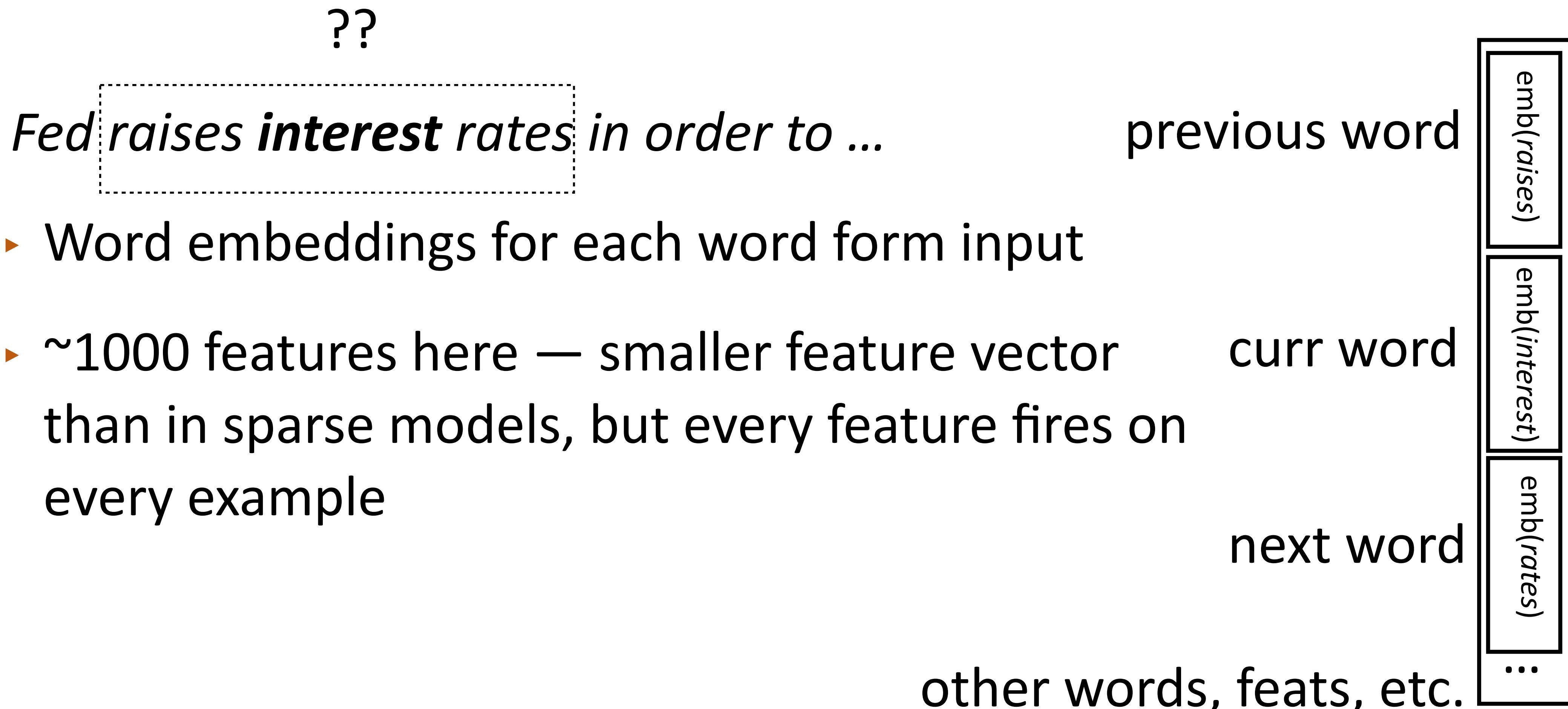
??
Fed raises interest rates in order to ...

