THE OHIO STATE UNIVERSITY

CSE 5525: Foundations of Speech and Language Processing

Neural Networks + Word Embeddings

Huan Sun (CSE@OSU)

Many thanks to Prof. Greg Durrett @ UT Austin for sharing his slides.

This Lecture

- Neural network history (for offline reading)

- Neural network basics (for offline reading)

- **Feedforward neural networks + backpropagation**
- **Word embeddings**

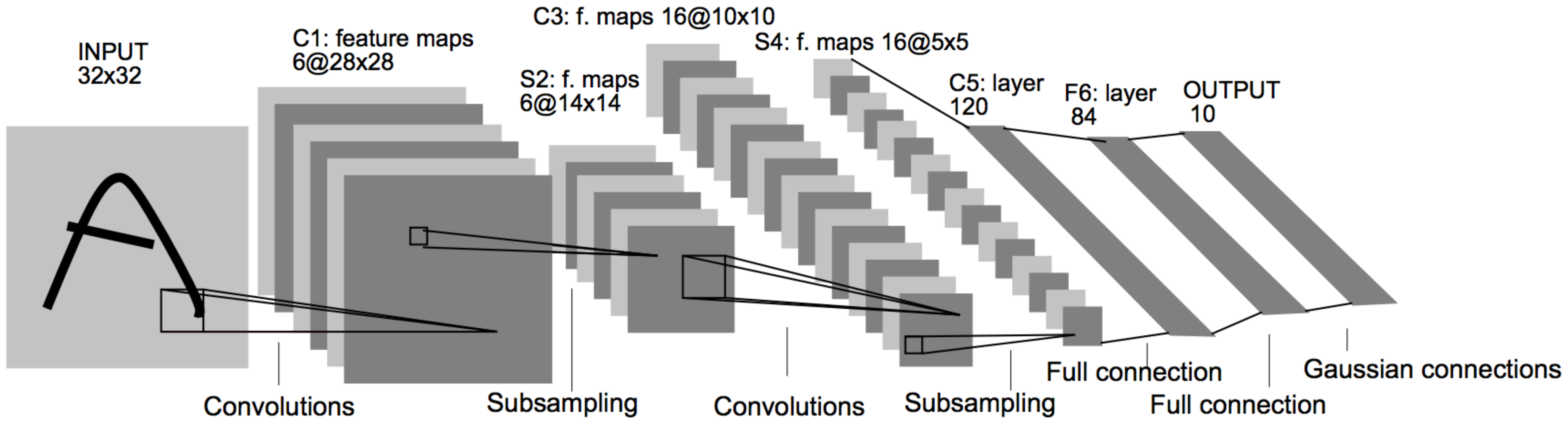
- Implementing neural networks (for offline reading & practice)

Neural Net History

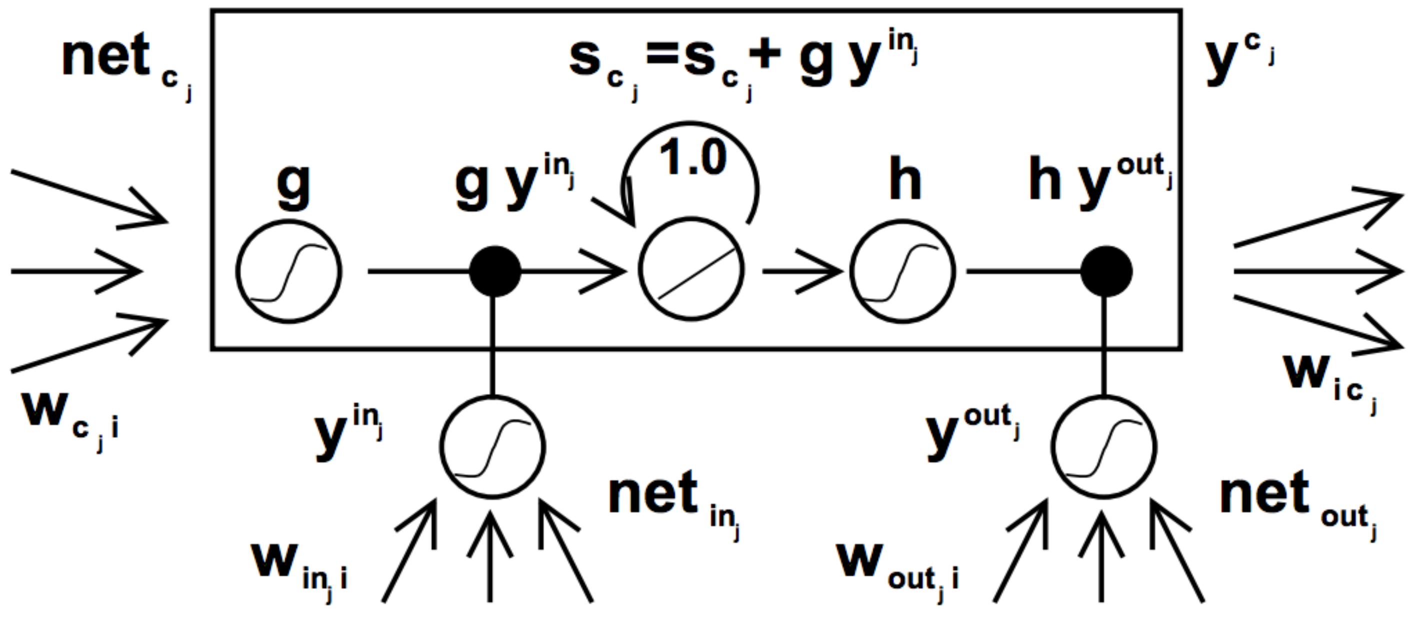
History: NN “dark ages”

Offline reading, if interested

- Convnets: applied to MNIST by LeCun in 1998



- LSTMs: Hochreiter and Schmidhuber (1997)

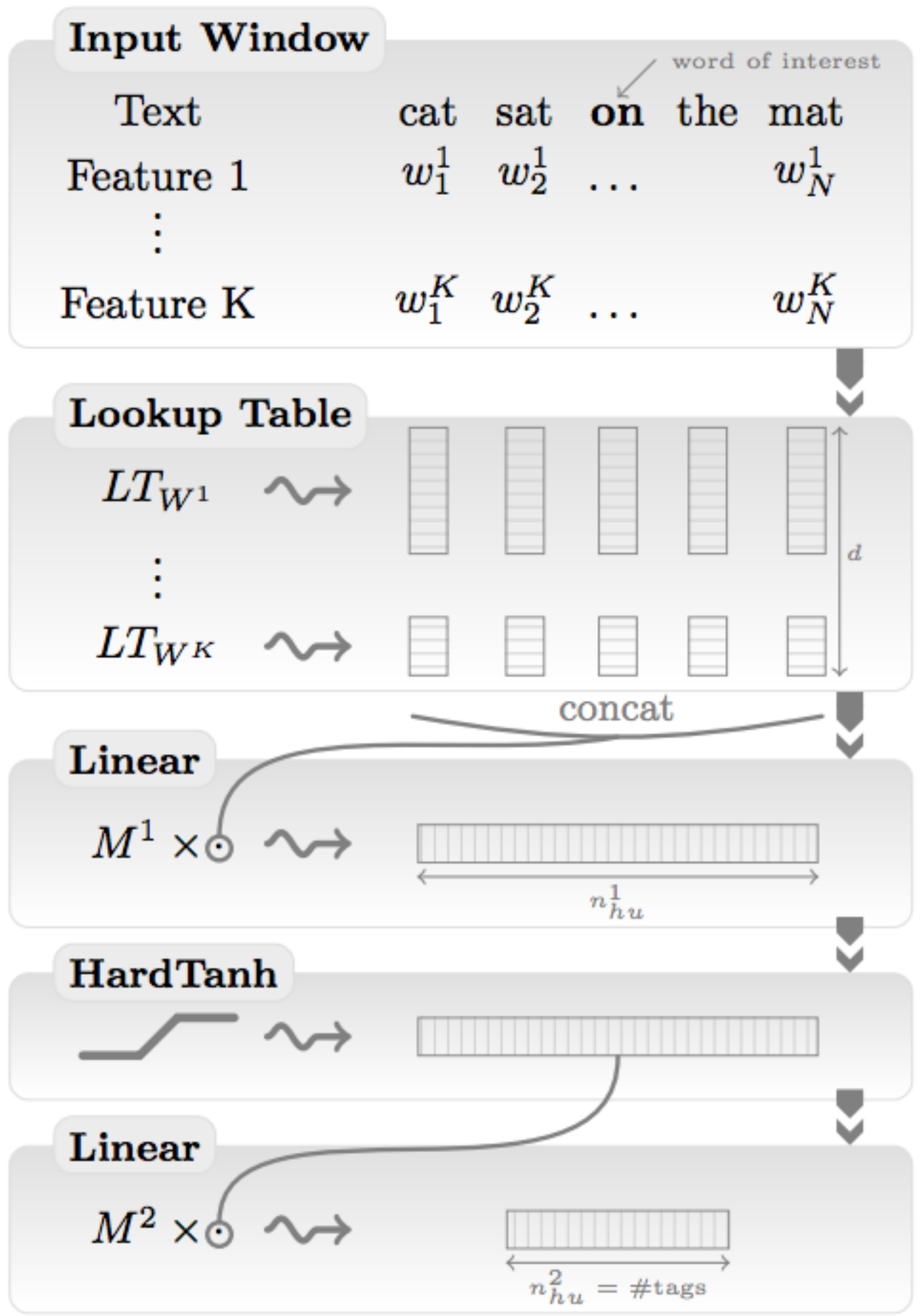


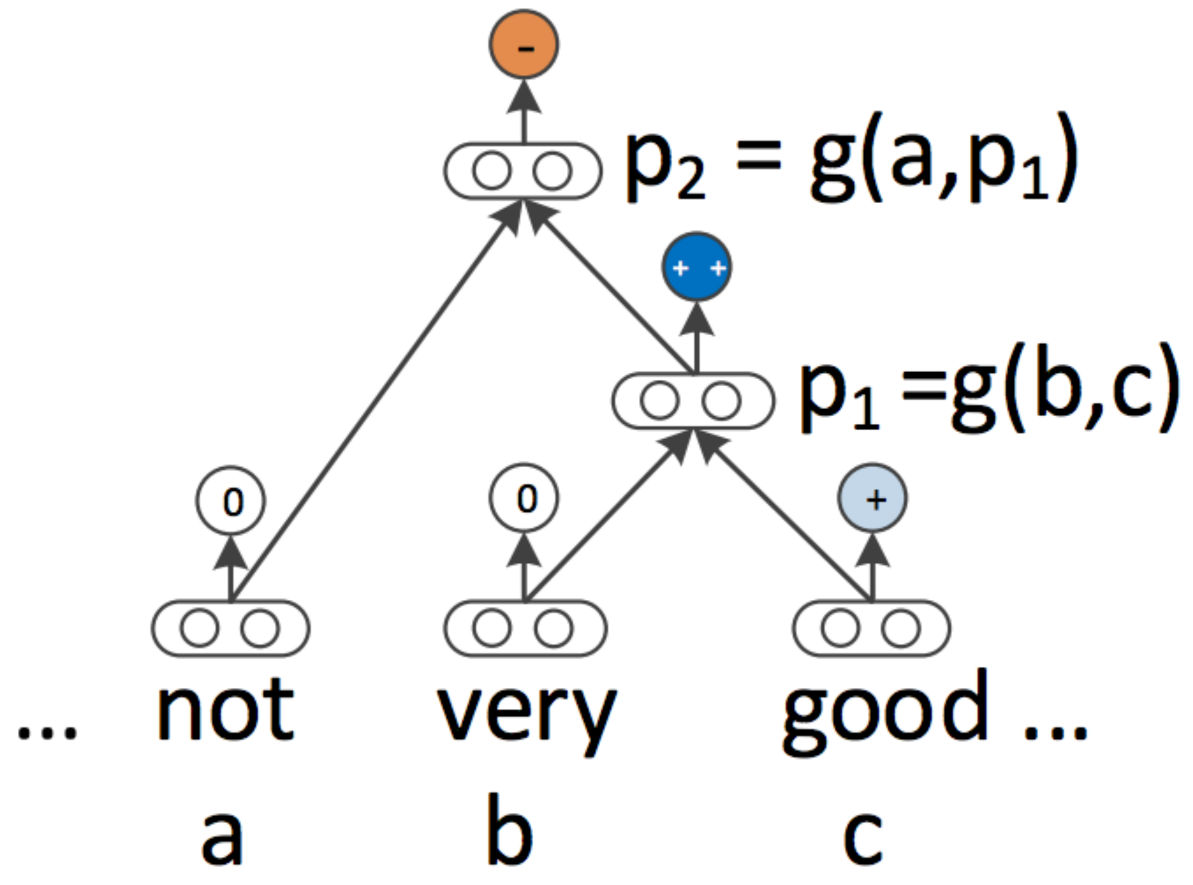
- Henderson (2003): neural shift-reduce parser, not SOTA

2008-2013: A glimmer of light…

Offline reading, if interested

- 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
- Socher 2011-2014: tree-structured RNNs working okay
- Krizhevskey et al. (2012): AlexNet for vision





2014: Stuff starts working

Offline reading, if interested

- Kim (2014) + Kalchbrenner et al. (2014): sentence classification / sentiment (convnets)
- Sutskever et al. + Bahdanau et al.: seq2seq for neural MT (LSTMs)
- Chen and Manning transition-based dependency parser (based on feedforward networks)
- 2015: explosion of neural nets for everything under the sun
- What made these work? **Data** (not as important as you might think), **optimization** (initialization, adaptive optimizers), **representation** (good word embeddings)

Neural Net Basics

Neural Networks

Offline reading, if interested

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

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

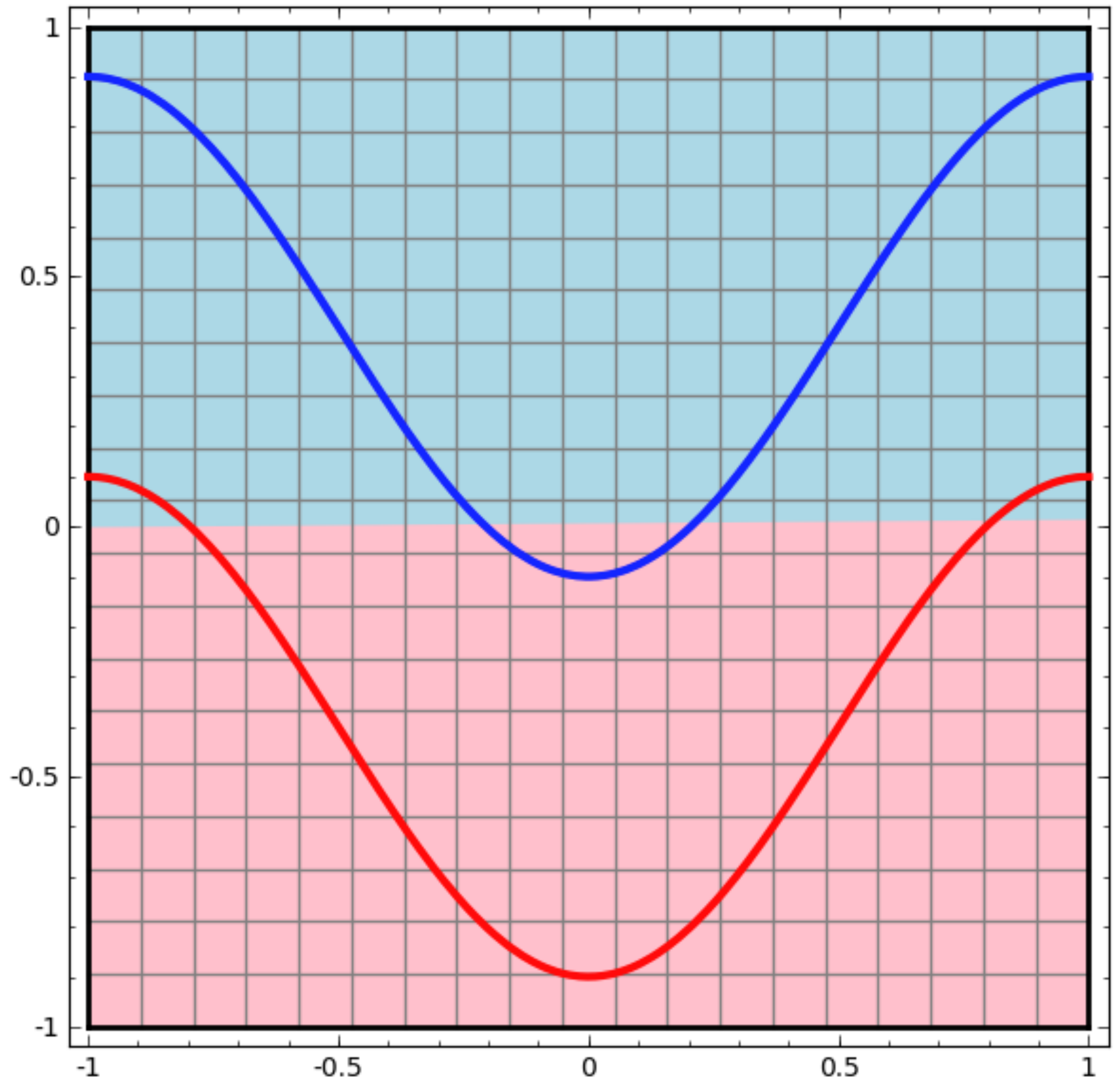
$$\mathbf{y} = \underset{\substack{\uparrow\\\text{Nonlinear}\\\text{transformation}}}{g}(\underset{\substack{\uparrow\\\text{Warp}\\\text{space}}}{\mathbf{W}}\mathbf{x} + \underset{\substack{\uparrow\\\text{Shift}}}{\mathbf{b}})$$

