COS 484/584

(Advanced) Natural Language Processing

L7: Word Embeddings (cont'd) + Feedforward Neural Networks

Spring 2021

Recap: Skip-gram

center word x context word c

(apricot, tablespoon)

$$P(c \mid x) = \frac{\exp(\mathbf{u}_x \cdot \mathbf{v}_c)}{\sum_{k \in V} \exp(\mathbf{u}_x \cdot \mathbf{v}_k)}$$

$\mathbf{u}_{\text{apricot}}$

$$\begin{bmatrix} \mathbf{v}_{\text{aardvark}} \\ \mathbf{v}_{\text{a}} \\ \cdots \\ \mathbf{v}_{\text{tablespoon}} \\ \cdots \\ \mathbf{v}_{\text{zebra}} \end{bmatrix}$$

Similar to multinomial logistic regression (|V|-way classification)!

Skip-gram with negative sampling

center word x context word c

+: (apricot, tablespoon) → $P(y = 1 \mid x, c) = \sigma(\mathbf{u}_x \cdot \mathbf{v}_c)$

-: (apricot, aardvark) → $P(y = 0 \mid x, c') = 1 - \sigma(\mathbf{u}_x \cdot \mathbf{v}_{c'})$

-: (apricot, seven)

-: (apricot, my)

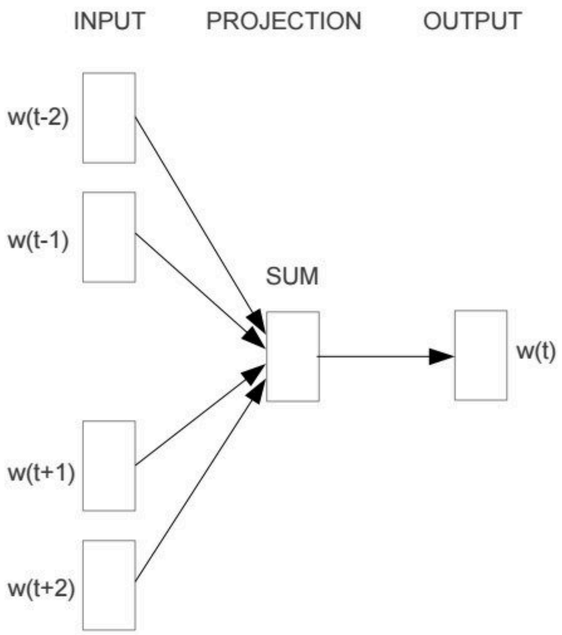
-: (apricot, where)

-: (apricot, dear)

$$= \sigma(-\mathbf{u}_x \cdot \mathbf{v}_{c'})$$

Similar to (binary) logistic regression!

Continuous Bag of Words (CBOW)



$$L(\theta) = \prod_{t=1}^{T} P\left(w_t \mid \{w_{t+j}\}, -m \leq j \leq m, j \neq 0\right)$$

$\mathbf{u}_k \in \mathbb{R}^d$: embedding for center word k, $\forall k \in V$

$\mathbf{v}_k \in \mathbb{R}^d$: embedding for context word k, $\forall k \in V$

$$\bar{\mathbf{v}}_t = \frac{1}{2m} \sum_{-m \leq j \leq m, j \neq 0} \mathbf{v}_{w_{t+j}}$$

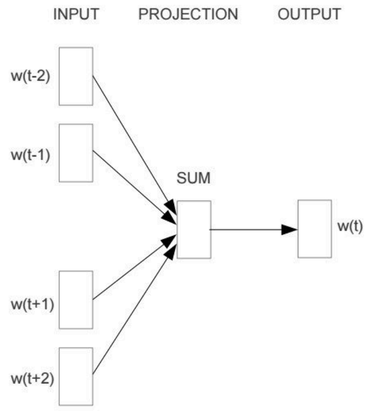
$$P(w_t \mid \{w_{t+j}\}) = \frac{\exp(\mathbf{u}_{w_t} \cdot \bar{\mathbf{v}}_t)}{\sum_{k \in V} \exp(\mathbf{u}_k \cdot \bar{\mathbf{v}}_t)}$$

Zoom poll

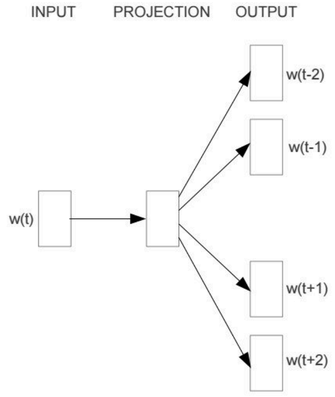
Let's make a comparison between skip-gram and CBOW. Which of the following is correct?

(a) Skip-gram is a simpler task compared to CBOW
(b) Skip-gram is faster to train than CBOW
(c) Skip-gram works better than CBOW for rare words
(d) None of the above

The answer is (c). See the next slide.