- ▶ Word embeddings for each word form input



NLP with Feedforward Networks

- ▶ Part-of-speech tagging with FFNNs

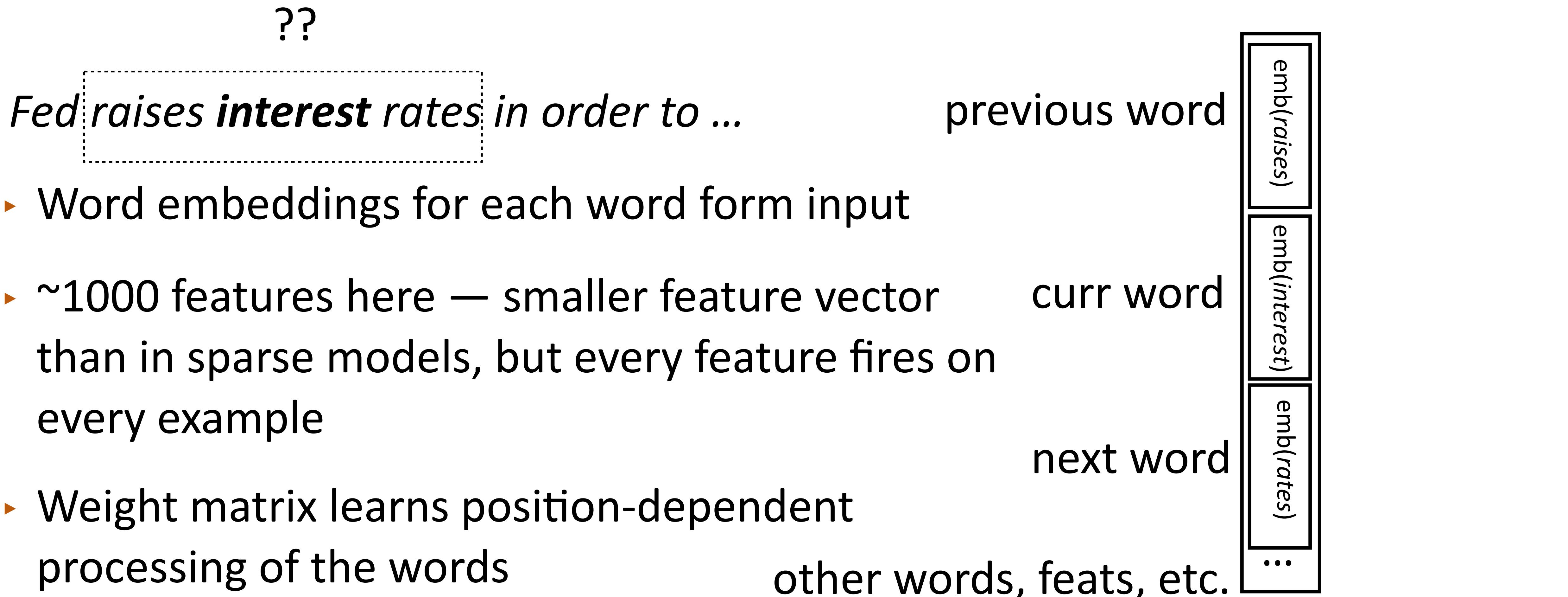


- ▶ Word embeddings for each word form input

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