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

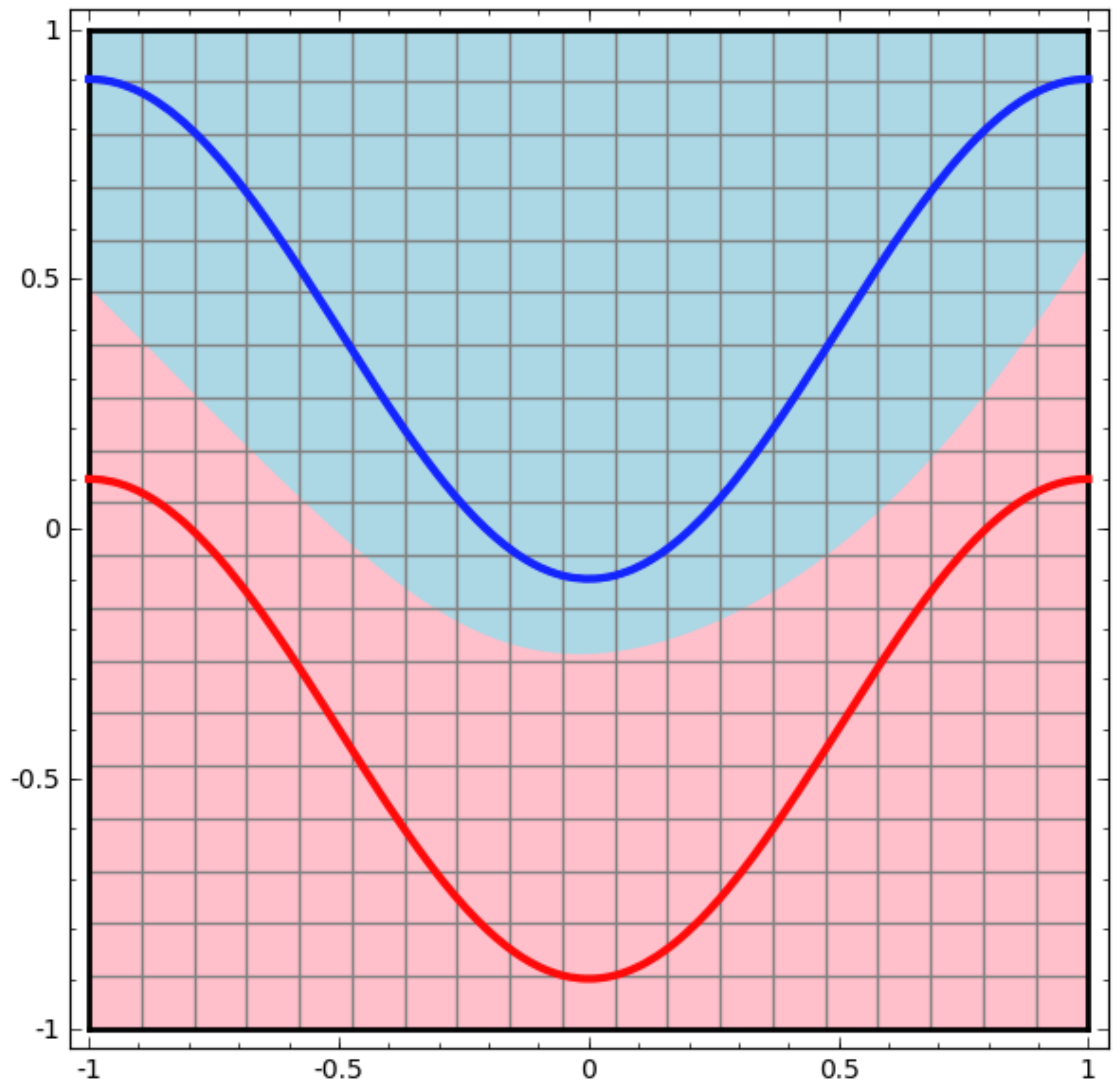
Neural Networks

Offline reading,
if interested

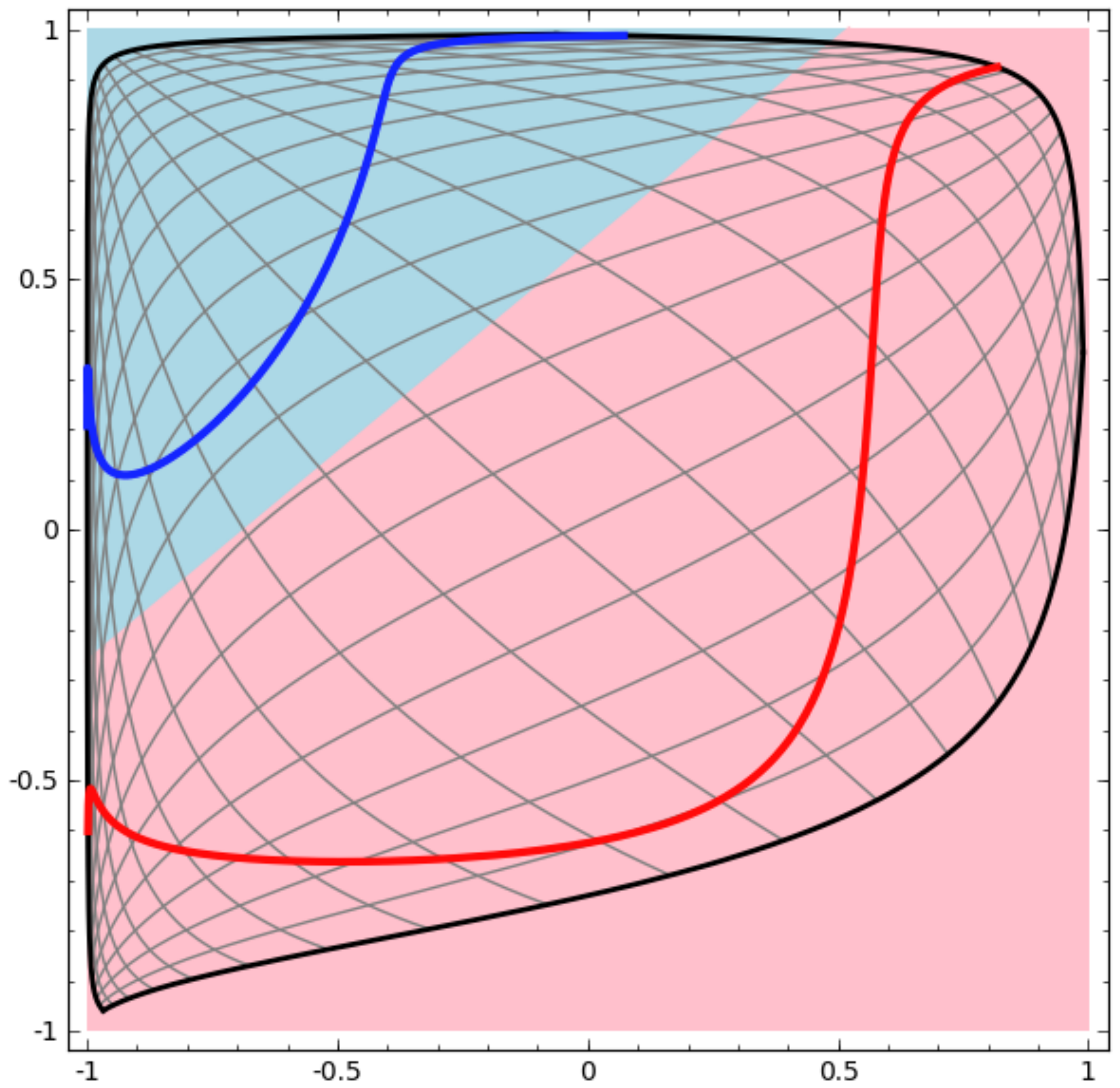
Linear classifier



Neural network



…possible because we transformed the space!



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

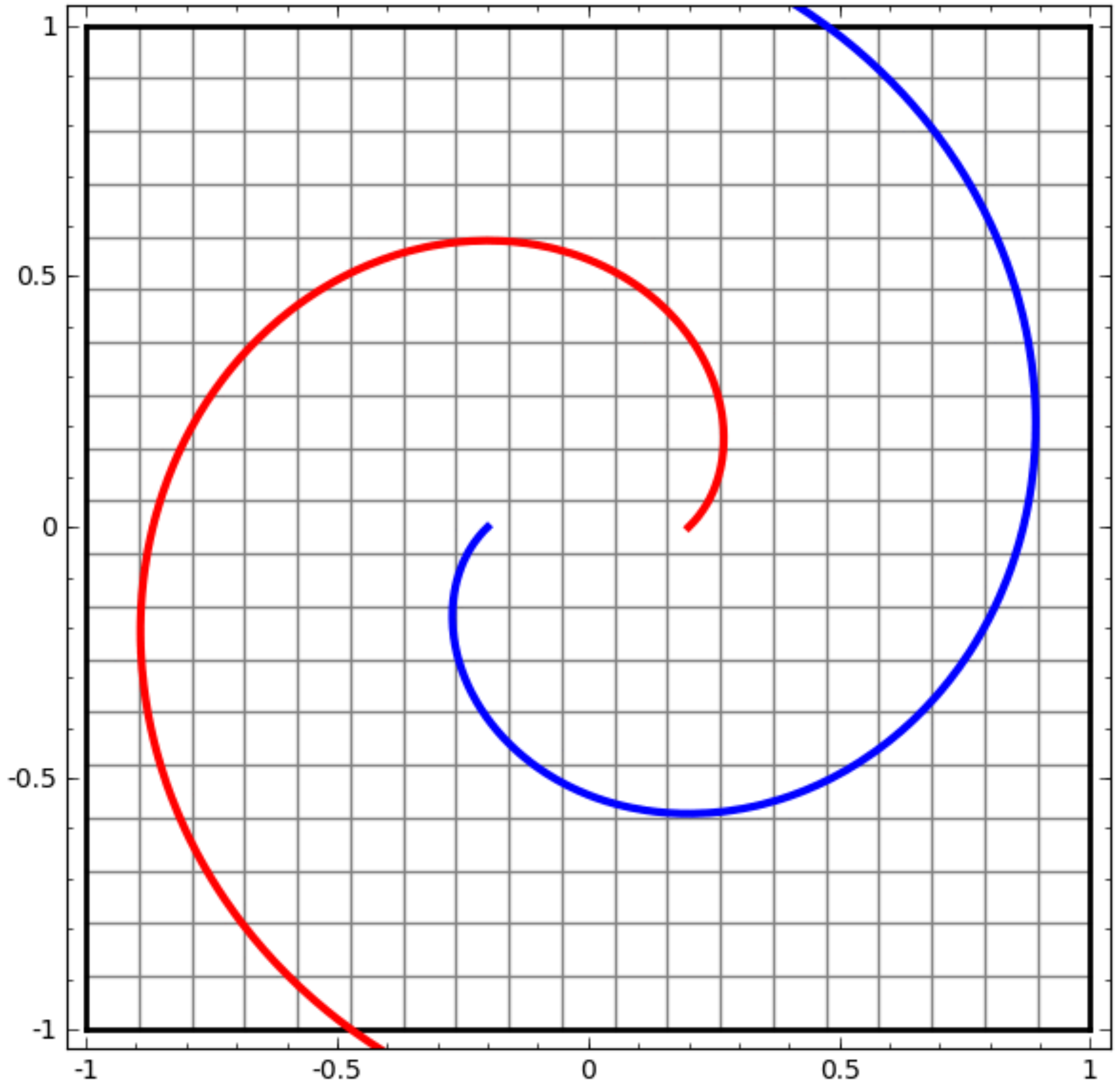
Deep Neural Networks

Offline reading, if interested

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

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

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



Check: what happens if no nonlinearity?

More powerful than basic linear models?

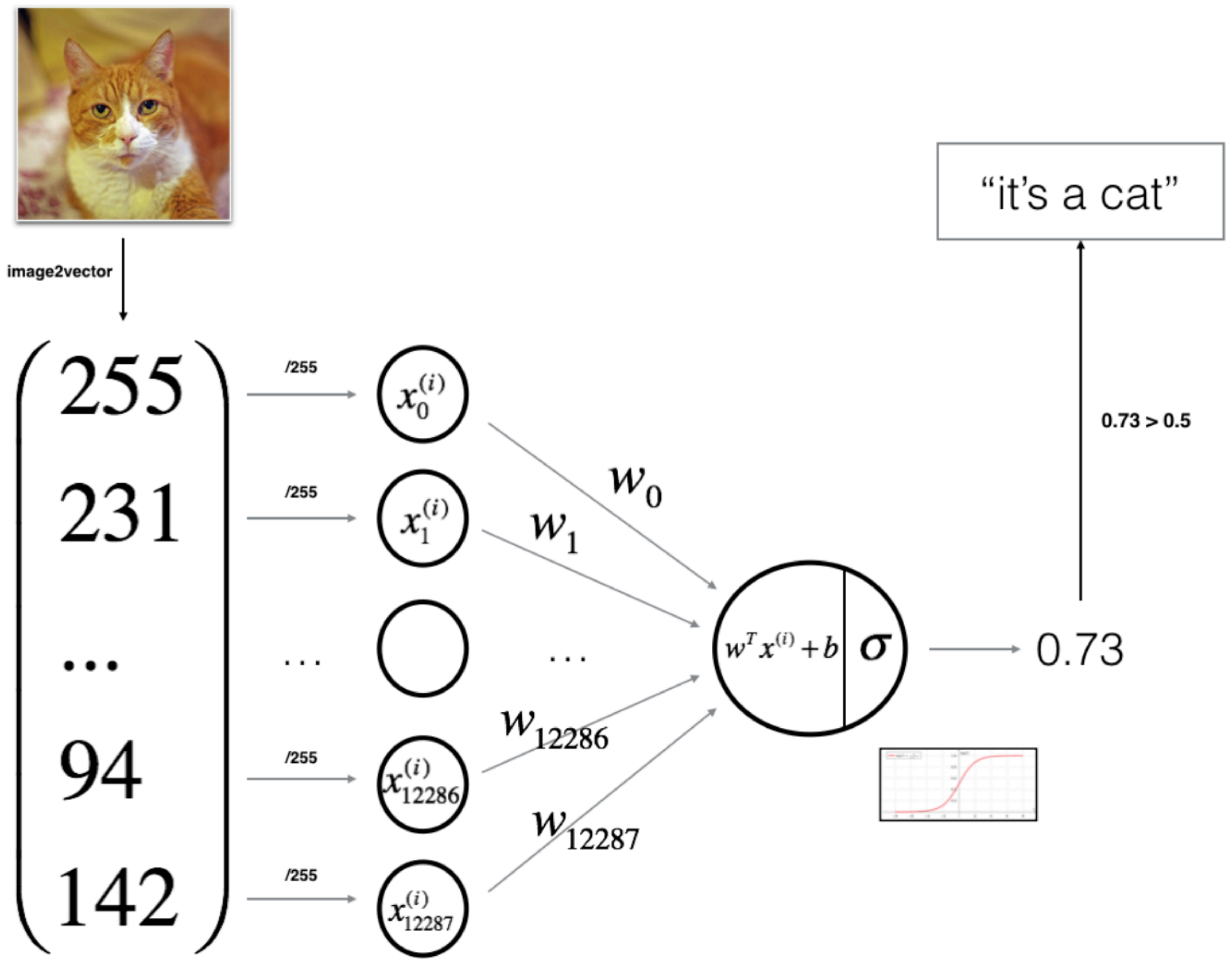
$$\mathbf{z} = \mathbf{V}(\mathbf{W}\mathbf{x} + \mathbf{b}) + \mathbf{c}$$

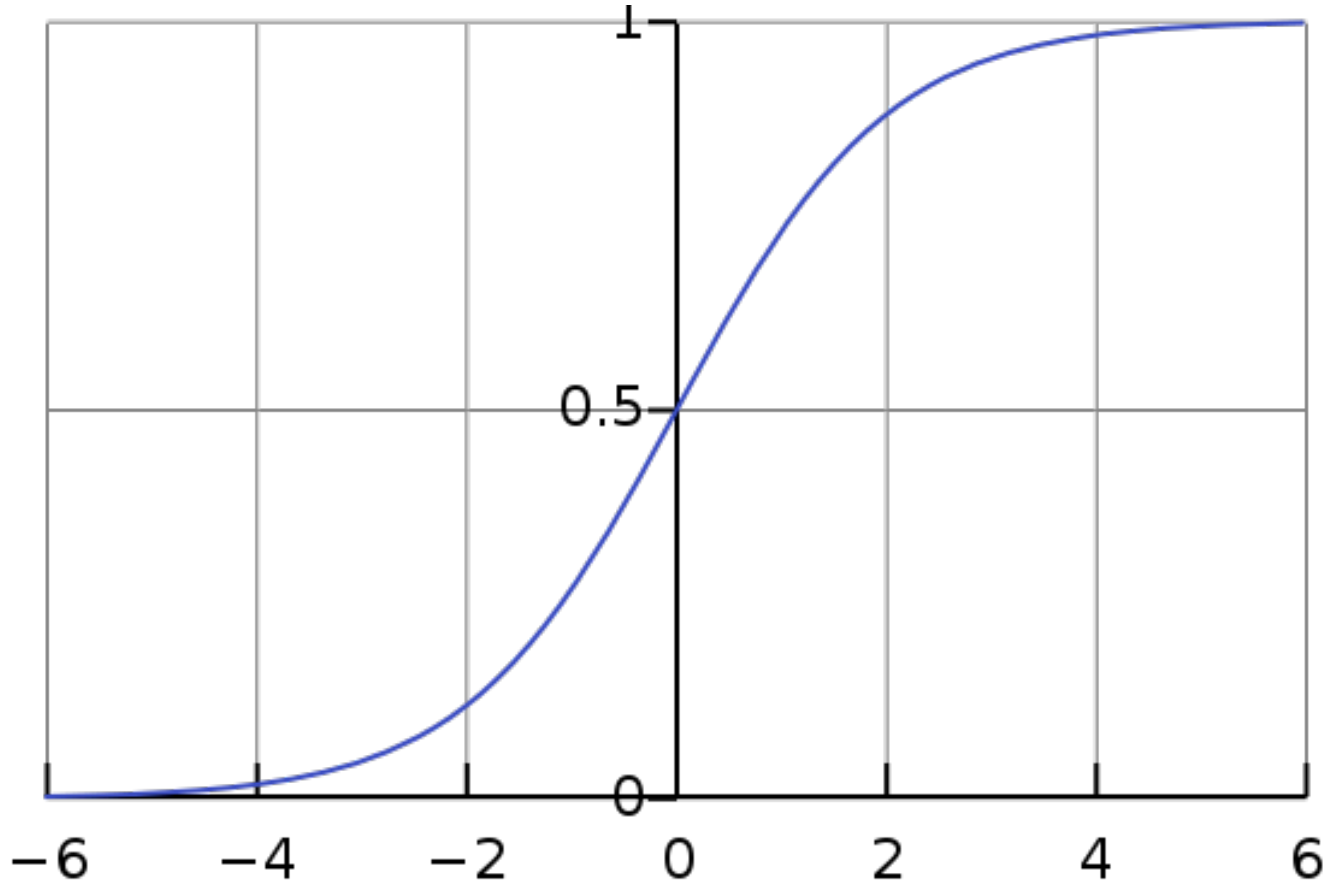
Taken from http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Feedforward Neural Networks, Backpropagation