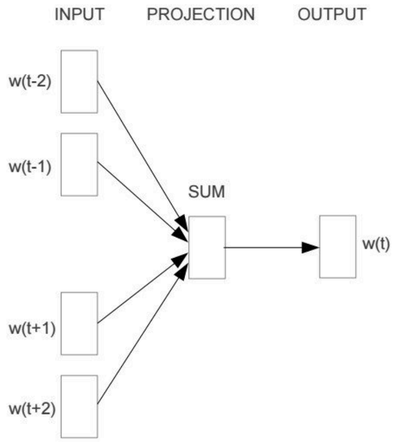
Continuous Bag of Words (CBOW)



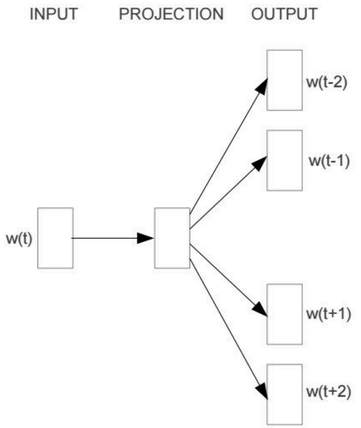
Skip-gram

Skip-gram vs CBOW

- CBOW is comparatively faster to train than skip-gram
- CBOW is a **simpler** problem than Skip-gram because in CBOW we just need to predict the one center word given many context words.
- Skip-gram is slower but works well for smaller amount of data and works well for less frequently occurring words



Continuous Bag of Words (CBOW)



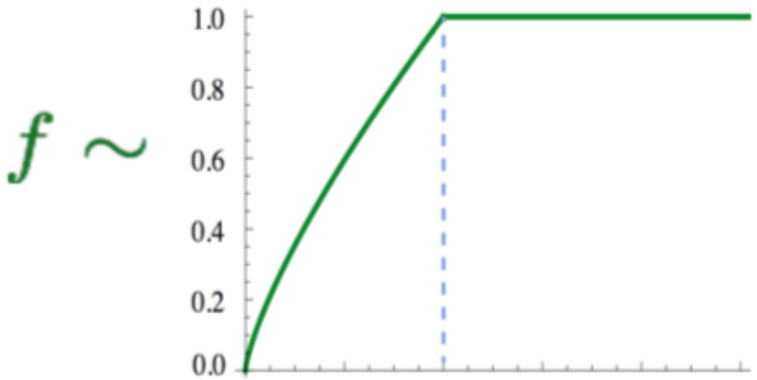
Skip-gram

GloVe: **Glo**bal **Ve**ctors [advanced]

- Key idea: let's approximate $\mathbf{u}_i \cdot \mathbf{v}_j$ using their co-occurrence counts directly.
- Take the global co-occurrence statistics: $X_{i,j}$

$$J(\theta) = \sum_{i,j \in V} f(X_{i,j}) \left(\mathbf{u}_i \cdot \mathbf{v}_j + b_i + \tilde{b}_j - \log X_{i,j} \right)^2$$

- Training faster
- Scalable to very large corpora



(Pennington et al, 2014): GloVe: Global Vectors for Word Representation

FastText: sub-word embeddings [advanced]

- Similar to skip-gram, but break words into n-grams with n = 3 to 6

<where>: 3-grams: <wh, whe, her, ere, re>

4-grams: <whe, wher, here, ere>

5-grams: <wher, where, here>

6-grams: <where, where>

- Replace $\mathbf{u}_x \cdot \mathbf{v}_c$ by $\sum_{g \in n\text{-grams}(x)} \mathbf{u}_g \cdot \mathbf{v}_c$

(Bojanowski et al, 2017): Enriching Word Vectors with Subword Information

Trained word embeddings available

- word2vec: https://code.google.com/archive/p/word2vec/
- GloVe: https://nlp.stanford.edu/projects/glove/
- FastText: https://fasttext.cc/

Download pre-trained word vectors

- Pre-trained word vectors. This data is made available under the Public Domain Dedication and License v1.0 whose full text can be found at: http://www.opendatacommons.org/licenses/pddl/1.0/.
  - Wikipedia 2014 + Gigaword 5 (6B tokens, 400K vocab, uncased, 50d, 100d, 200d, & 300d vectors, 822 MB download): glove.6B.zip
  - Common Crawl (42B tokens, 1.9M vocab, uncased, 300d vectors, 1.75 GB download): glove.42B.300d.zip
  - Common Crawl (840B tokens, 2.2M vocab, cased, 300d vectors, 2.03 GB download): glove.840B.300d.zip
  - Twitter (2B tweets, 27B tokens, 1.2M vocab, uncased, 25d, 50d, 100d, & 200d vectors, 1.42 GB download): glove.twitter.27B.zip
- Ruby script for preprocessing Twitter data

Differ in algorithms, text corpora, dimensions, cased/uncased…

Applied to many other languages

Easy to use!

```python
from gensim.models import KeyedVectors
# Load vectors directly from the file
model = KeyedVectors.load_word2vec_format('data/GoogleGoogleNews-vectors-negative300.bin', binary=True)
# Access vectors for specific words with a keyed lookup:
vector = model['easy']
```