NLP with Feedforward Networks

- ▶ Part-of-speech tagging with FFNNs

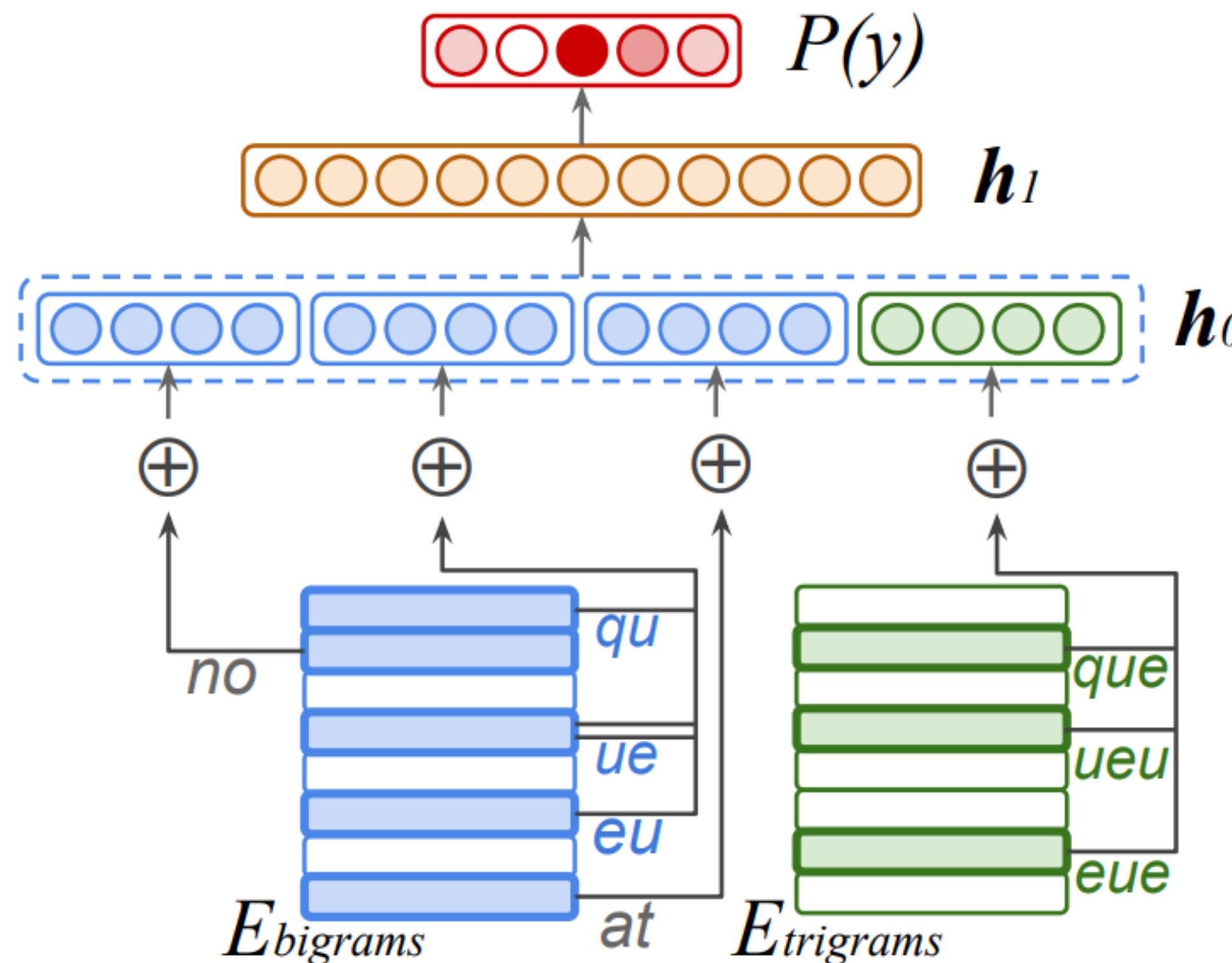


- ▶ Word embeddings for each word form input

- ▶ ~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

NLP with Feedforward Networks



There was no queue at the ...