Recap: Logistic Regression as a NN

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





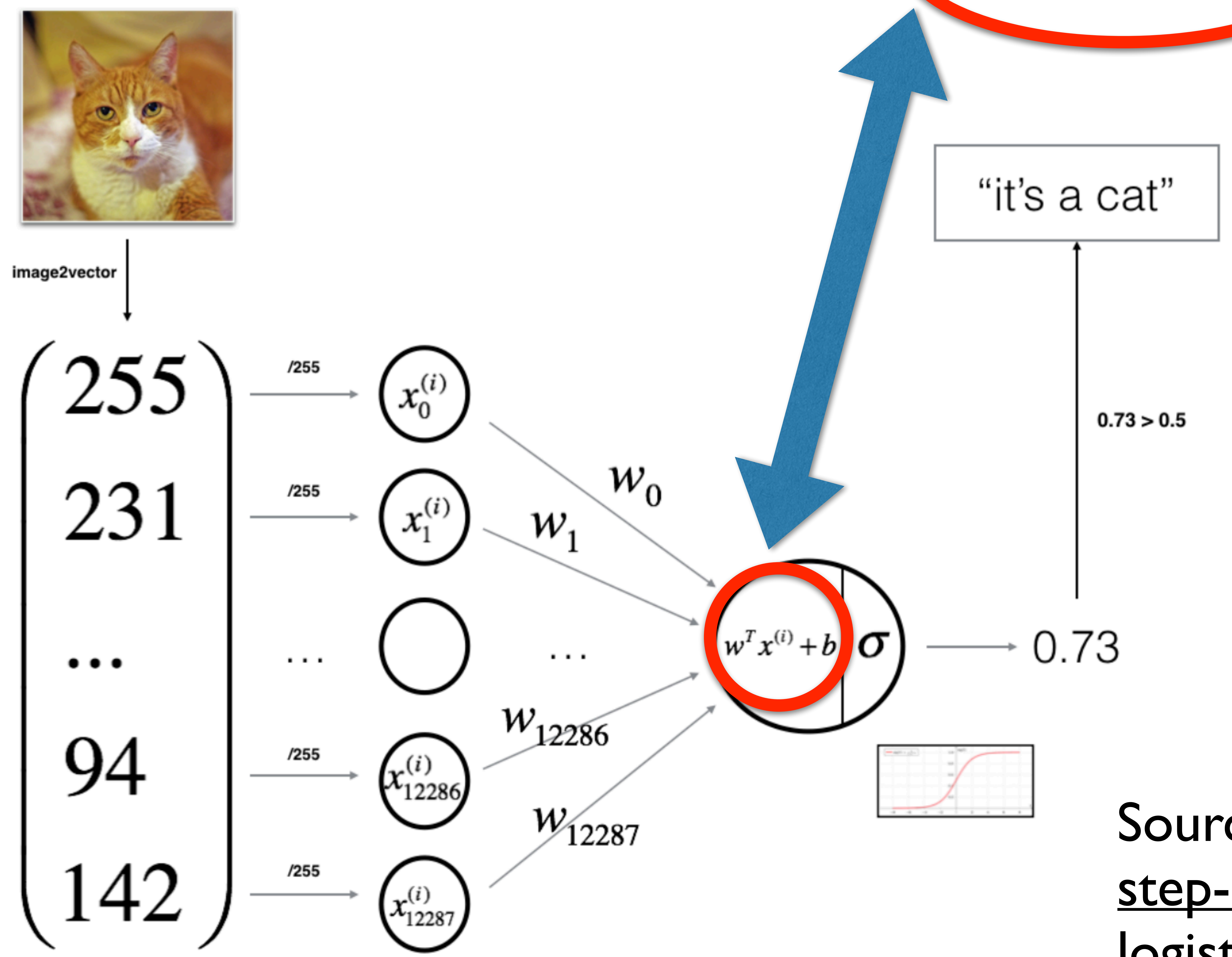
$$\sigma(z) = \frac{1}{1+e^{-z}}$$

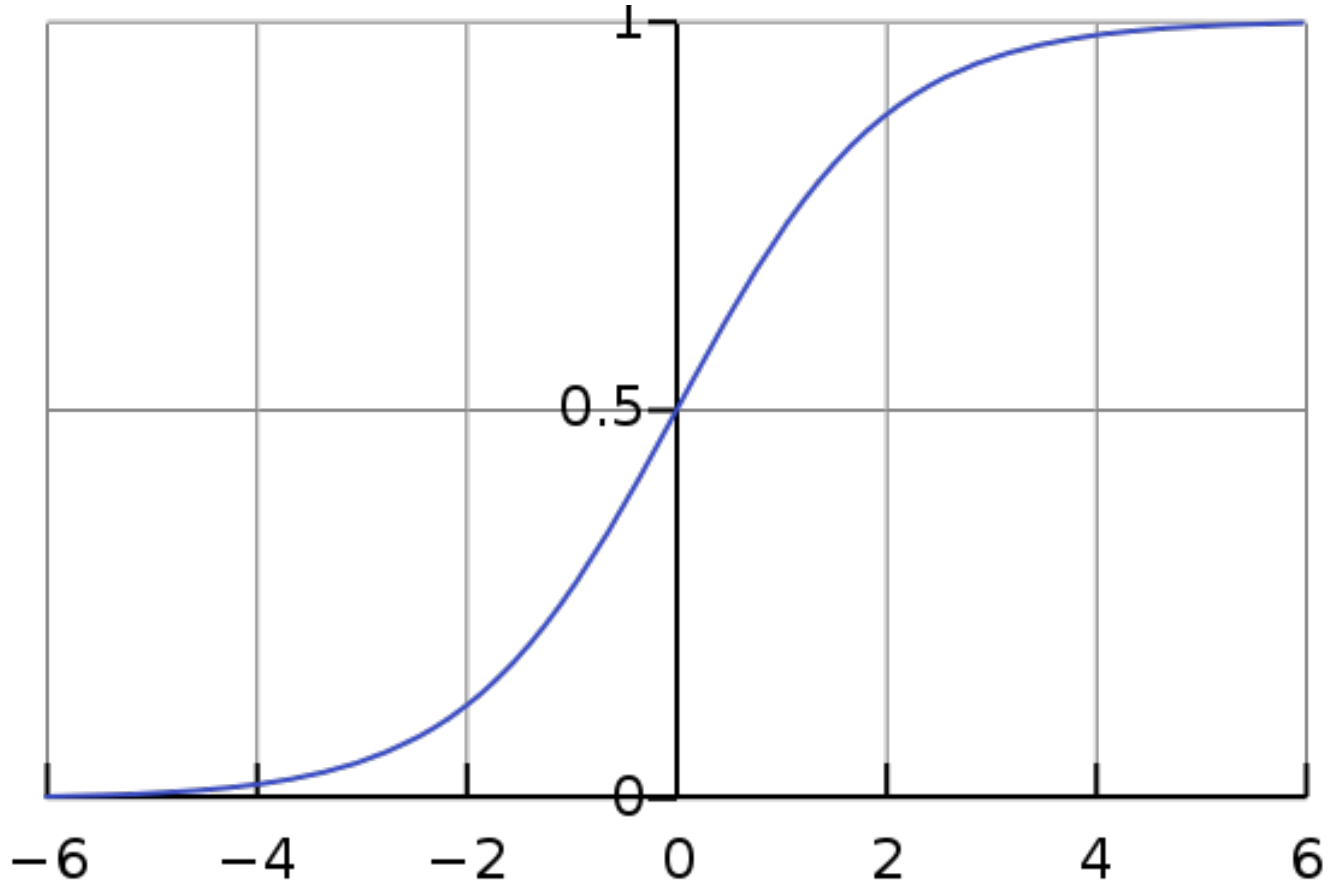
Sigmoid function

e: https://medium.com/@opetundeadepoju/a-
y-step-tutorial-on-coding-neural-network-
c-regression-model-from-scratch-5f9025bd3d6

Recap: Logistic Regression as a NN

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





$$\sigma(z) = \frac{1}{1+e^{-z}}$$ (Sigmoid function)

It considers the bias term b by augmenting features with "1"

Source: https://medium.com/@opetundeadepoju/a-step-by-step-tutorial-on-coding-neural-network-logistic-regression-model-from-scratch-5f9025bd3d6

Recap: Multi-class Logistic Regression

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

- Formulation using single weight vector w; scalar probability for class y

Recap: Multi-class Logistic Regression

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

- Formulation using single weight vector w; scalar probability for class y

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

- We can also compute scores for all possible labels at once (a vector **y** (bold) is returned)

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

- softmax: exps and normalizes a given vector

Recap: Multi-class Logistic Regression

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

- Formulation using single weight vector w; scalar probability for class y

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

- We can also compute scores for all possible labels at once (a vector **y** (bold) is returned)

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

- softmax: exps and normalizes a given vector

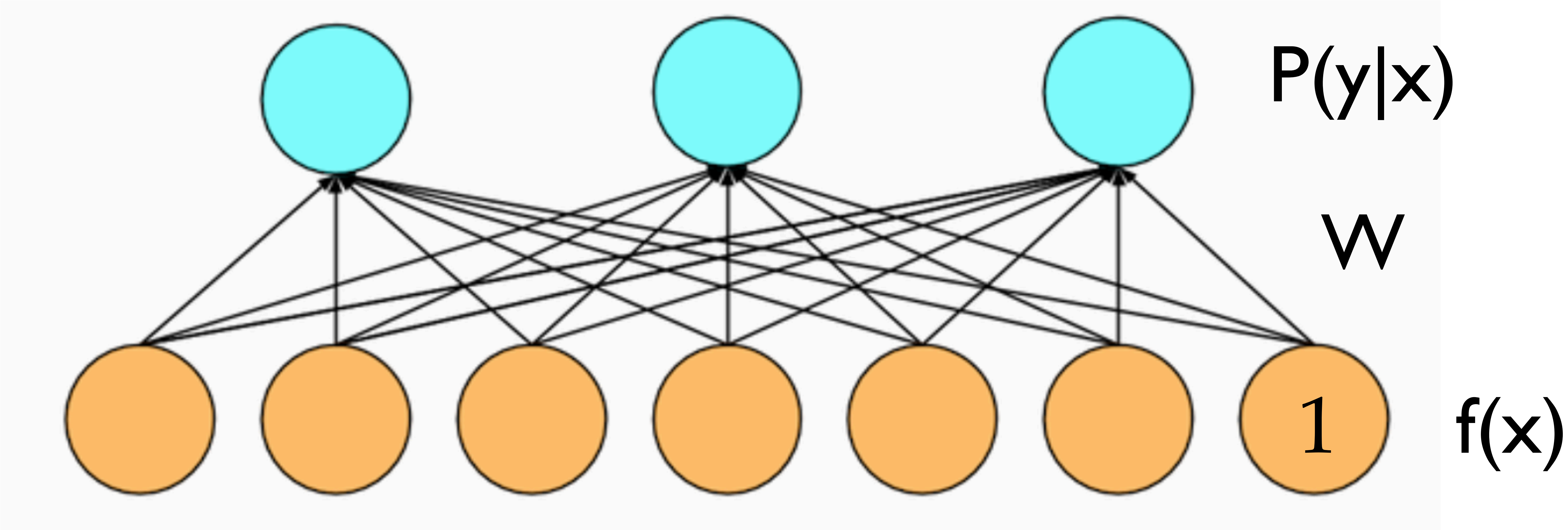
$$P(\mathbf{y}|\mathbf{x}) = \text{softmax}(Wf(\mathbf{x}))$$

- Formulation using single feature vector f(x), but weight vector per class; W is [num classes x num feats]

Recap: Multi-class Logistic Regression as a NN

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

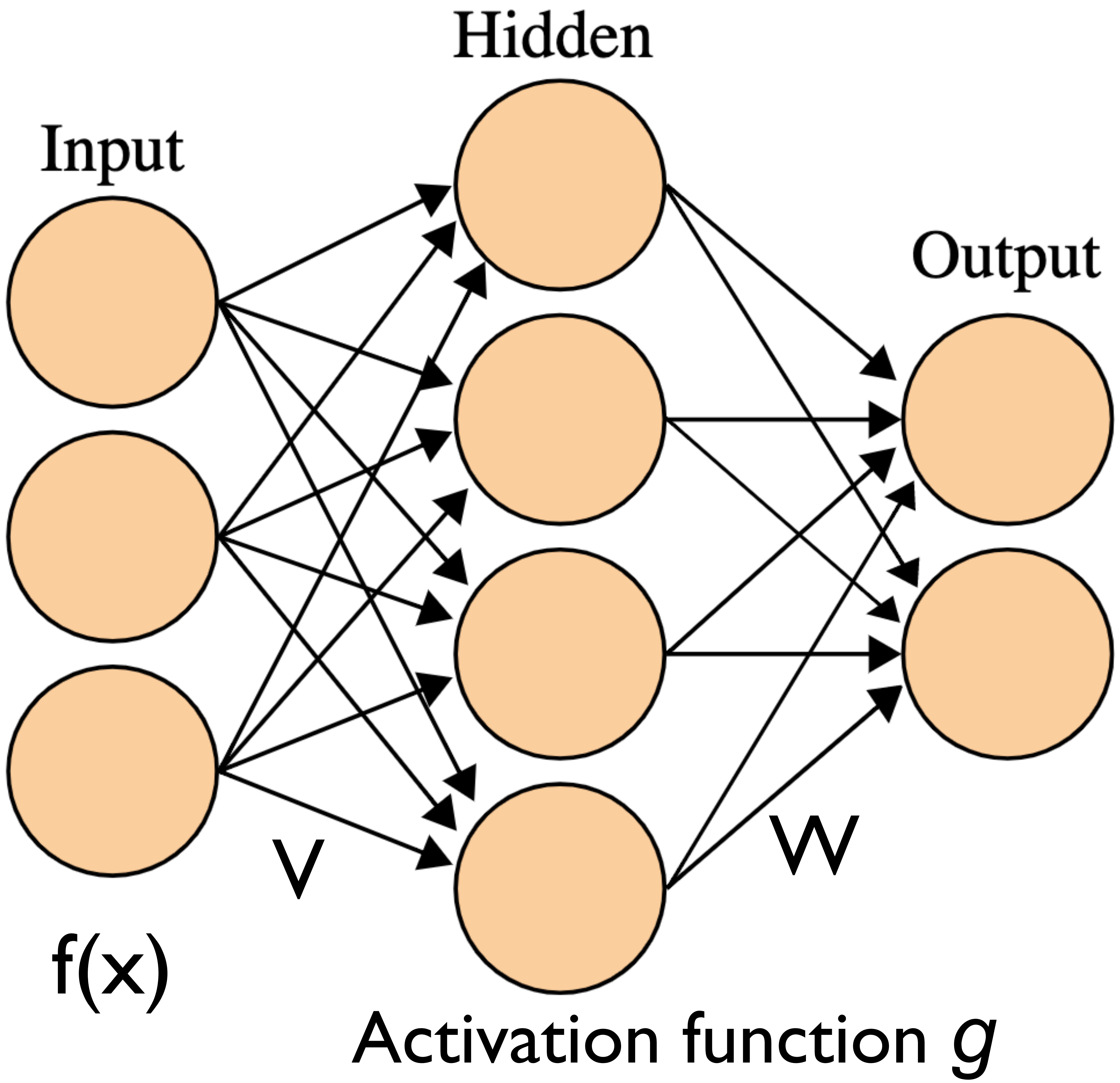
- Formulation using single feature vector f(x), but weight vector per class; W's size is [num classes x num feats]



source: https://gluon.mxnet.io/chapter02_supervised-learning/softmax-regression-scratch.html

Note that when we use W*x or W*f(x), we have included the bias term (as in W*x +b) by augmenting features with "1"

Now, add one hidden layer



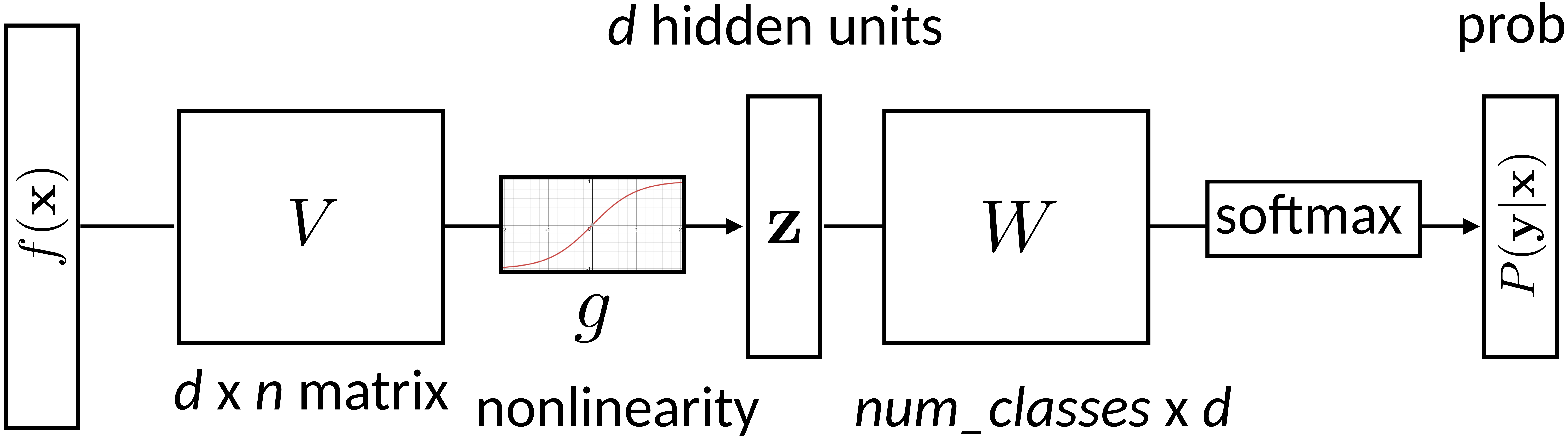
$$P(\mathbf{y}|\mathbf{x}) = \mathrm{softmax}(W f(\mathbf{x}))$$