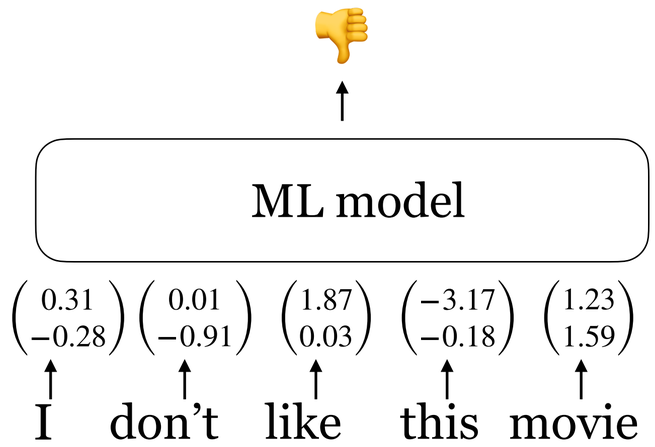
```
In [17]: model.similarity('straightforward', 'easy')
Out[17]: 0.5717043285477517

In [18]: model.similarity('simple', 'impossible')
Out[18]: 0.29156160264633707

In [19]: model.most_similar('simple')
Out[19]: [('straightforward', 0.7460169196128845),
          ('Simple', 0.7108174562454224),
          ('uncomplicated', 0.6297484636306763),
          ('simplest', 0.6171397566795349),
          ('easy', 0.5990299582481384),
          ('fairly_straightforward', 0.5893306732177734),
          ('deceptively_simple', 0.5743066072463989),
          ('simpler', 0.5537199378013611),
          ('simplistic', 0.5516539216041565),
          ('disarmingly_simple', 0.5365327000617981)]
```

How to evaluate word embeddings?

Extrinsic vs intrinsic evaluation

Extrinsic evaluation

- Let's plug these word embeddings into a real NLP system and see whether this improves performance
- Could take a long time but still the most important evaluation metric



Intrinsic evaluation

- Evaluate on a specific/intermediate subtask
- Fast to compute
- Not clear if it really helps downstream tasks

Intrinsic evaluation: word similarity

Word similarity

Example dataset: wordsim-353
353 pairs of words with human judgement
http://www.cs.technion.ac.il/~gabr/resources/data/wordsim353/

Word 1	Word 2	Human (mean)
tiger	cat	7.35
tiger	tiger	10
book	paper	7.46
computer	internet	7.58
plane	car	5.77
professor	doctor	6.62
stock	phone	1.62
stock	CD	1.31
stock	jaguar	0.92

Cosine similarity:

$$\cos(\boldsymbol{u}_i, \boldsymbol{u}_j) = \frac{\boldsymbol{u}_i \cdot \boldsymbol{u}_j}{||\boldsymbol{u}_i||_2 \times ||\boldsymbol{u}_j||_2}.$$

Metric: Spearman rank correlation

Intrinsic evaluation: word analogy

Word analogy

man: woman ≈ king: ? a : b ≈ c : ?

$$\arg\max_i \left(\cos(\mathbf{u}_i, \mathbf{u}_b - \mathbf{u}_a + \mathbf{u}_c)\right)$$

semantic

syntactic

Chicago:Illinois≈Philadelphia: ? bad:worst ≈ cool: ?

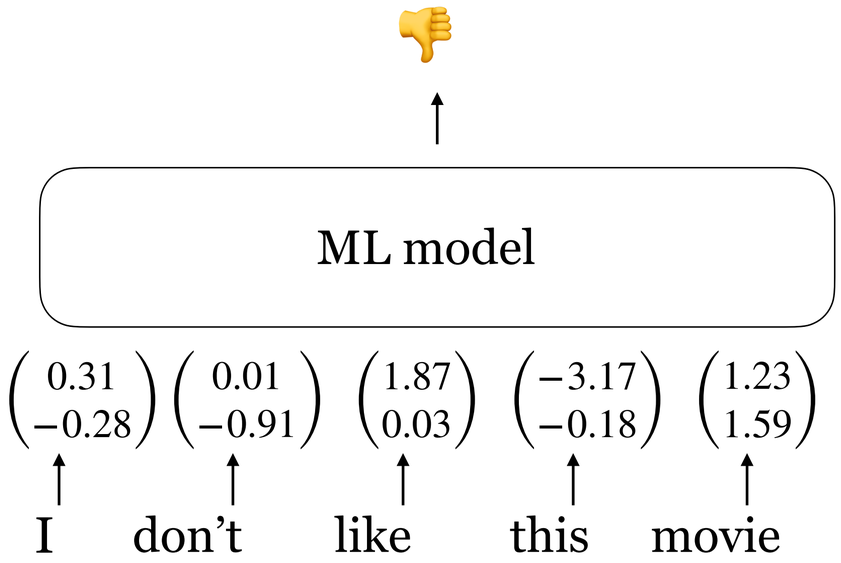
More examples at
http://download.tensorflow.org/data/questions-words.txt

Model	Dim.	Size	Sem.	Syn.	Tot.
ivLBL	100	1.5B	55.9	50.1	53.2
HPCA	100	1.6B	4.2	16.4	10.8
GloVe	100	1.6B	67.5	54.3	60.3
SG	300	1B	61	61	61
CBOW	300	1.6B	16.1	52.6	36.1
vLBL	300	1.5B	54.2	64.8	60.0
ivLBL	300	1.5B	65.2	63.0	64.0
GloVe	300	1.6B	80.8	61.5	70.3
SVD	300	6B	6.3	8.1	7.3
SVD-S	300	6B	36.7	46.6	42.1
SVD-L	300	6B	56.6	63.0	60.1
CBOW†	300	6B	63.6	67.4	65.7
SG†	300	6B	73.0	66.0	69.1
GloVe	300	6B	77.4	67.0	71.7
CBOW	1000	6B	57.3	68.9	63.7
SG	1000	6B	66.1	65.1	65.6
SVD-L	300	42B	38.4	58.2	49.2
GloVe	300	42B	**81.9**	**69.3**	**75.0**