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

NLP with Feedforward Networks

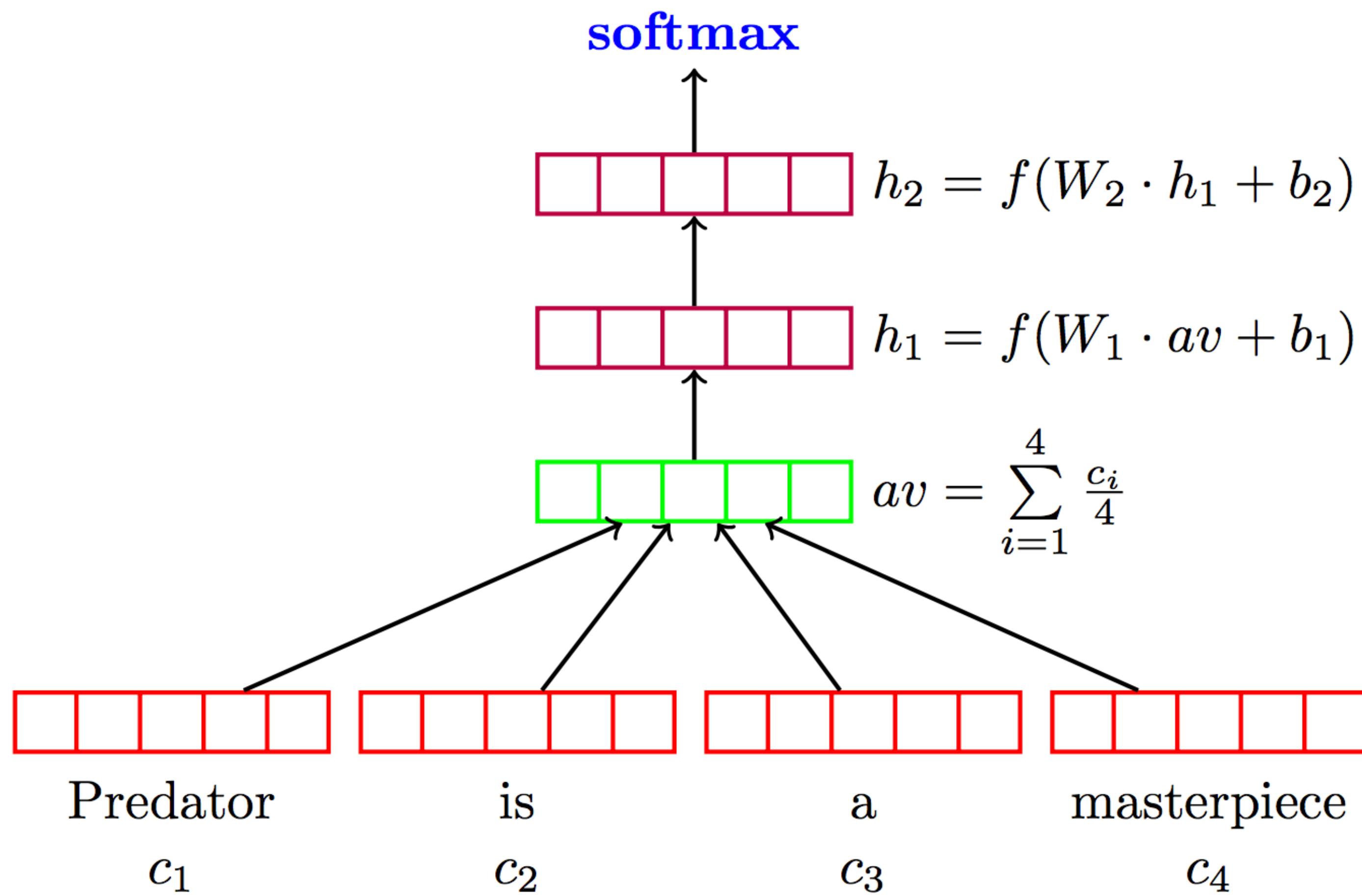
- ▶ Multilingual tagging results:

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

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	Iyyer et al. (2015)
Bag-of-words	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	—	Wang and Manning (2012)
Tree RNNs / CNNs / LSTMs	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	Kim (2014)
	WRRBM*	—	—	—	89.2	—	