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

- one hidden layer g(*) added

Neural Networks for Classification

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



n features

d x *n* matrix

nonlinearity
(tanh, relu, …)

d hidden units

num_classes x *d*
matrix

num_classes
probs

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

Training Neural Networks

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

- Maximize log likelihood of training data

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

- *i**: index of the gold label

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

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

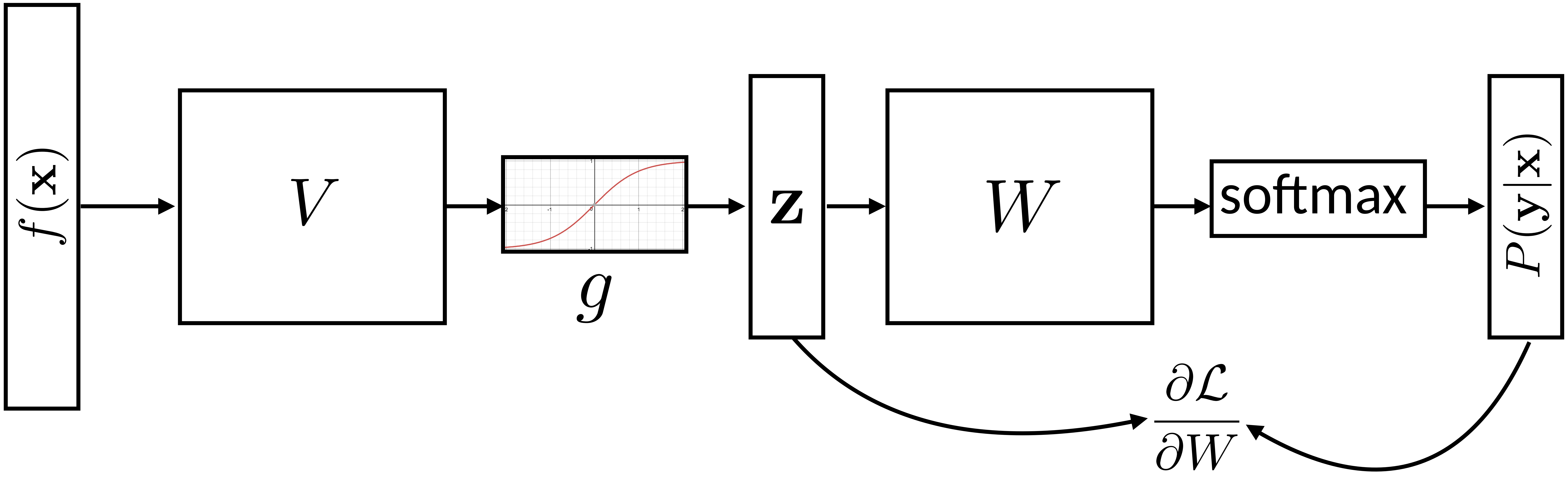
- Gradient with respect to *W will be a matrix of the same size*

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

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

Neural Networks for Classification

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


$f(\mathbf{x})$
V
g
$\mathbf{z}$
W
softmax
$P(\mathbf{y}|\mathbf{x})$
$\frac{\partial \mathcal{L}}{\partial W}$

- How to compute gradients with respect to *V*?

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

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

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 = m

(error signal at the final layer)

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 = m

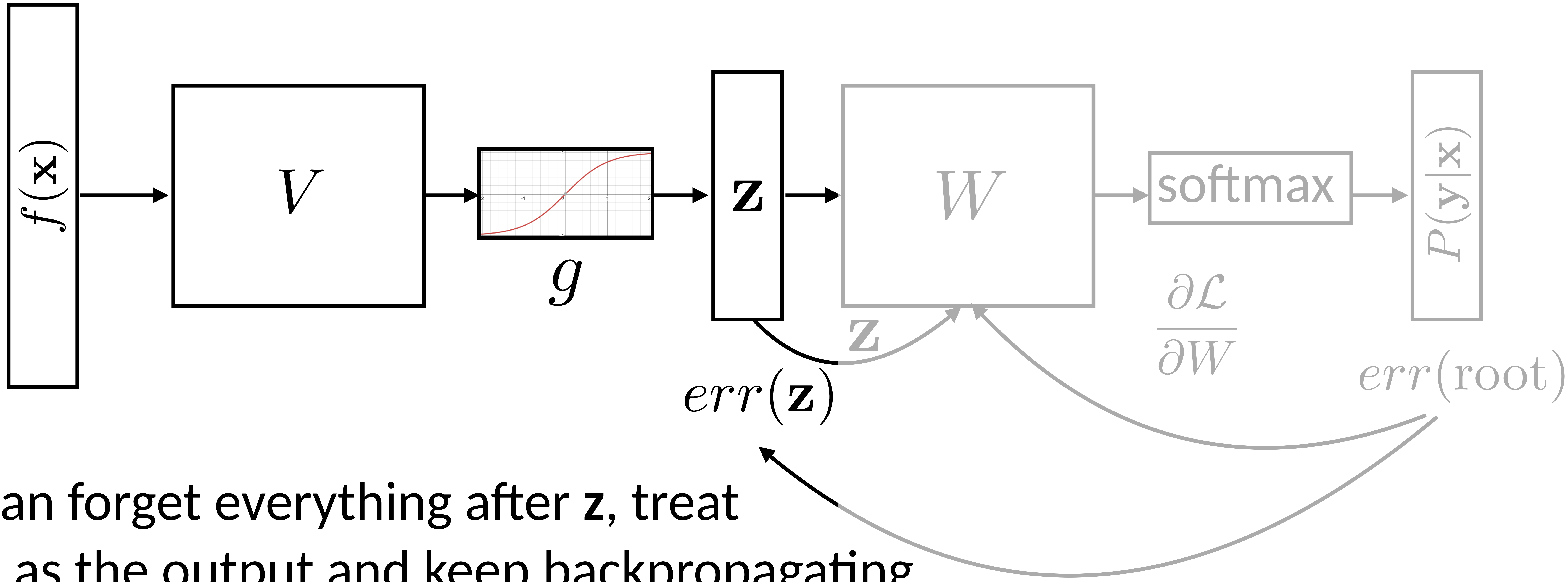
$$\boxed{\frac{\partial \mathcal{L}(\mathbf{x}, i^*)}{\partial \mathbf{z}} = err(\mathbf{z}) = W^\top err(\text{root})}$$

dim = d

Offline practice

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 backpropagating

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

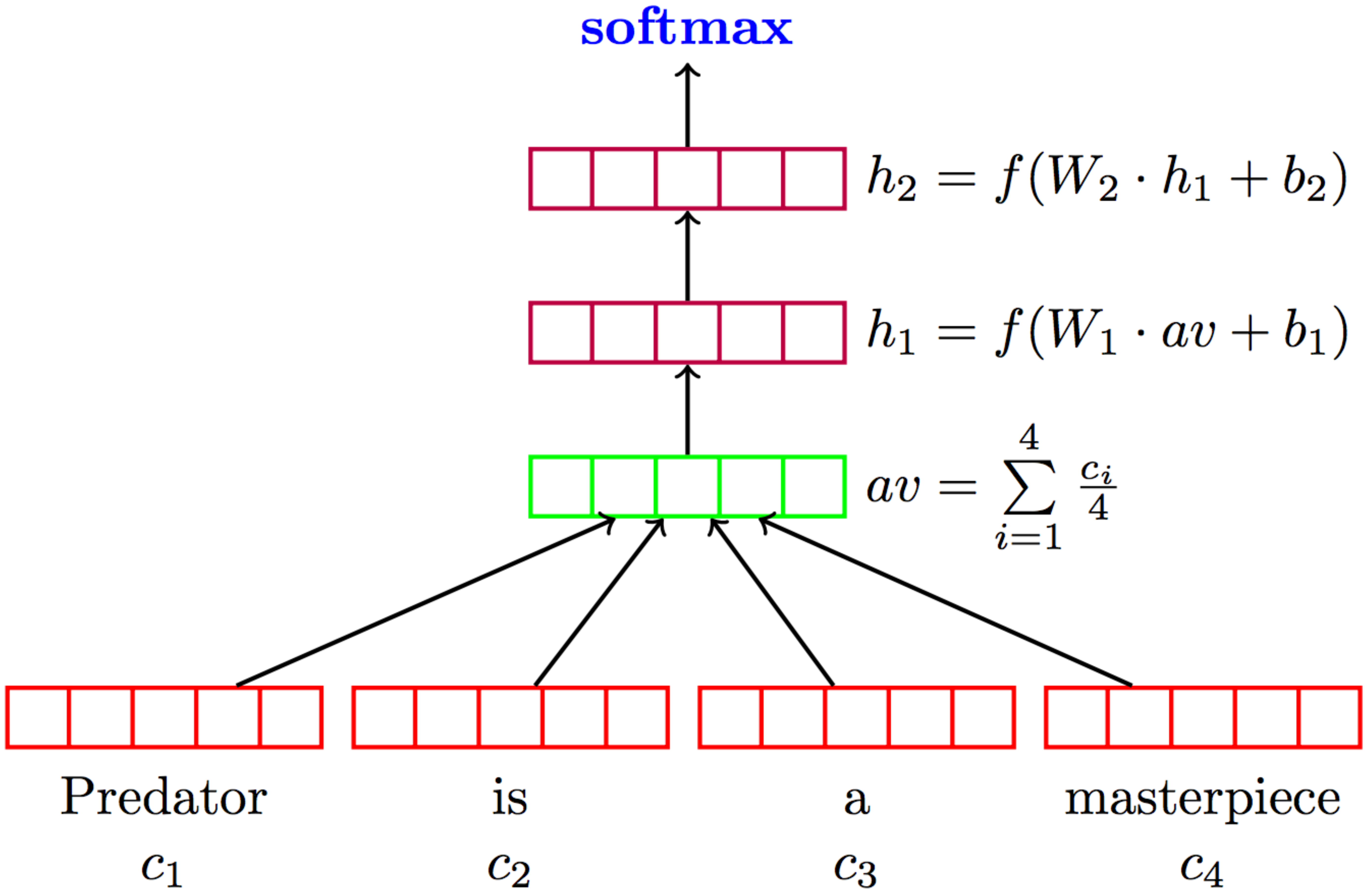
- Easy to update parameters based on “error signal” from next layer, keep pushing error signal back as backpropagation

- Need to store the values from the forward computation

Applications

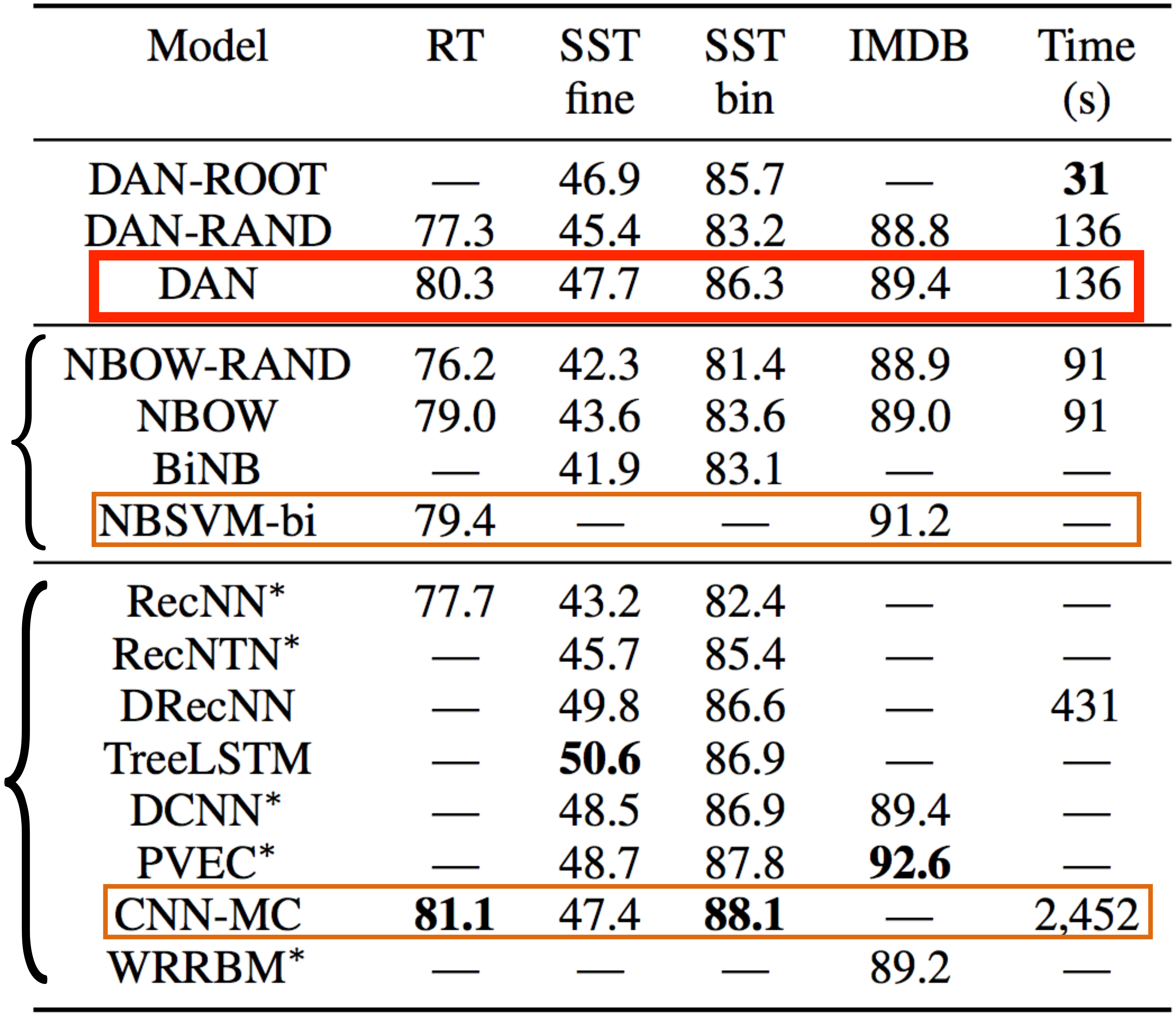
Sentiment Analysis

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


softmax
$h_2 = f(W_2 \cdot h_1 + b_2)$
$h_1 = f(W_1 \cdot av + b_1)$
$av = \sum_{i=1}^{4} \frac{c_i}{4}$
Predator
c_1
is
c_2
a
c_3
masterpiece
c_4

Iyyer et al. (2015)

Sentiment Analysis



Model	RT	SST	SST	IMDB	Time
		fine	bin		(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)

Word Representations

- Neural networks work very well at continuous data, but words are discrete
- Continuous model <-> expects continuous semantic representation from input