From word embeddings to neural networks

$$v_{\text{cat}} = \begin{pmatrix} -0.224 \\ 0.130 \\ -0.290 \\ 0.276 \end{pmatrix} \qquad v_{\text{dog}} = \begin{pmatrix} -0.124 \\ 0.430 \\ -0.200 \\ 0.329 \end{pmatrix}$$

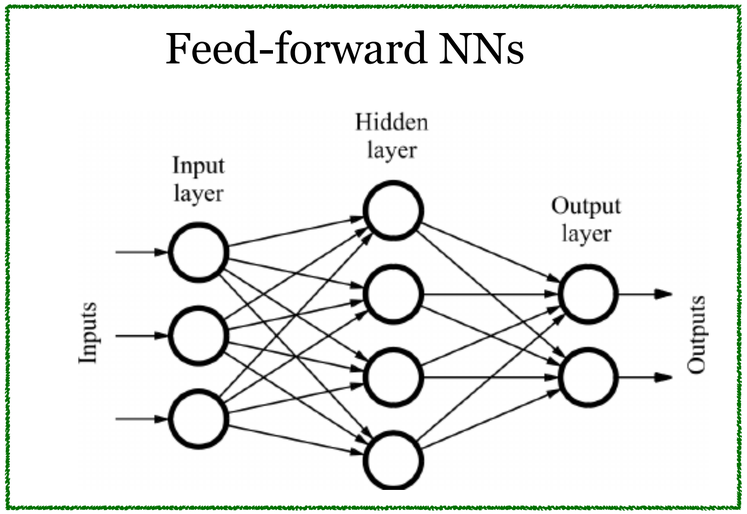
$$v_{\text{the}} = \begin{pmatrix} 0.234 \\ 0.266 \\ 0.239 \\ -0.199 \end{pmatrix} \qquad v_{\text{language}} = \begin{pmatrix} 0.290 \\ -0.441 \\ 0.762 \\ 0.982 \end{pmatrix}$$



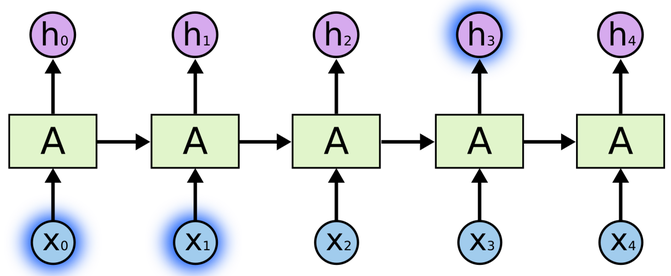
Neural networks in NLP

Always coupled with word embeddings…

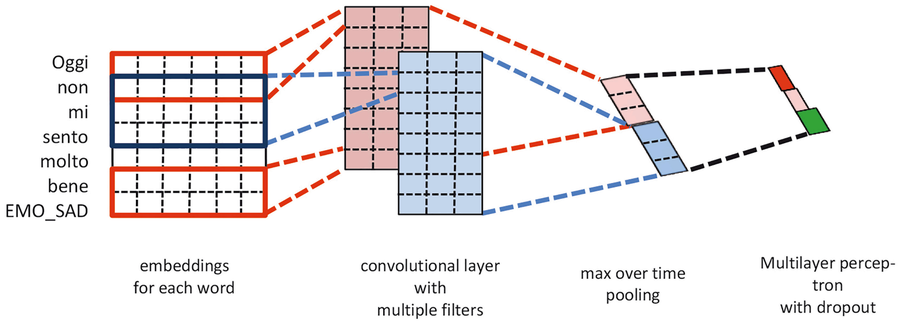
Feed-forward NNs



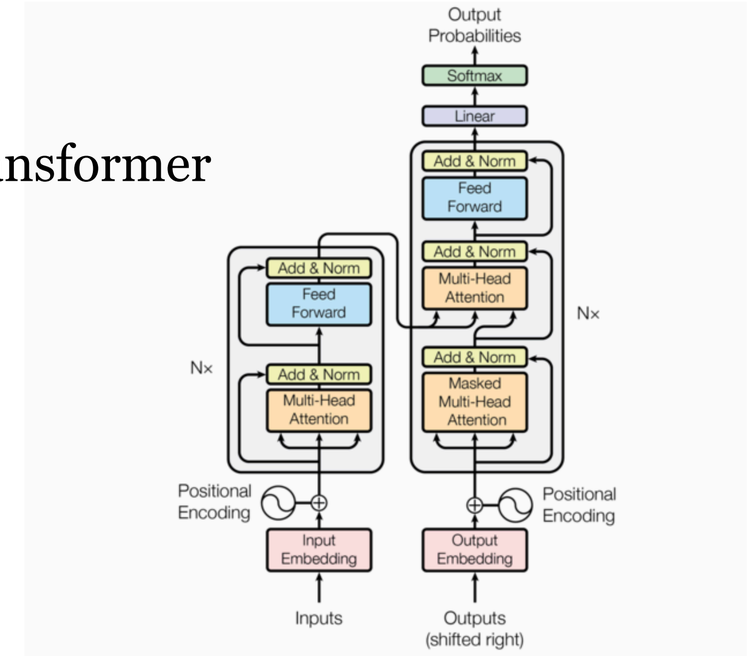
Recurrent NNs



Convolutional NNs



Transformer



Zoom poll

How much do you know about neural networks (or deep learning)?