Coreference Resolution

- ▶ Feedforward networks identify coreference arcs

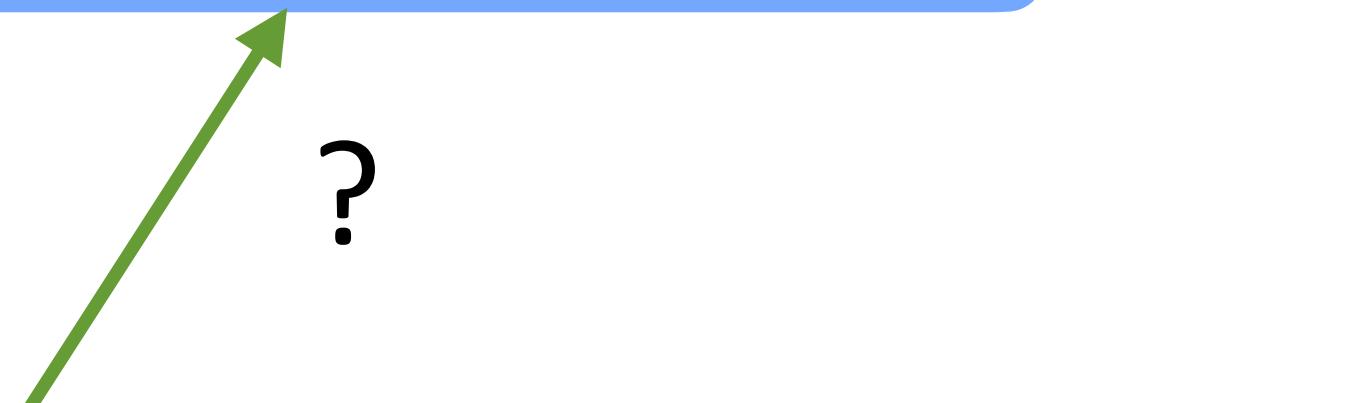
Coreference Resolution

- ▶ Feedforward networks identify coreference arcs

President Obama signed...

He later gave a speech...

?