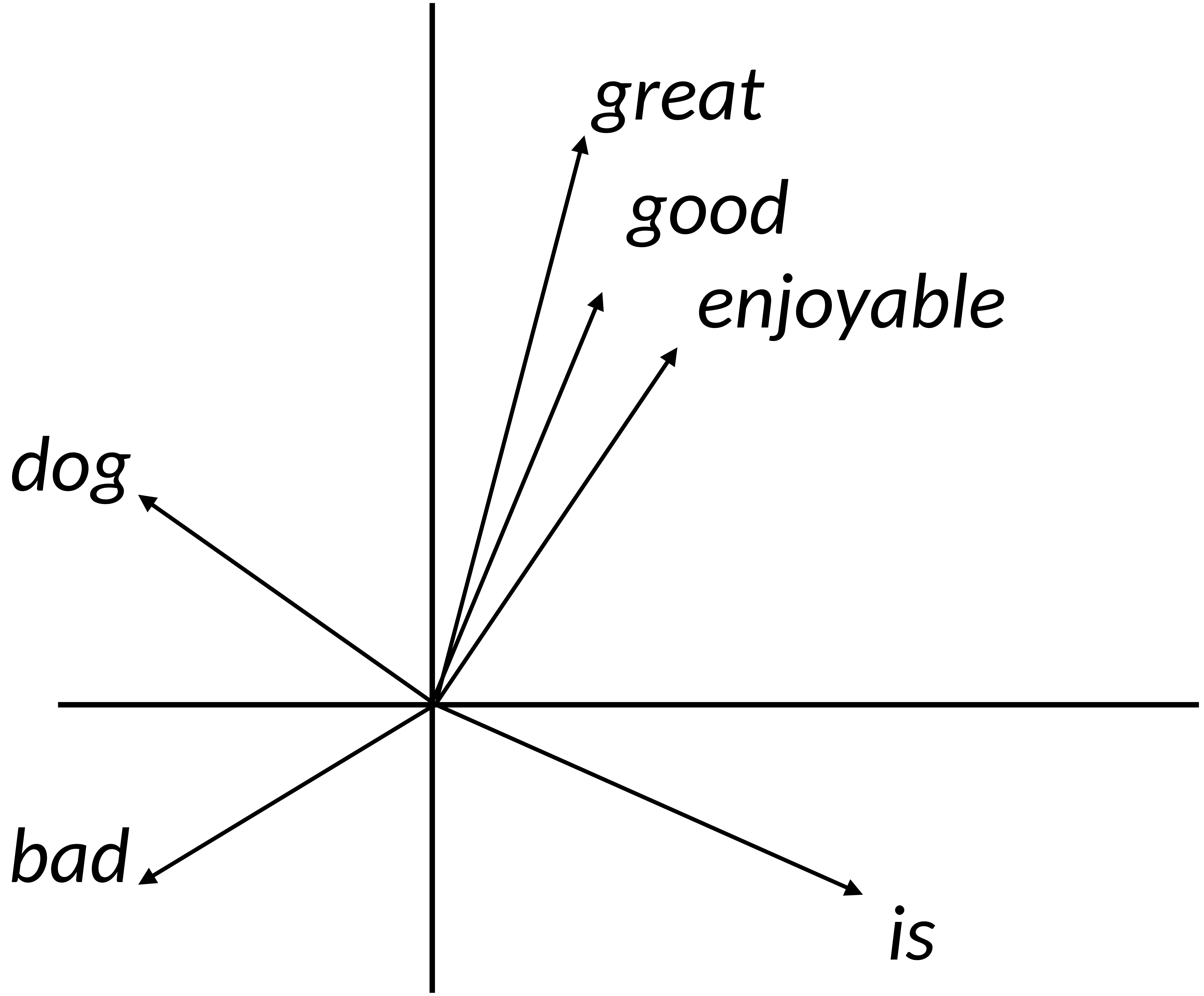
Word Embeddings

- Want a vector space where similar words have similar embeddings

the movie was great

≈

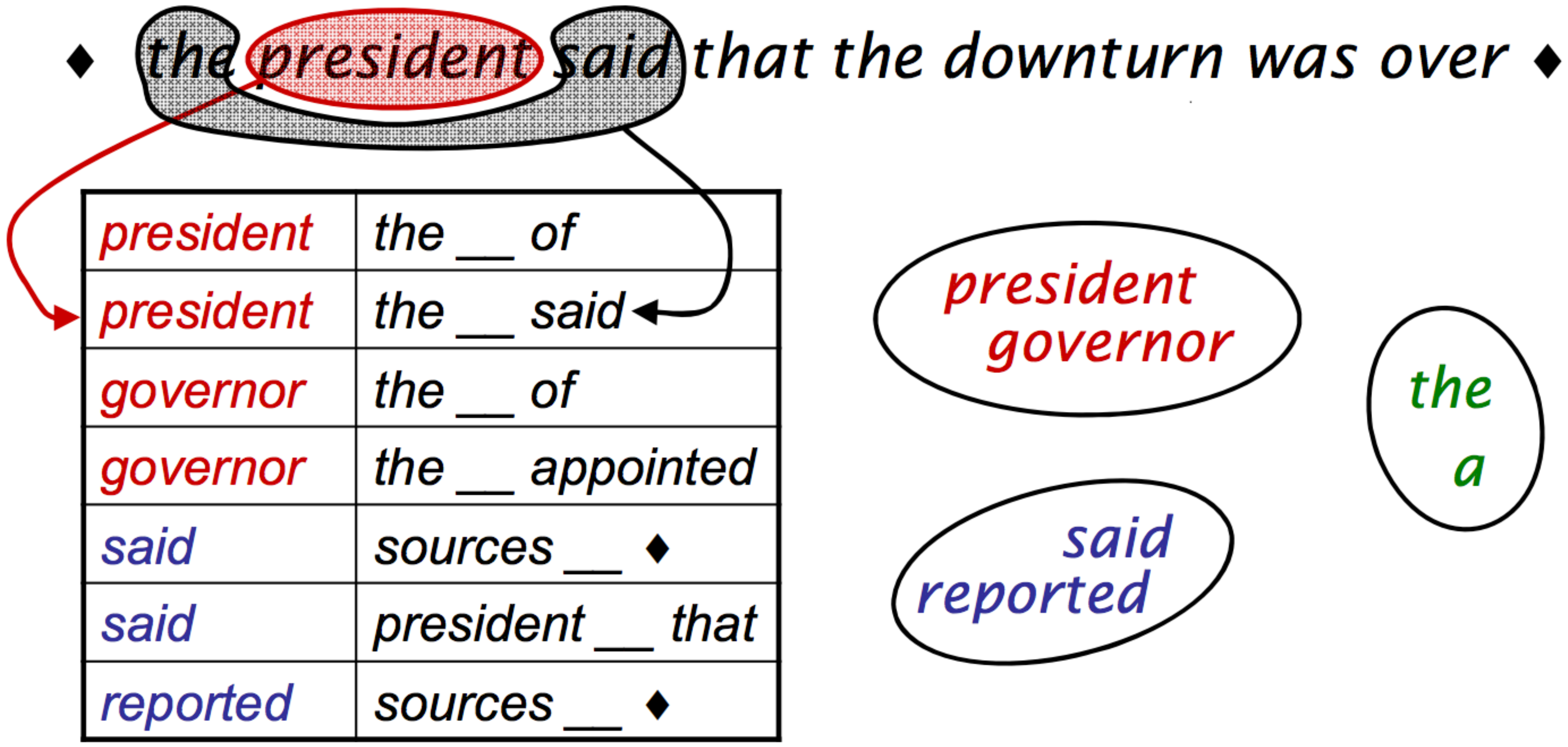
the movie was good



- Goal: come up with a way to produce these embeddings
- For each word, want "medium" dimensional vector (50-300 dims) representing it

Word Representations

- Neural networks work very well at continuous data, but words are discrete
- Continuous model <-> expects continuous semantic representation from input
- “You shall know a word by the company it keeps” Firth (1957)



♦ the president said that the downturn was over ♦

president	the __ of
president	the __ said
governor	the __ of
governor	the __ appointed
said	sources __ ♦
said	president __ that
reported	sources __ ♦

[Finch and Chater 92, Shuetze 93, many others]

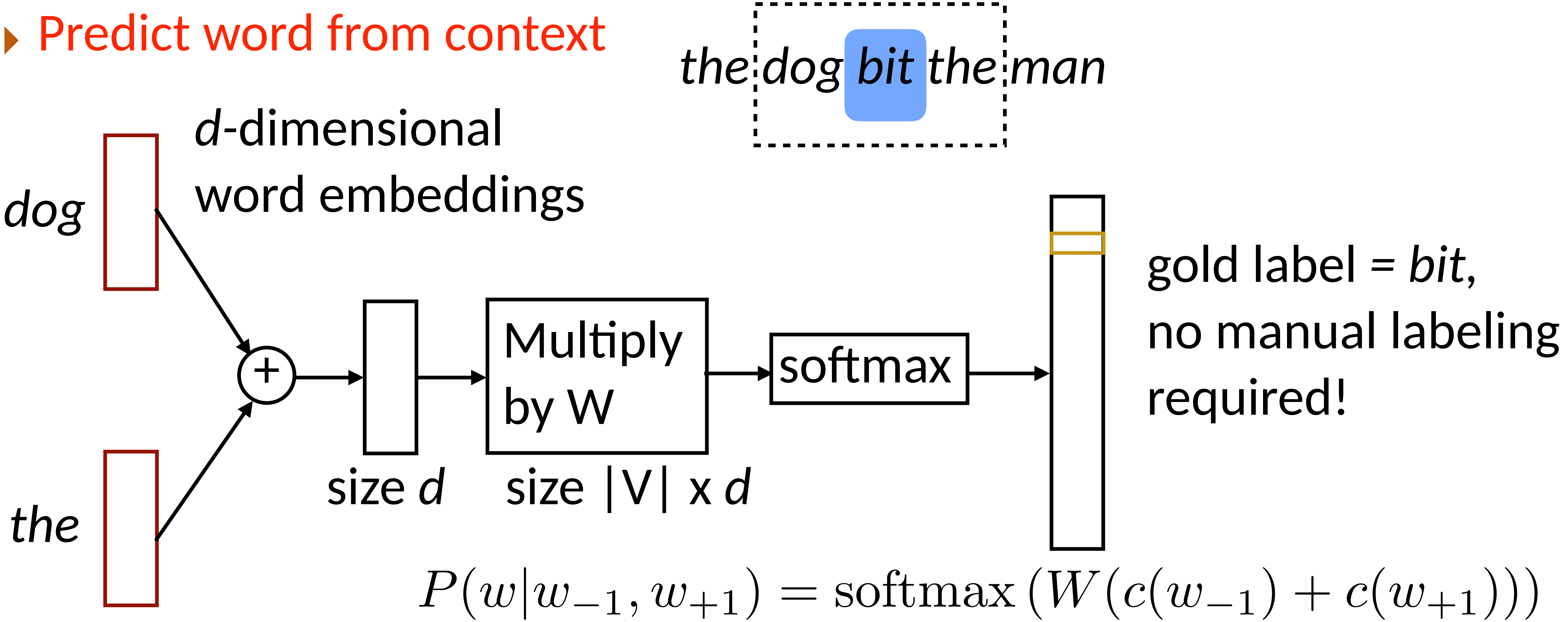
slide credit: Dan Klein

word2vec/GloVe

To know more:
(1) Word2vec: https://arxiv.org/pdf/1301.3781.pdf
(2) GloVe: https://nlp.stanford.edu/pubs/glove.pdf

Continuous Bag-of-Words (CBOW)

- Predict word from context



the dog bit the man
d-dimensional word embeddings
dog
the
+
size *d*
Multiply by W
size |V| x *d*
softmax
gold label = *bit*, no manual labeling required!

$$P(w|w_{-1}, w_{+1}) = \text{softmax}\left(W(c(w_{-1}) + c(w_{+1}))\right)$$

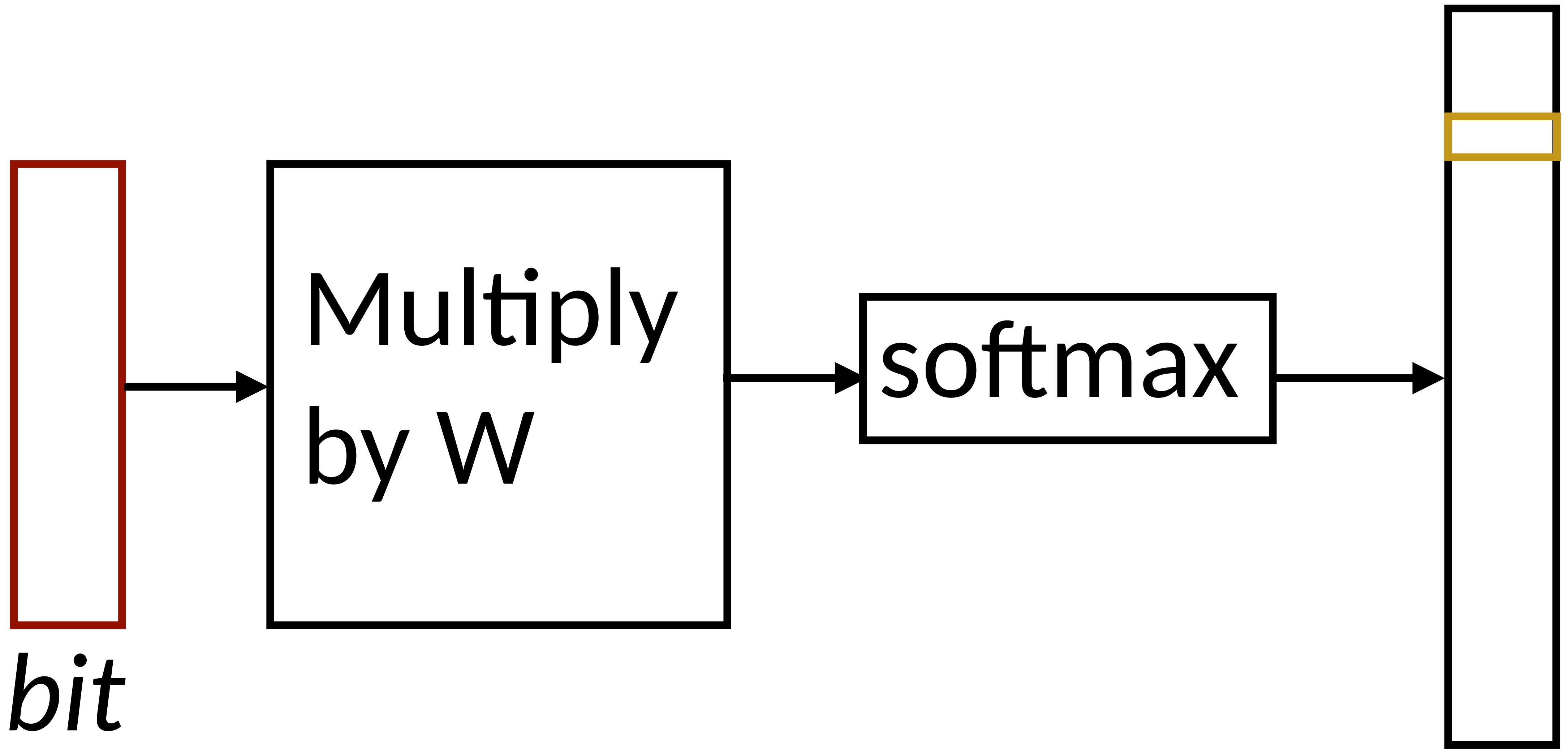
- Parameters: *d* x |V| (one *d*-length context vector per word in the vocab.), |V| x *d* output parameters (W)

Mikolov et al. (2013)

Skip-Gram

- Predict one word of context from word

the dog bit the man



Multiply by W

softmax

bit

gold = *dog*

$$P(w'|w) = \mathrm{softmax}(We(w))$$

- Another training example: *bit* -> *the*
- Parameters: *d* x |V| vectors, |V| x *d* output parameters (W) (also usable as vectors!)

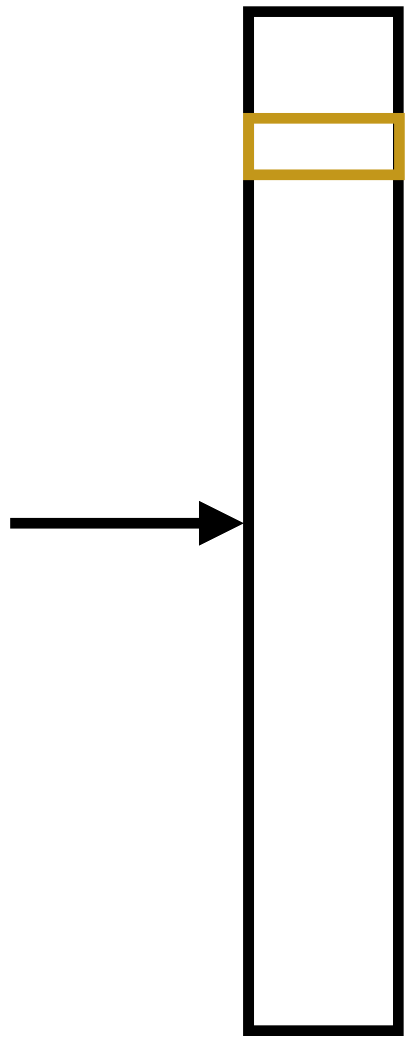
Mikolov et al. (2013)

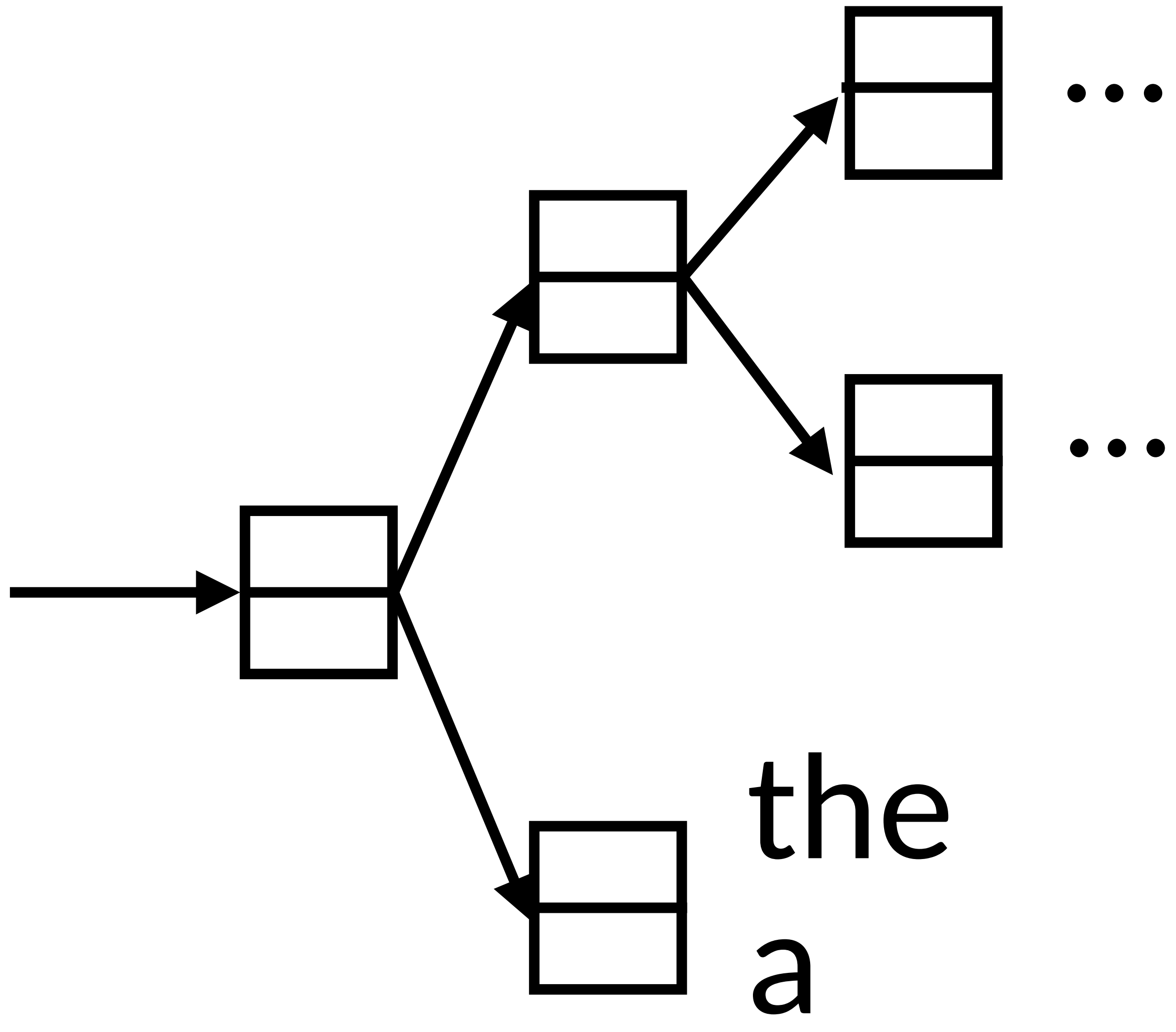
Hierarchical Softmax

Offline reading, if interested

$P(w|w_{-1}, w_{+1}) = \mathrm{softmax}\left(W(c(w_{-1}) + c(w_{+1}))\right)$ $P(w'|w) = \mathrm{softmax}(We(w))$

- Matrix multiplication + softmax over |V| is very slow to compute for CBOW and SG





- Huffman encode vocabulary, use binary classifiers to decide which branch to take
- log(|V|) binary decisions

- Standard softmax: [|V| x *d*] x *d*

- Hierarchical softmax: log(|V|) dot products of size *d*, |V| x *d* parameters

Mikolov et al. (2013)

Skip-Gram with Negative Sampling

Offline reading, if interested

- Take (word, context) pairs and classify them as “real” or not. Create random negative examples by sampling from uniform distribution