(a) I have NOT heard of these concepts

(b) I have heard of feedforward NNs, LSTMs or Transformers but I don't understand their inner workings

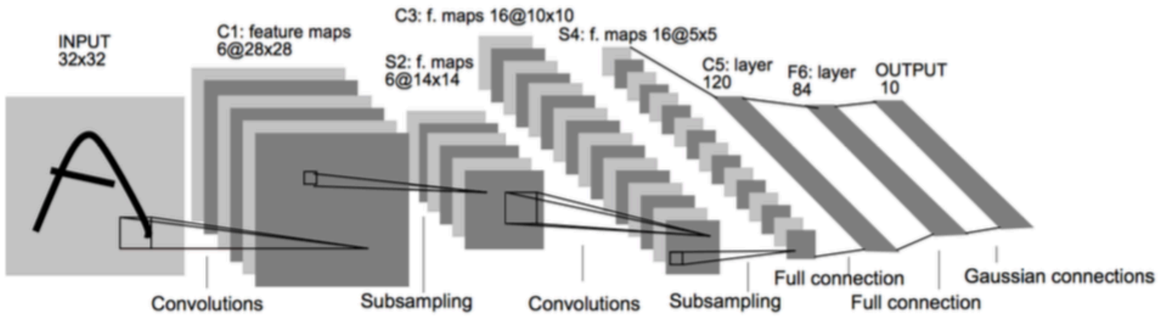
(c) I have used a deep learning library (e.g., PyTorch, Tensorflow) as a "blackbox"

(d) I have used neural networks extensively in my work and understand how backpropagation works.

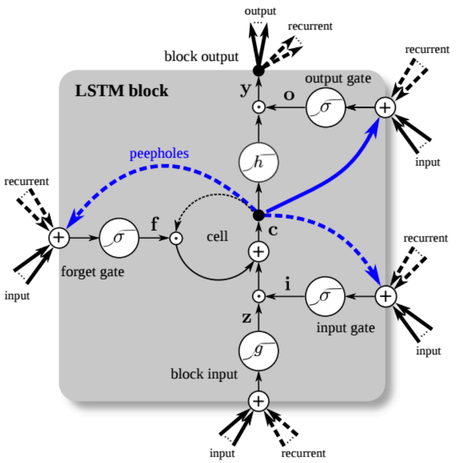
Neural Networks: History

NN “dark ages”

- Neural network algorithms date from the 80s
- ConvNets: applied to MNIST by LeCun in 1998



- Long Short-term Memory Networks (LSTMs): Hochreiter and Schmidhuber 1997



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

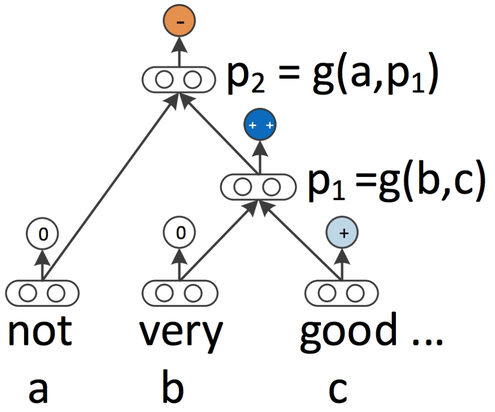
Slide credit: Greg Durrett

2008-2013: A glimmer of light

- Collobert and Weston 2011: "**NLP (almost) from Scratch**"
  - Feedforward NNs can replace "feature engineering"
  - 2008 version was marred by bad experiments, claimed SOTA but wasn't, 2011 version tied SOTA

- Krizhevskey et al, 2012: AlexNet for ImageNet Classification

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



Slide credit: Greg Durrett

2014: Stuff starts working

- Kim (2014) + Kalchbrenner et al, 2014: sentence classification
  - ConvNets work for NLP!
- Sutskever et al, 2014: sequence-to-sequence for neural MT
  - LSTMs work for NLP!
- Chen and Manning 2014: dependency parsing
  - Even feedforward networks work well for NLP!
- 2015: explosion of neural networks for everything under the sun
- 2018-2019: NLP has entered the era of pre-trained models (ELMo, BERT)

Slide credit: Greg Durrett

Why didn't they work before?

- **Datasets too small**: for machine translation, 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/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 embeddings** to represent continuous semantics

Slide credit: Greg Durrett

The "Promise" of deep learning

- Most NLP works in the past focused on human-designed representations and input features

Var	Definition	Value in Fig. 5.2
x_1	count(positive lexicon) $\in$ doc)	3
x_2	count(negative lexicon) $\in$ doc)	2
x_3	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns $\in$ doc)	3
x_5	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	log(word count of doc)	$\ln(64) = 4.15$