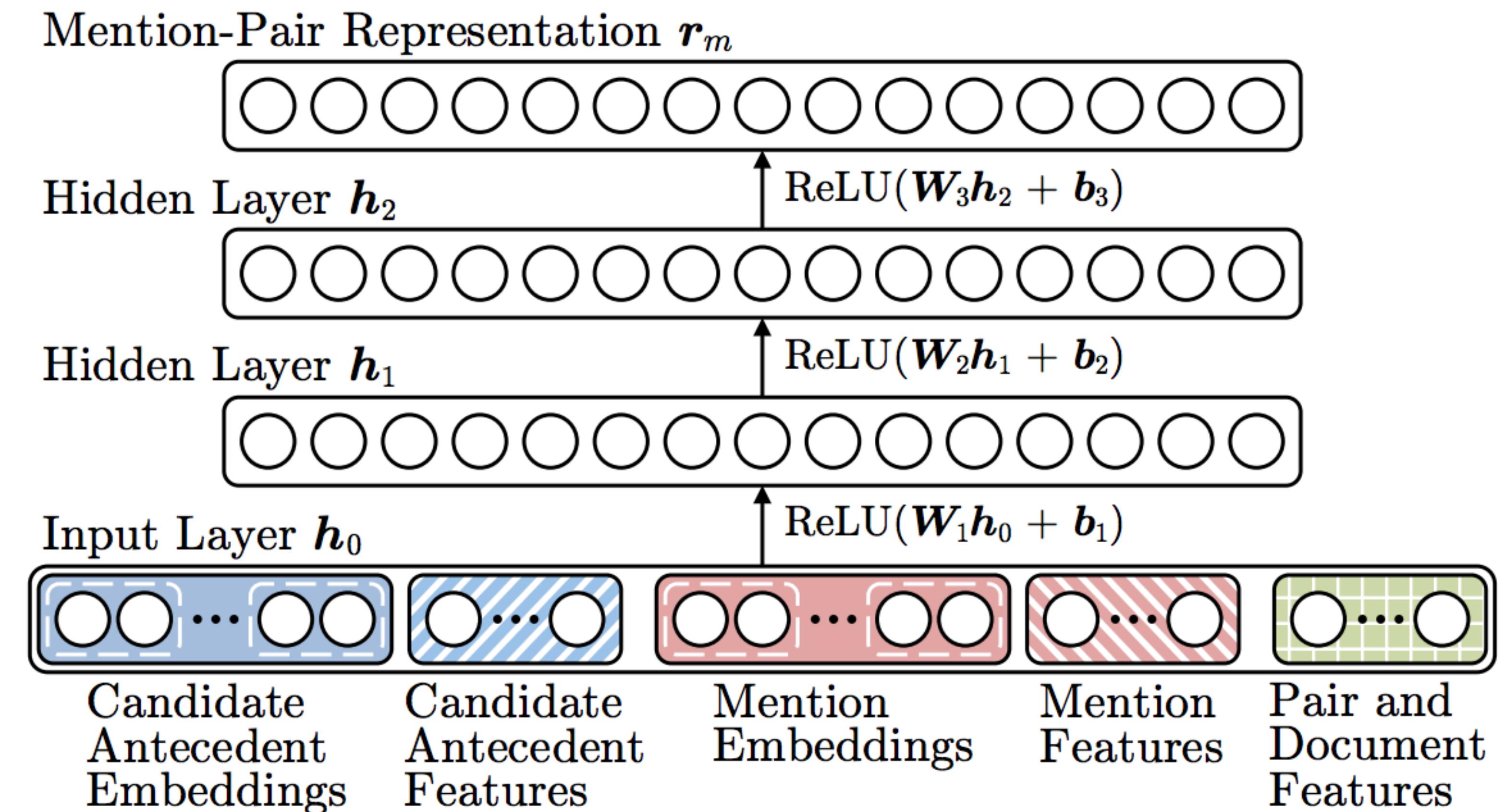
Coreference Resolution

- Feedforward networks identify coreference arcs

President Obama signed...

?

He later gave a speech...



Implementation Details

Computation Graphs

- ▶ Computing gradients is hard!

Computation Graphs

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

Computation Graphs

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

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

Computation Graphs

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

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

- ▶ Computation is now something we need to reason about symbolically

Computation Graphs

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

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

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

```
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

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

Computation Graphs in Pytorch

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

```
ffnn = FFNN()
```

Computation Graphs in Pytorch

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

```
ffnn = FFNN()
def make_update(input, gold_label):
```

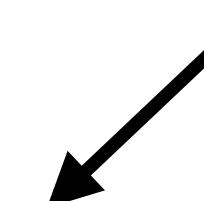
Computation Graphs in Pytorch

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

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

```
ffnn = FFNN()
```

```
def make_update(input, gold_label):
```

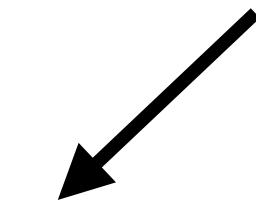


Computation Graphs in Pytorch

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

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
```

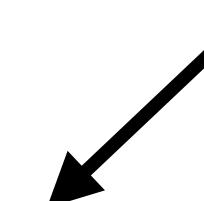


Computation Graphs in Pytorch

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

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

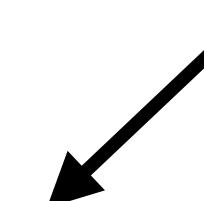
```
ffnn = FFNN()
def make_update(input, gold_label):
    ffnn.zero_grad() # clear gradient variables
    probs = ffnn.forward(input)
```



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])

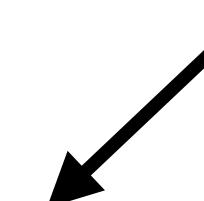


```
ffnn = FFNN()
def make_update(input, gold_label):
    ffnn.zero_grad() # clear gradient variables
    probs = ffnn.forward(input)
    loss = torch.neg(torch.log(probs)).dot(gold_label)
```

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])



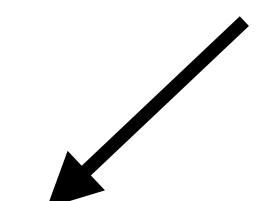
```
ffnn = FFNN()

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()
```

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

Training a Model

Define a computation graph

Training a Model

Define a computation graph

For each epoch:

Training a Model

Define a computation graph

For each epoch:

For each batch of data:

Training a Model

Define a computation graph

For each epoch:

For each batch of data:

Compute loss on batch

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

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

Batching

- ▶ Batching data gives speedups due to more efficient matrix operations
- ▶ Need to make the computation graph process a batch at the same time

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)
```

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]
```

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))
    ...

```

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