(*bit*, *the*) => +1

(*bit*, *cat*) => -1

(*bit*, *a*) => -1

(*bit*, *fish*) => -1

$$P(y=1|w,c) = \frac{e^{w\cdot c}}{e^{w\cdot c}+1}$$

words in similar contexts select for similar c vectors

- *d* x |V| vectors, *d* x |V| context vectors (same # of params as before)

sampled

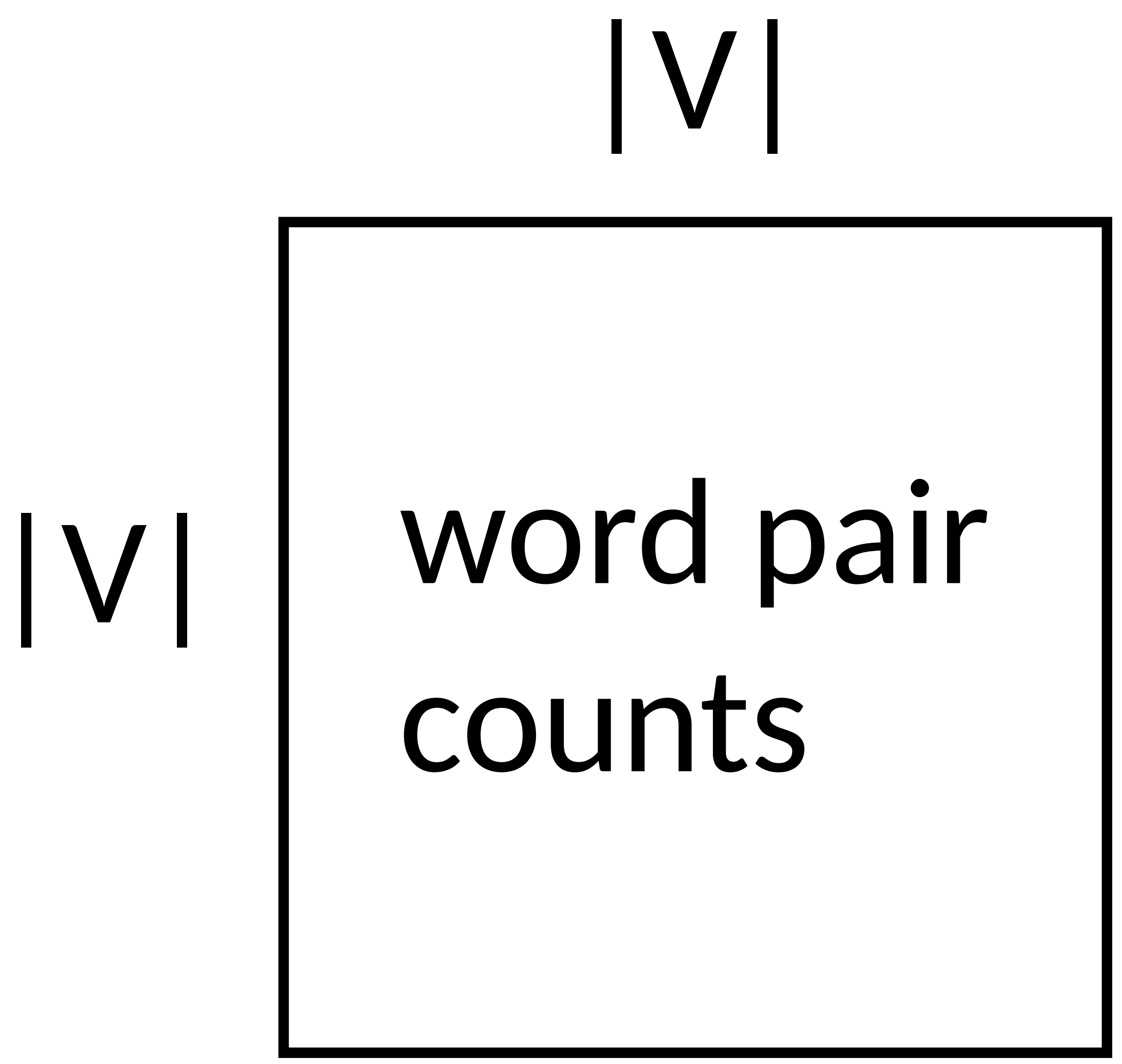
- Objective = $\log P(y=1|w,c) + \frac{1}{k}\sum_{i=1}^{n} \log P(y=0|w_i,c)$

https://arxiv.org/pdf/1301.3781.pdf

Mikolov et al. (2013)

GloVe (**Glo**bal **Ve**ctors)

- Operates on counts matrix; weighted regression on the log co-occurrence matrix



- Objective = $\sum_{i,j} f(\mathrm{count}(w_i, c_j)) \left(w_i^\top c_j + a_i + b_j - \log \mathrm{count}(w_i, c_j)\right)^2$
- Constant in the dataset size (just need counts), quadratic in vocab. size

https://nlp.stanford.edu/pubs/glove.pdf

Pennington et al. (2014)

Using Word Embeddings

- Approach 1: learn embeddings as parameters from your data
  - Often works pretty well
- Approach 2: initialize using GloVe, keep fixed
  - Faster because no need to update these parameters
- Approach 3: initialize using GloVe, fine-tune
  - Works best for some tasks

download GloVe here: https://nlp.stanford.edu/projects/glove/
word2vec: https://github.com/tmikolov/word2vec

Evaluation

Evaluating Word Embeddings

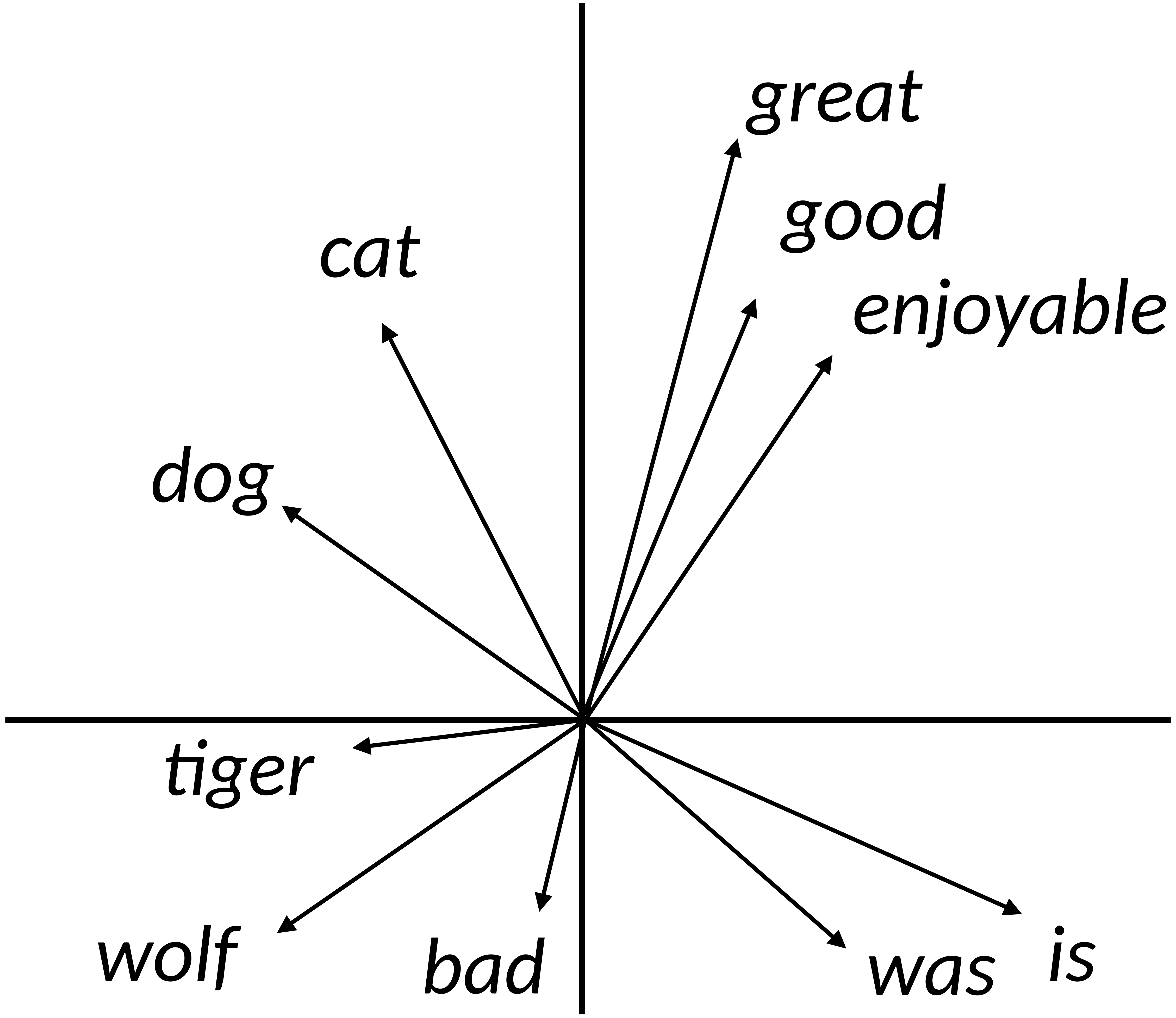
- What properties of language should word embeddings capture?

- Similarity: similar words are close to each other

- Analogy:

  good is to best as smart is to ???

  Paris is to France as Tokyo is to ???

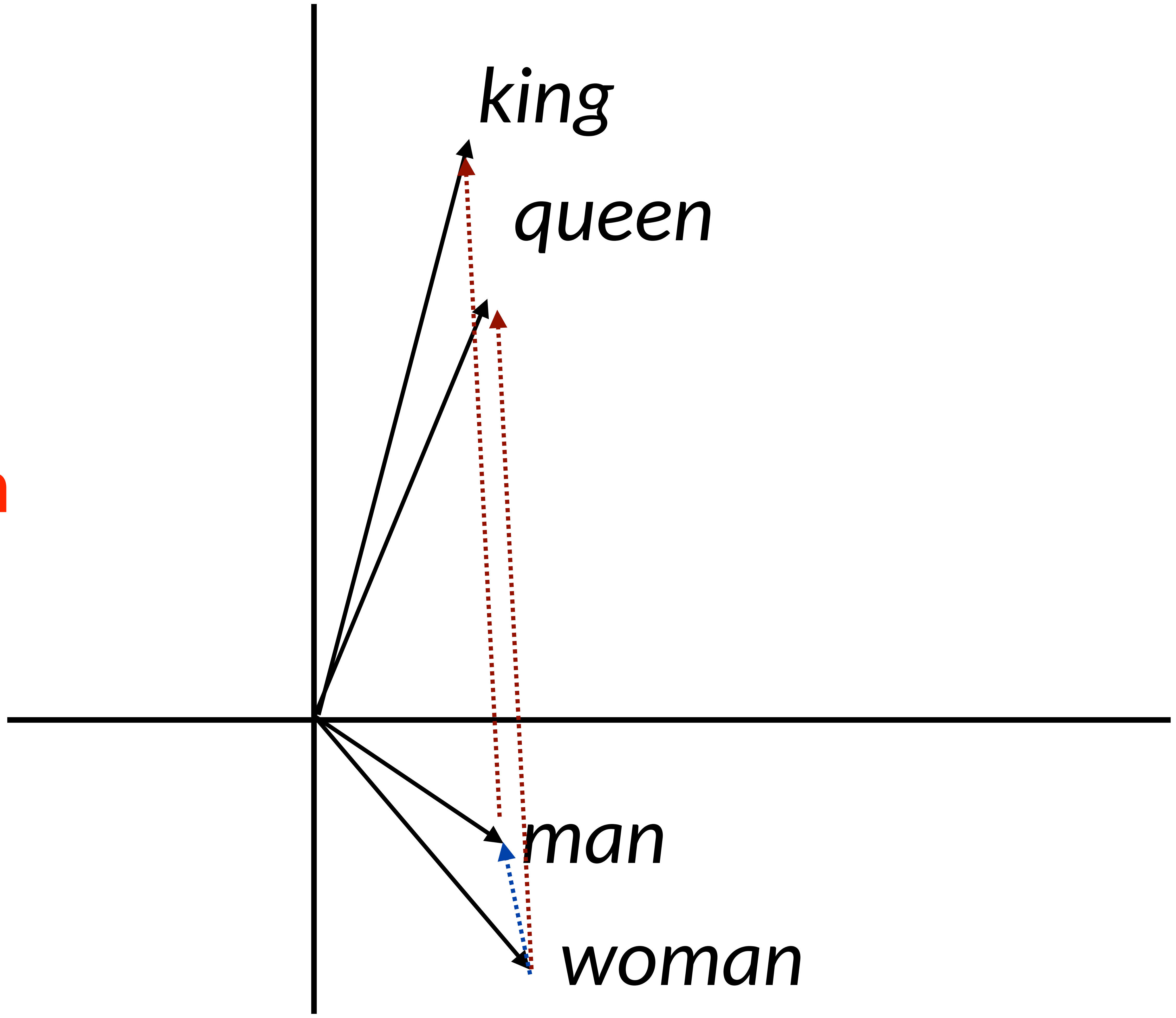


Analogies

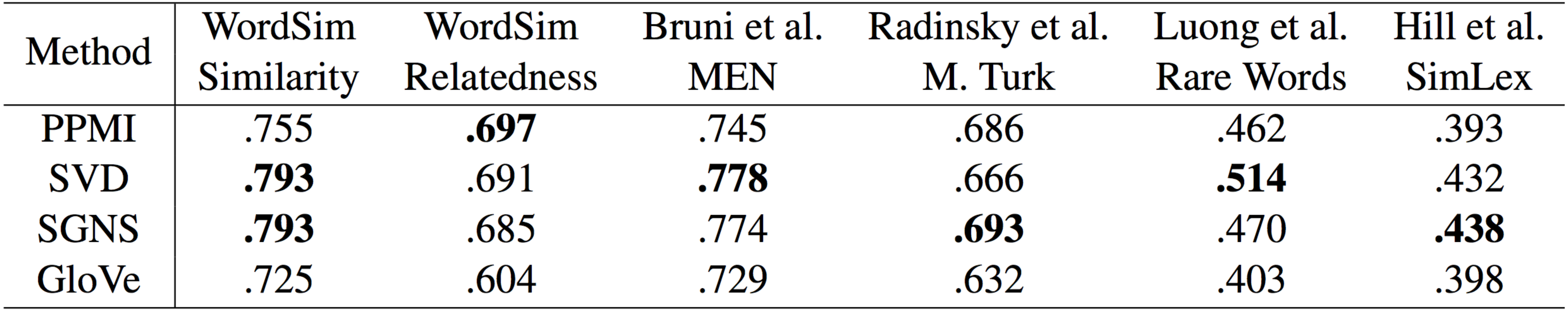
(king - man) + woman = queen

king + (woman - man) = queen

- Why would this be?
- woman - man captures the difference in the contexts that these occur in
- Dominant change: more “he” with man and “she” with woman — similar to difference between king and queen
- Can evaluate on this as well



Similarity



Method	WordSim Similarity	WordSim Relatedness	Bruni et al. MEN	Radinsky et al. M. Turk	Luong et al. Rare Words	Hill et al. SimLex
PPMI	.755	**.697**	.745	.686	.462	.393
SVD	**.793**	.691	**.778**	.666	**.514**	.432
SGNS	**.793**	.685	.774	**.693**	.470	**.438**
GloVe	.725	.604	.729	.632	.403	.398

- SVD = singular value decomposition on PMI matrix
- GloVe does not appear to be the best when experiments are carefully controlled, but it depends on hyperparameters + these distinctions don't matter in practice

Levy et al. (2015)

Use Word Vectors for NLP Tasks

- Named Entity Recognition

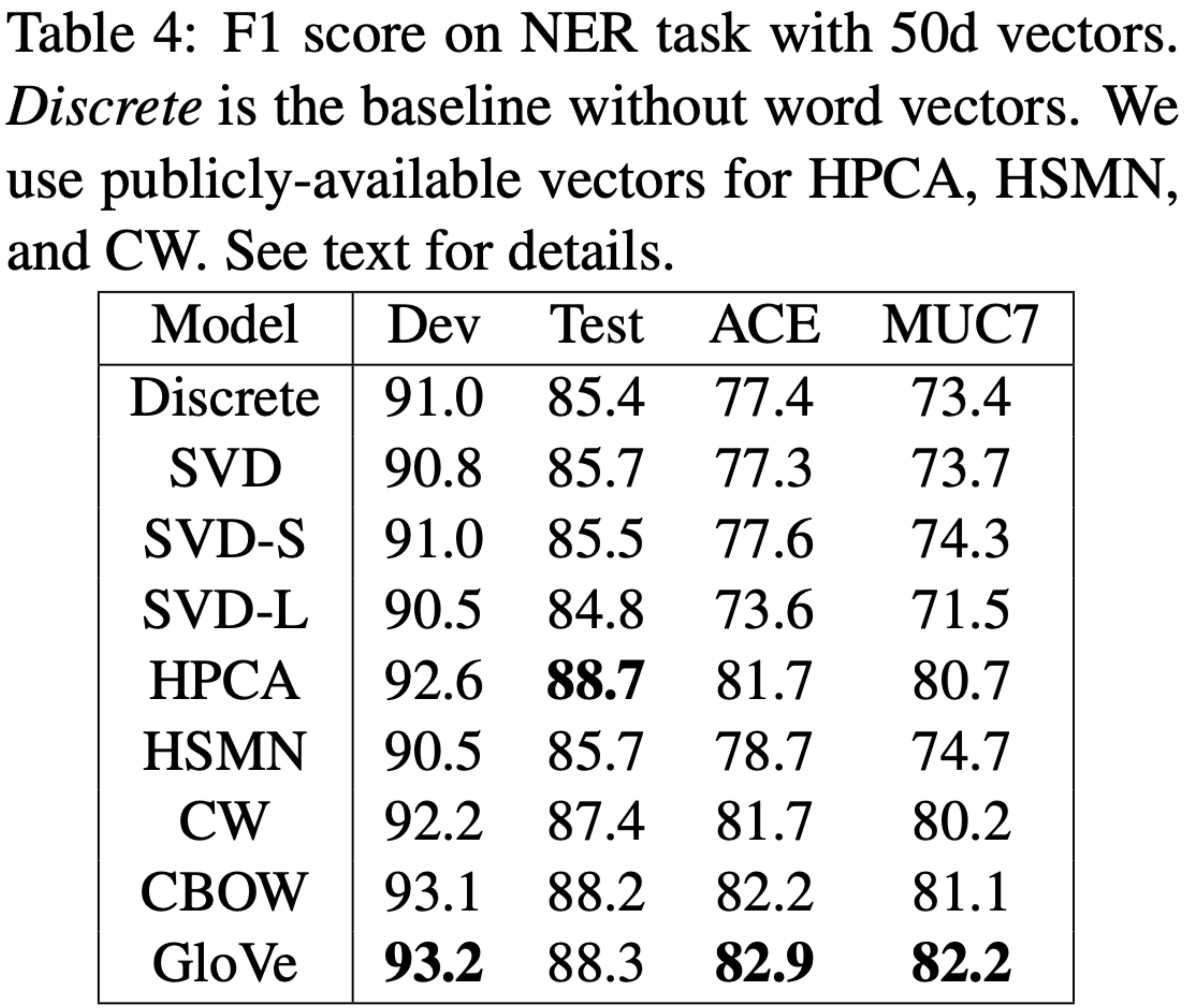


Table 4: F1 score on NER task with 50d vectors. *Discrete* is the baseline without word vectors. We use publicly-available vectors for HPCA, HSMN, and CW. See text for details.