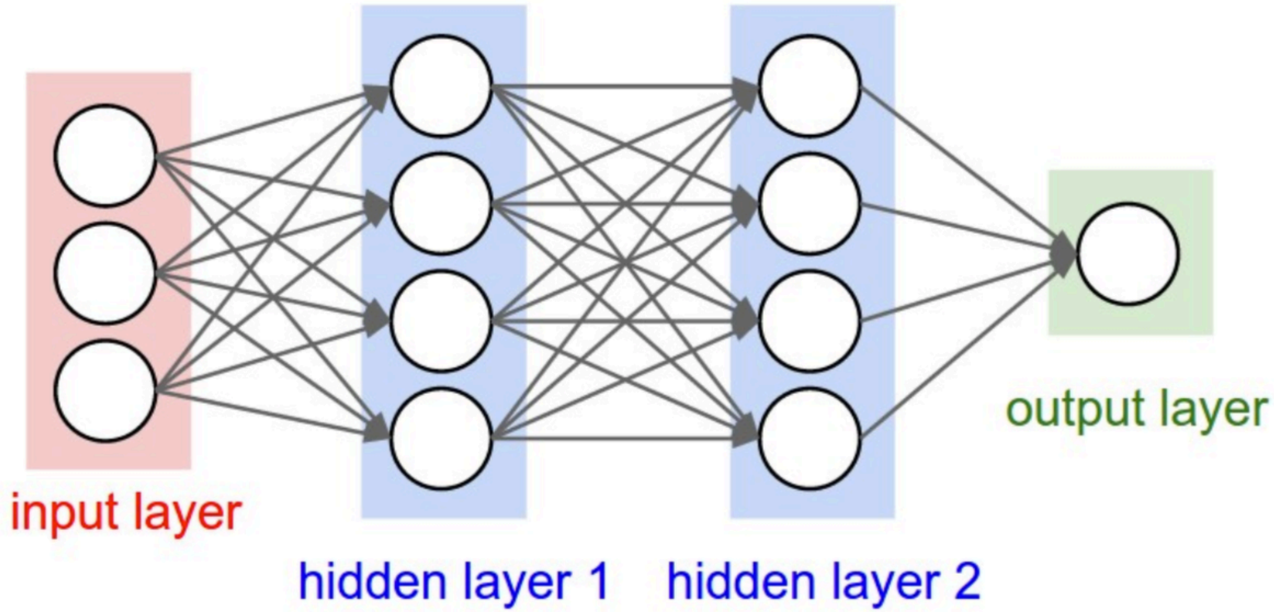
- **Representation learning** attempts to automatically learn good features and representations

- **Deep learning** attempts to learn multiple levels of representations on increasing complexity/abstraction