Model	Dev	Test	ACE	MUC7
Discrete	91.0	85.4	77.4	73.4
SVD	90.8	85.7	77.3	73.7
SVD-S	91.0	85.5	77.6	74.3
SVD-L	90.5	84.8	73.6	71.5
HPCA	92.6	**88.7**	81.7	80.7
HSMN	90.5	85.7	78.7	74.7
CW	92.2	87.4	81.7	80.2
CBOW	93.1	88.2	82.2	81.1
GloVe	**93.2**	88.3	**82.9**	**82.2**

See more details in GloVe: https://nlp.stanford.edu/pubs/glove.pdf

What can go wrong with word embeddings?

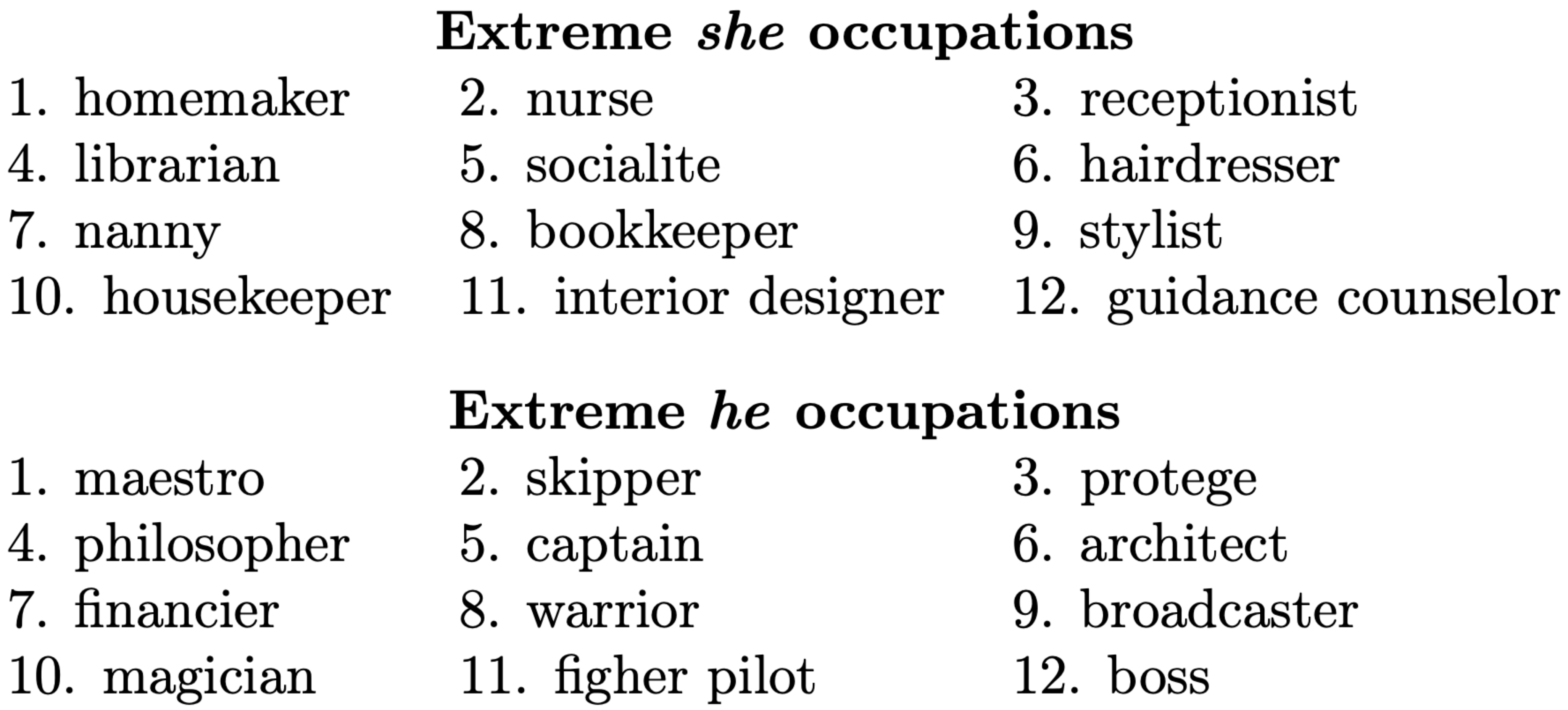
- What’s wrong with learning a word’s “meaning” from its usage?

- What data are we learning from?

- What are we going to learn from this data?

What do we mean by bias?

- Identify *she* - *he* axis in word vector space, project words onto this axis



Extreme *she* occupations

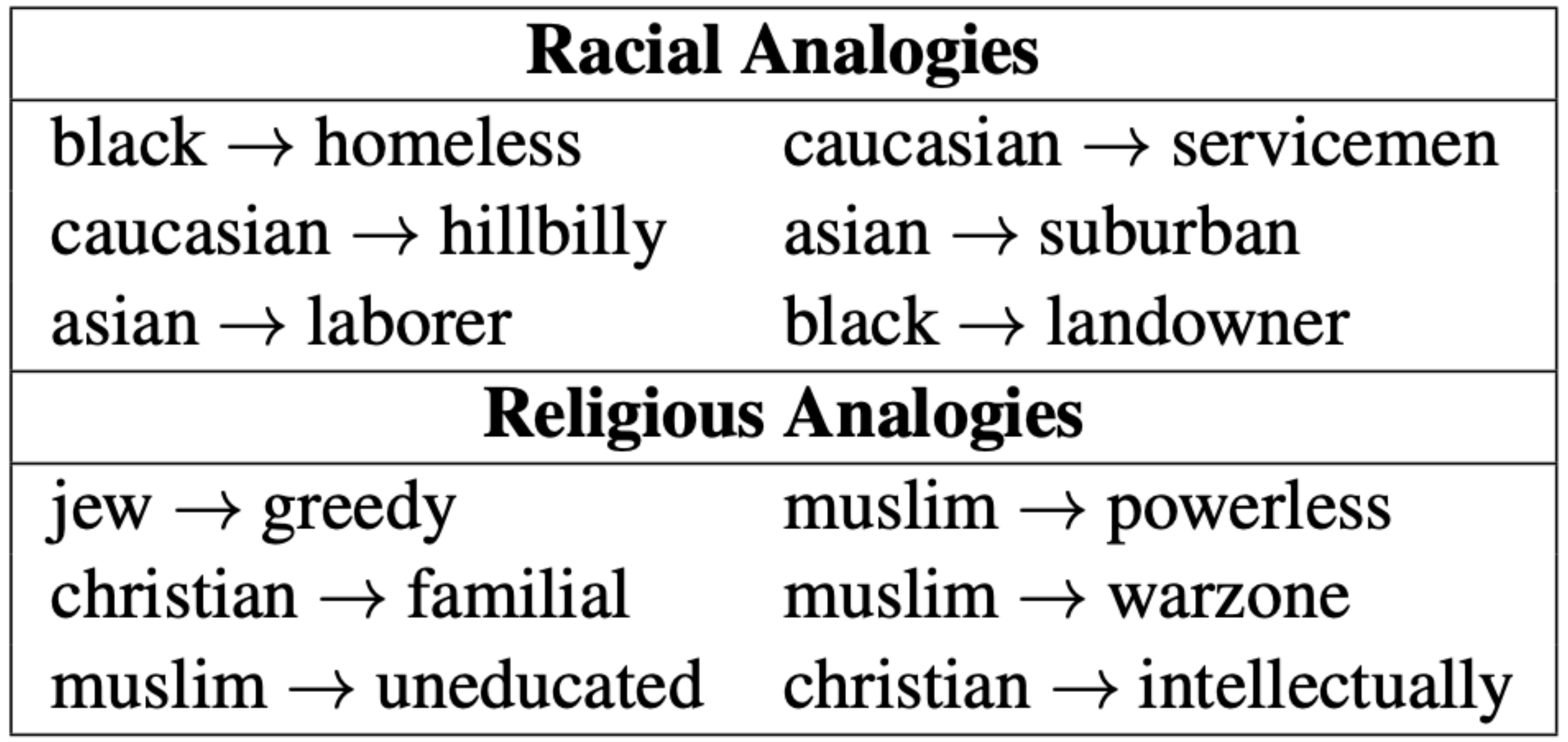
1. homemaker	2. nurse	3. receptionist
4. librarian	5. socialite	6. hairdresser
7. nanny	8. bookkeeper	9. stylist
10. housekeeper	11. interior designer	12. guidance counselor

Extreme *he* occupations

1. maestro	2. skipper	3. protege
4. philosopher	5. captain	6. architect
7. financier	8. warrior	9. broadcaster
10. magician	11. figher pilot	12. boss

Bolukbasi et al. (2016)

- Nearest neighbor of (b - a + c)

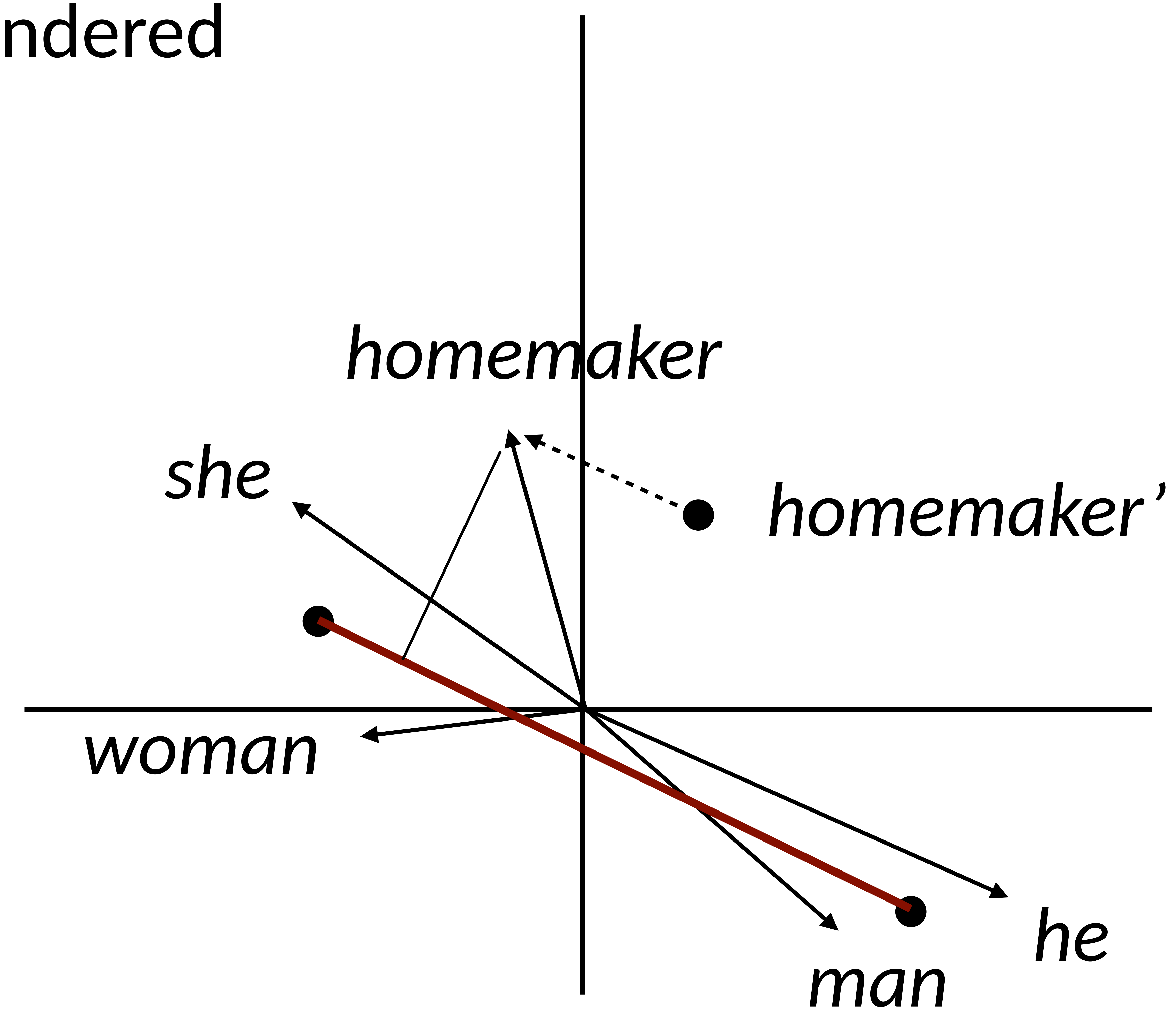


Racial Analogies	
black → homeless	caucasian → servicemen
caucasian → hillbilly	asian → suburban
asian → laborer	black → landowner
Religious Analogies	
jew → greedy	muslim → powerless
christian → familial	muslim → warzone
muslim → uneducated	christian → intellectually

Manzini et al. (2019)

Debiasing

- Identify gender subspace with gendered words
- Project words onto this subspace
- Subtract those projections from the original word

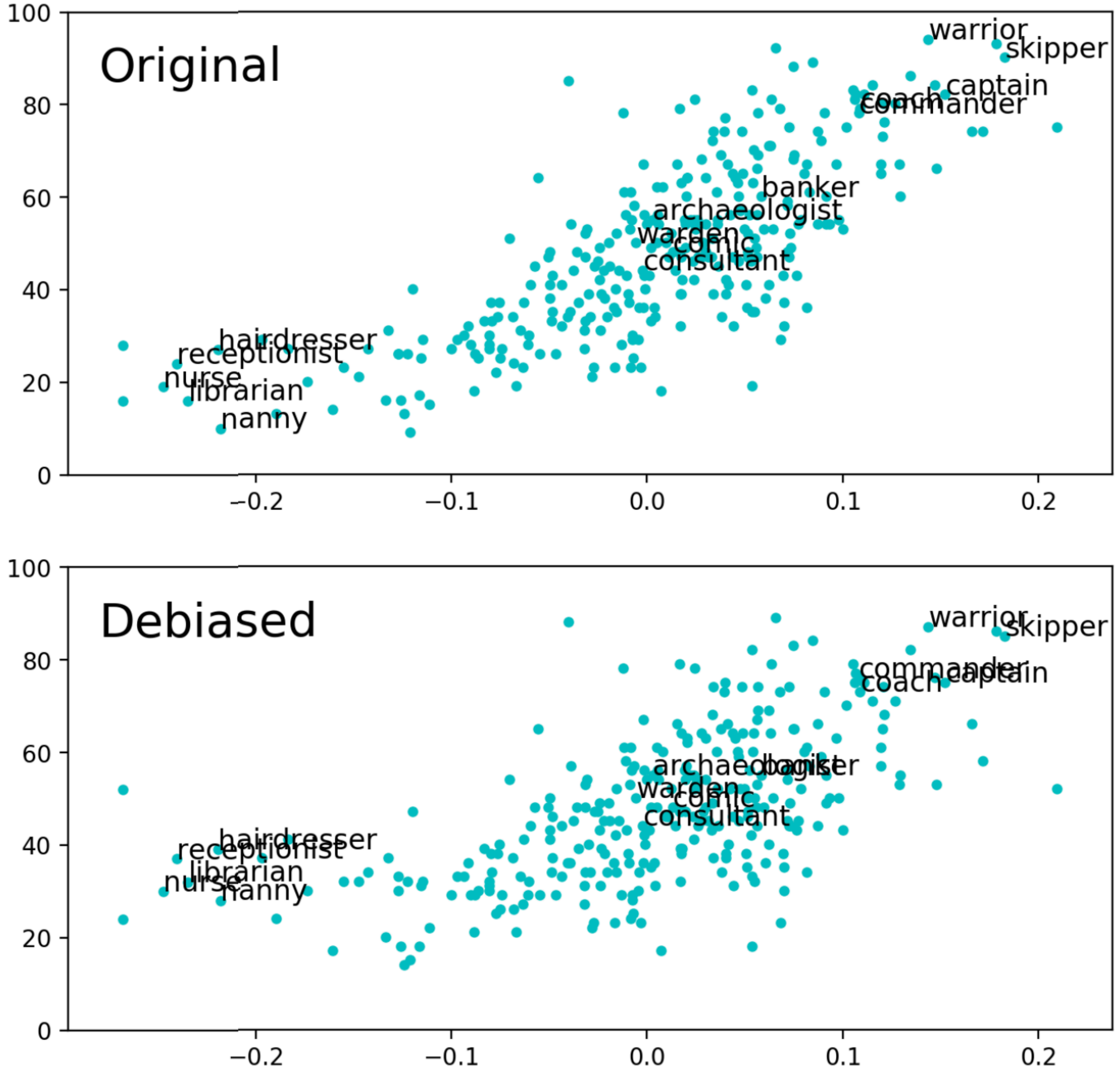


Bolukbasi et al. (2016)

Difficulty of Debiasing

- Not that effective…and the male and female words are still clustered together

- Bias pervades the word embedding space and isn’t just a local property of a few words



(a) The plots for HARD-DEBIASED embedding, before (top) and after (bottom) debiasing.