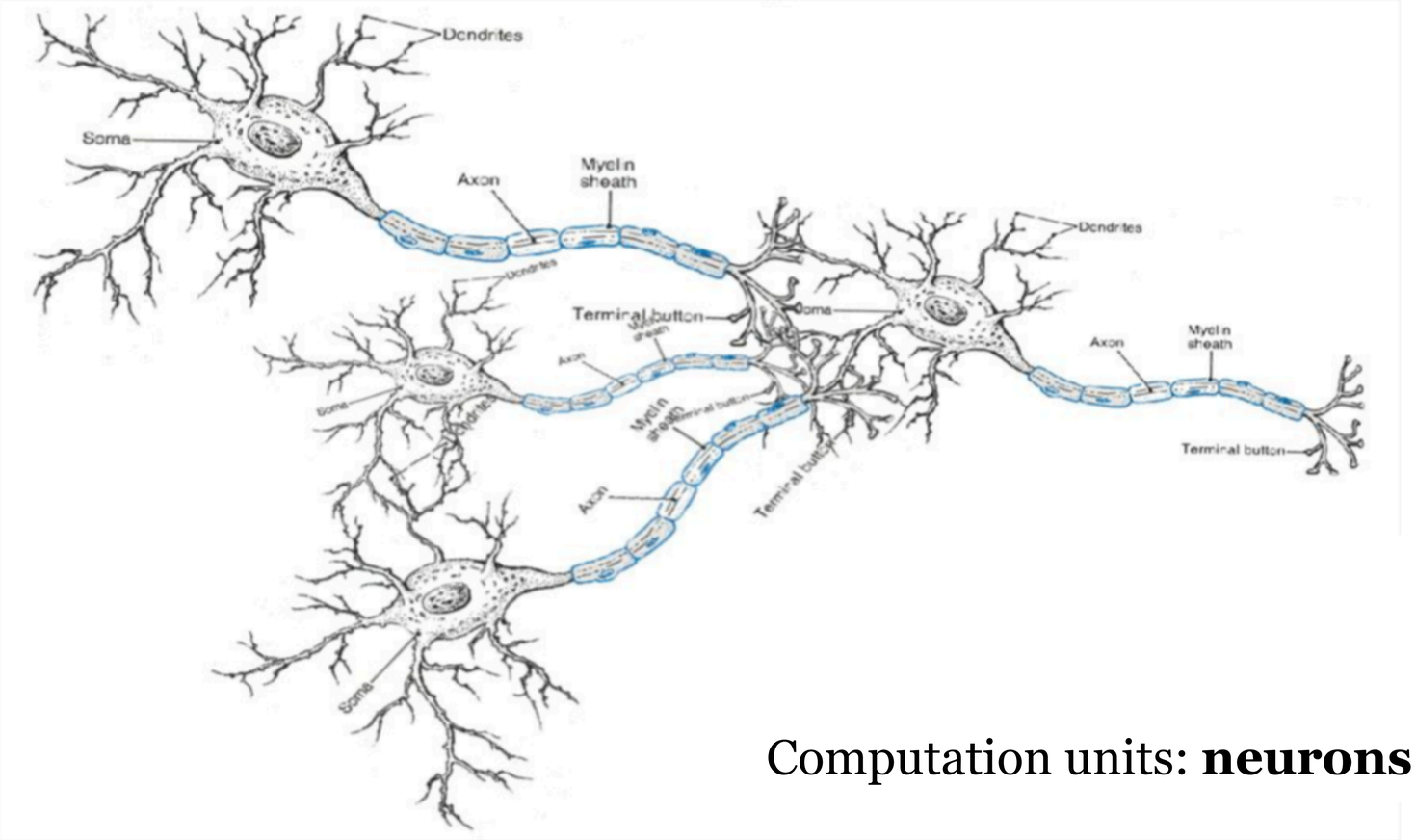
Feedforward Neural Networks

Feed-forward NNs

- Input: $x_1, \ldots, x_d \in \mathbb{R}$
- Output: $y^* \in \{0,1\}$



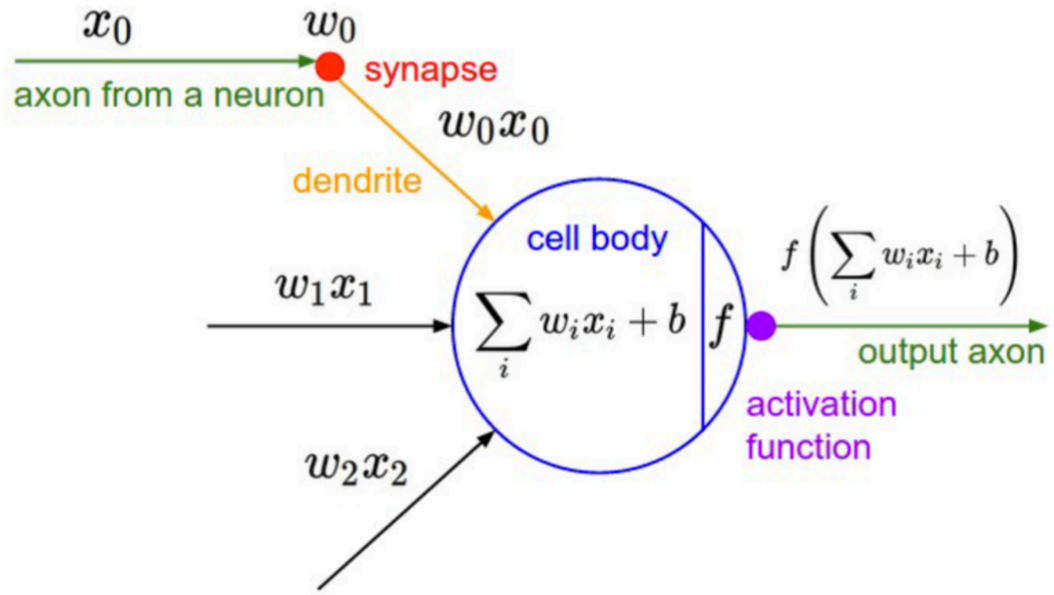
Neural computation



Computation units: **neurons**

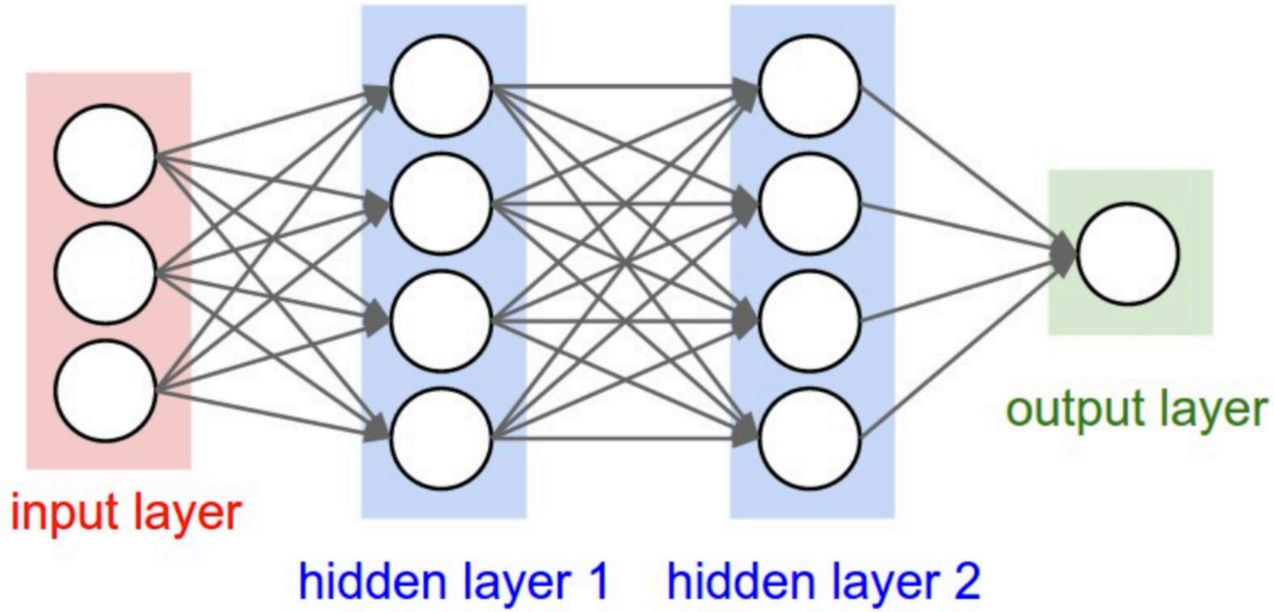
An artificial neuron

- A neuron is a computational unit that has scalar inputs and an output
- Each input has an associated weight.
- The neuron multiples each input by its weight, sums them, applied a **nonlinear activation function** to the result, and passes it to its output.