Gonen and Goldberg (2019)

Takeaways

- Lots to tune with neural networks
  - Training: optimizer, initializer, regularization (dropout), …
  - Hyperparameters: dimensionality of word embeddings, layers, …

- Lots of pretrained embeddings work well in practice, they capture some desirable properties

Implementation Details & Training Tips

You should read the following slides for HW3 later

Computation Graphs

- Computing gradients is hard! Computation graph abstraction allows us to define a computation symbolically and will do this for us

- Automatic differentiation: keep track of derivatives / be able to backpropagate through each function:

```
y = x * x    →    (y,dy) = (x * x, 2 * x * dx)
```

- Use a library like Pytorch or Tensorflow. This class: Pytorch

https://pytorch.org/tutorials/

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

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

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
    Take step with optimizer
Decode test set
```

Training Tips

Batching

- Batching data gives speedups due to more efficient matrix operations

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

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

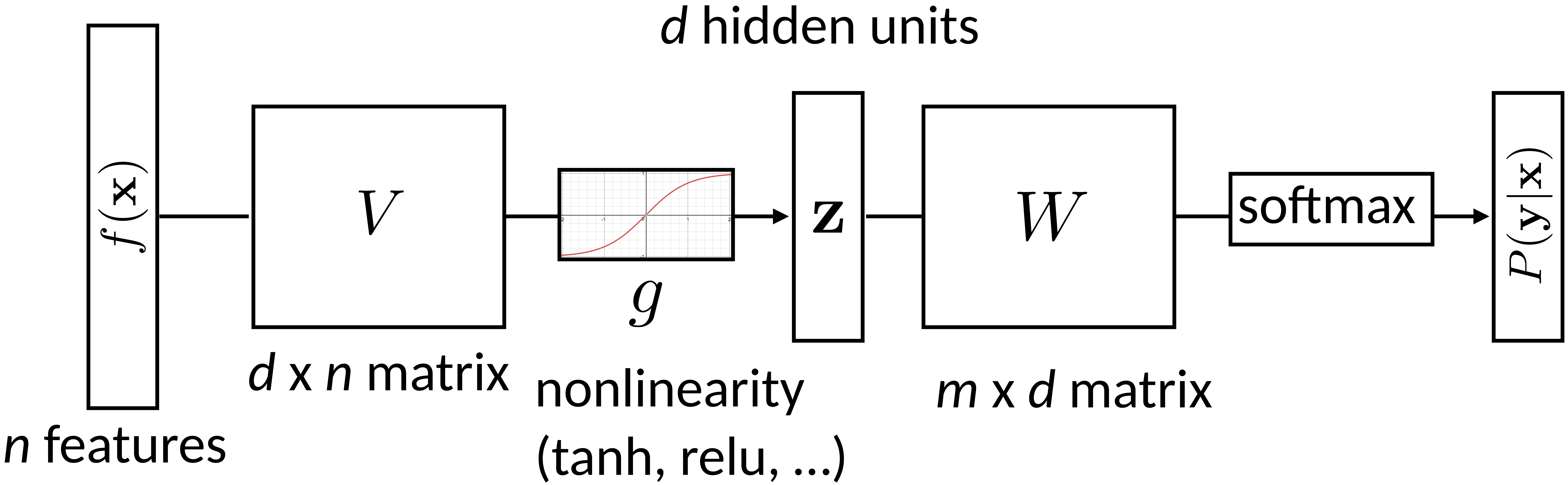
- Batch sizes from 1-100 often work well

Training Basics

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

How does initialization affect learning?

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



$f(\mathbf{x})$

n features

V

d x *n* matrix

g

nonlinearity

(tanh, relu, …)

d hidden units

$\mathbf{z}$

W

m x *d* matrix

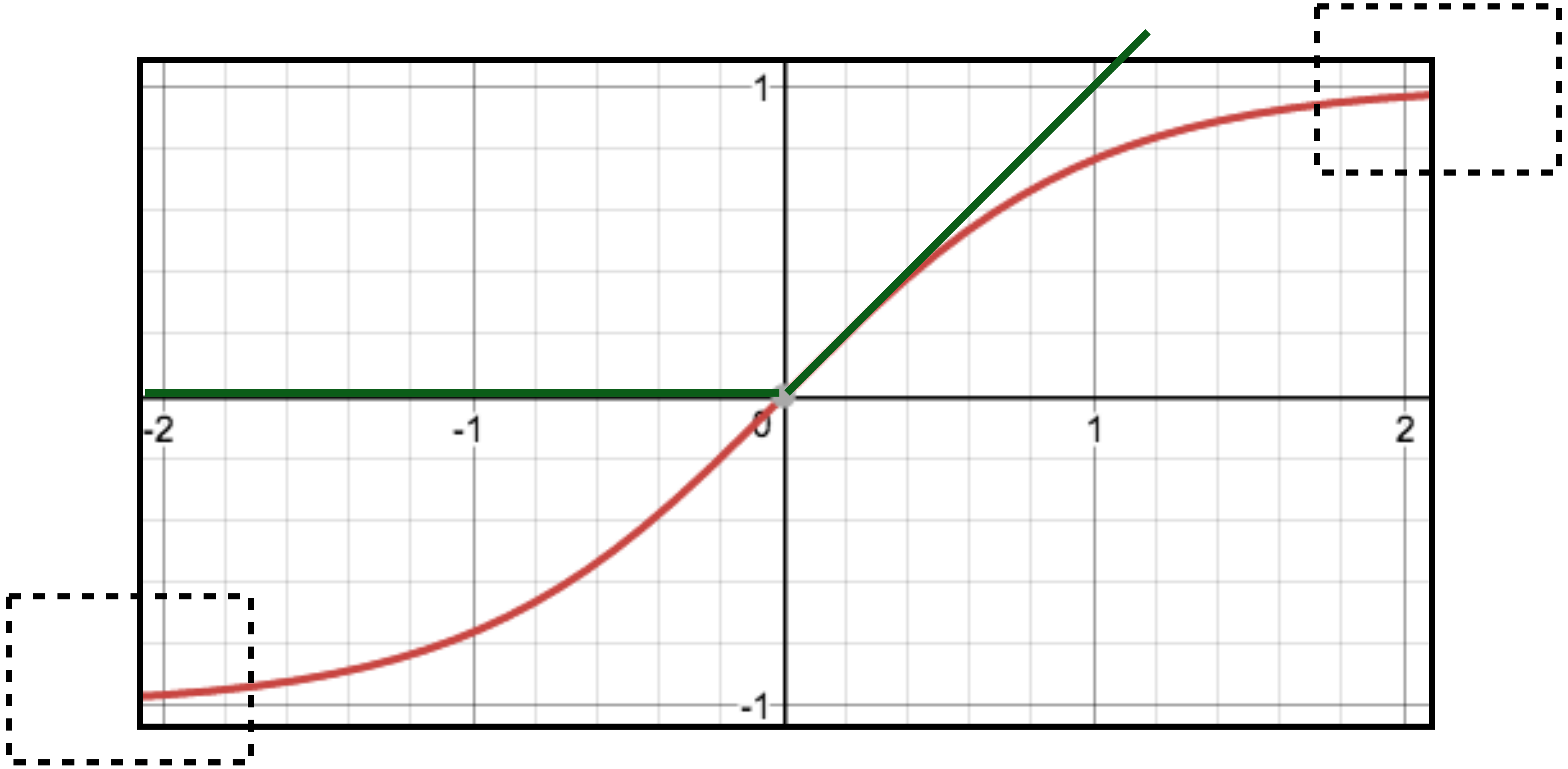
softmax

$P(\mathbf{y}|\mathbf{x})$

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

How does initialization affect learning?

- Nonlinear model…how does this affect things?



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

Initialization

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

2) Initialize too large and cells are saturated

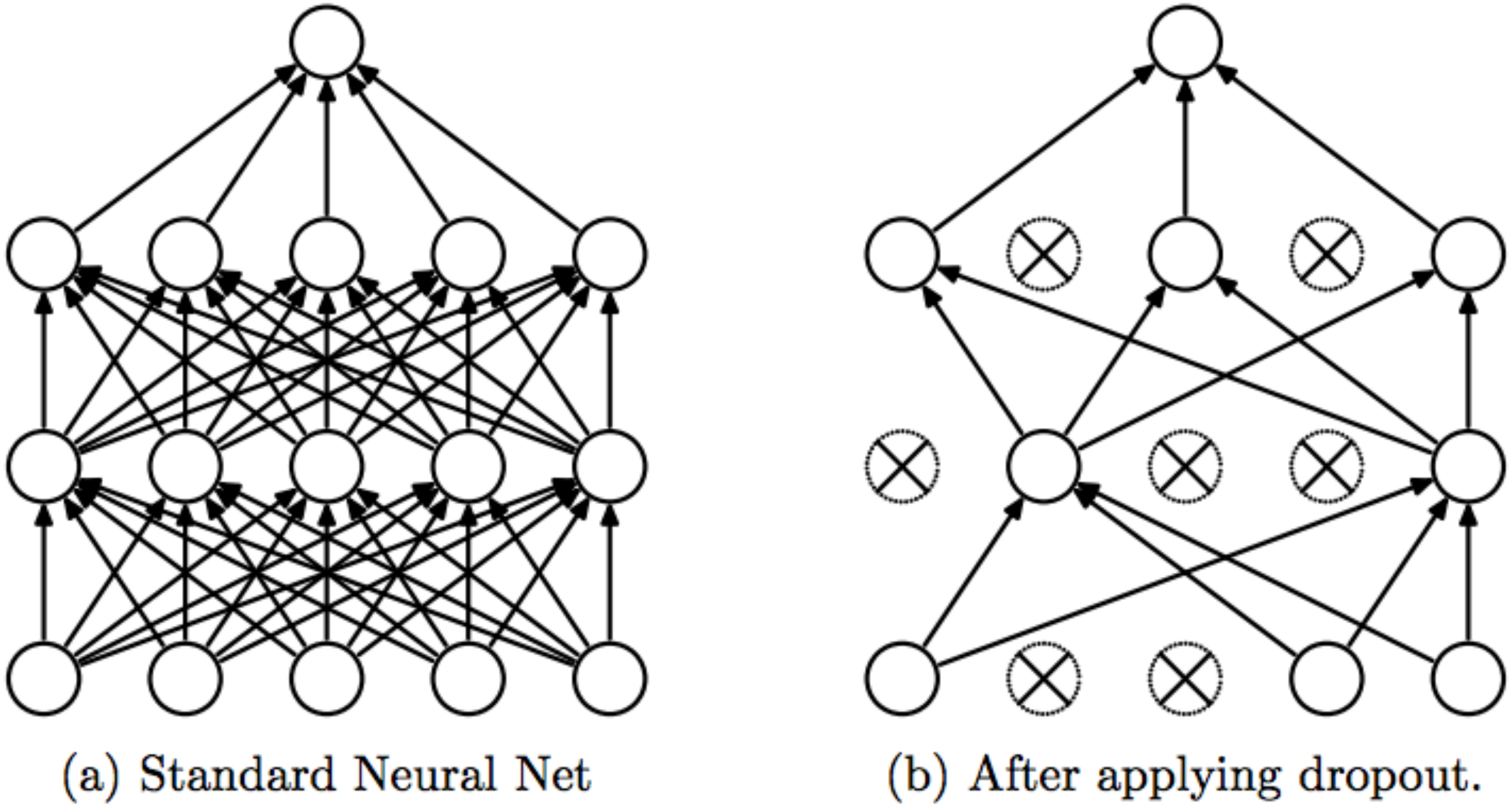
- Can do random uniform / normal initialization with appropriate scale
- Glorot initializer: $U\left[-\sqrt{\frac{6}{\text{fan-in}+\text{fan-out}}},+\sqrt{\frac{6}{\text{fan-in}+\text{fan-out}}}\right]$
  - Want variance of inputs and gradients for each layer to be the same
- Batch normalization (Ioffe and Szegedy, 2015): periodically shift+rescale each layer to have mean 0 and variance 1 over a batch (useful if net is deep)

Dropout

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

- Form of stochastic regularization

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

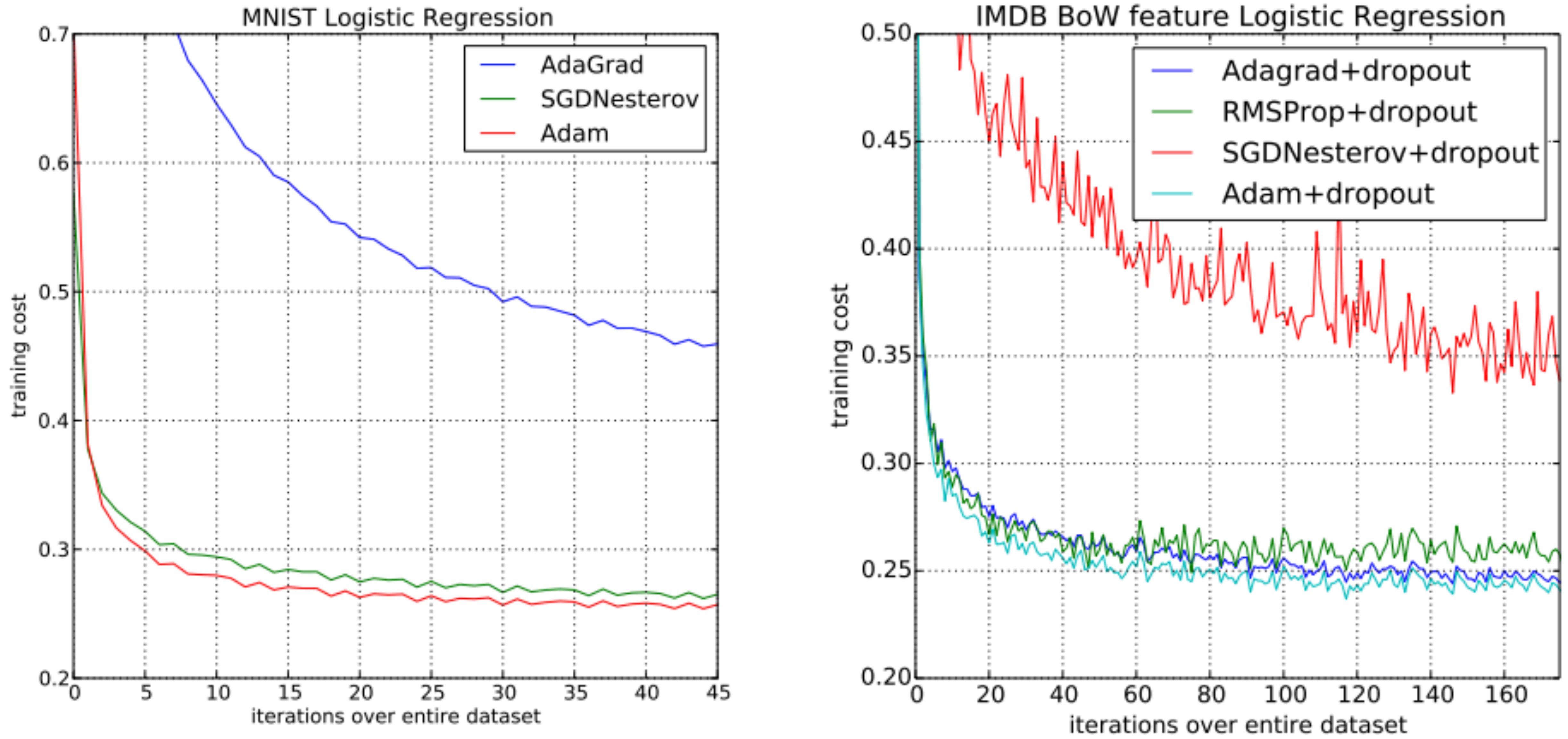


- One line in Pytorch/Tensorflow

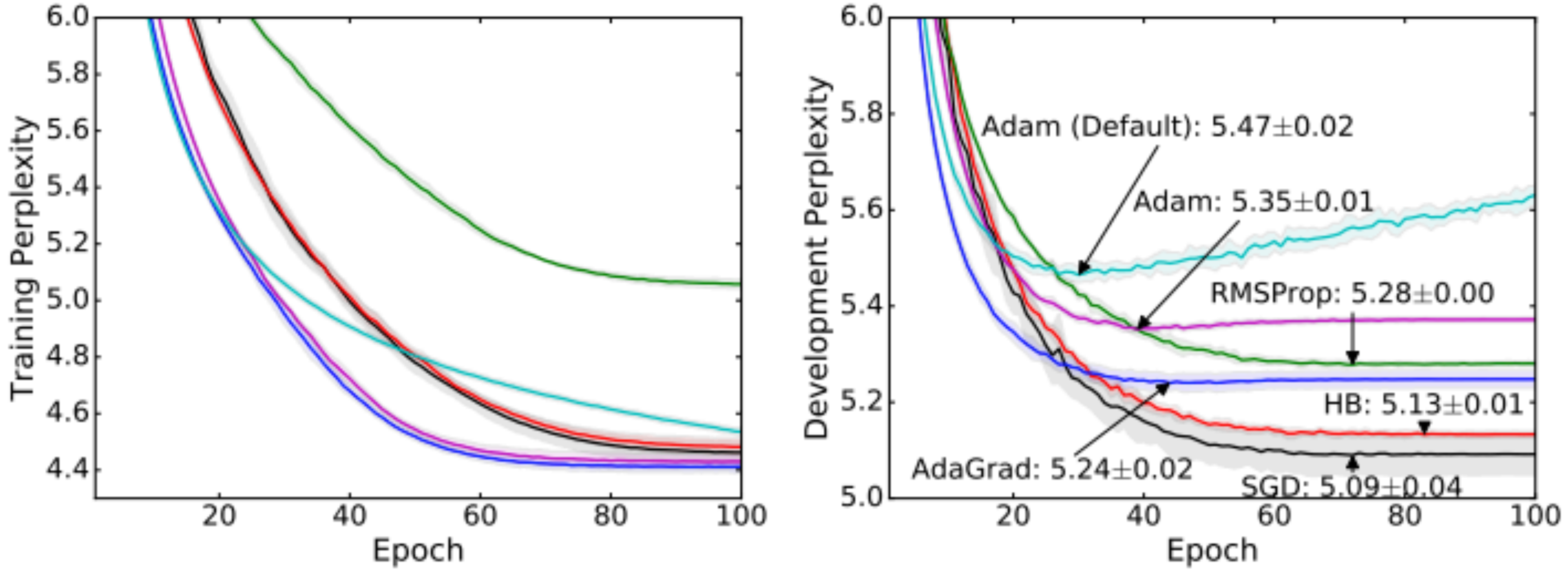
Srivastava et al. (2014)

Optimizer

- Adam (Kingma and Ba, ICLR 2015): very widely used. Adaptive step size + momentum



- Wilson et al. NIPS 2017: adaptive methods can actually perform badly at test time (Adam is in pink, SGD in black)



(e) Generative Parsing (Training Set)

(f) Generative Parsing (Development Set)

- One more trick: **gradient clipping** (set a max value for your gradients)