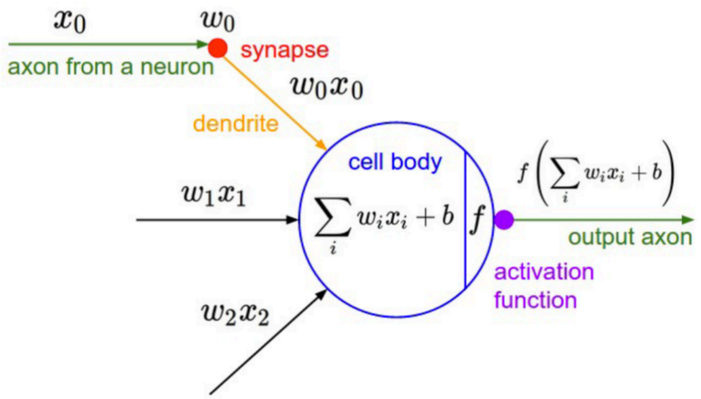


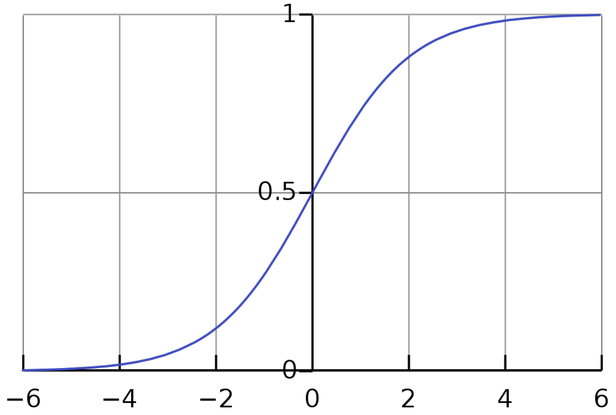
Neural networks

- The neurons are connected to each other, forming a **network**
- The output of a neuron may feed into the inputs of other neurons



A neuron can be a binary logistic regression unit





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

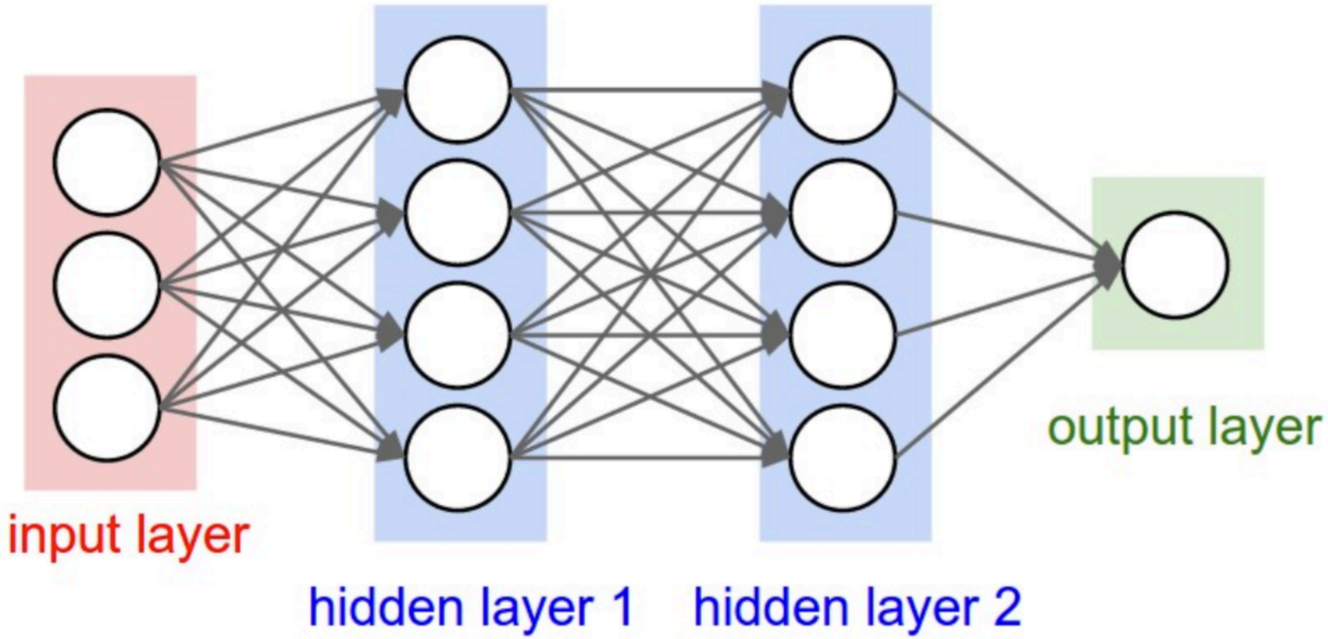
$$h_{\mathbf{w},b}(\mathbf{x}) = f(\mathbf{w}^\intercal \mathbf{x} + b)$$

$$w = [0.2, 0.3, 0.9]$$
$$b = 0.5$$
$$x = [0.5, 0.6, 0.1]$$

$$y = \sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} = \frac{1}{1 + e^{-(.5*.2 + .6*.3 + .1*.9 + .5)}} = \frac{1}{1 + e^{-0.87}} = .70$$

We are going to use *f* as the activation function—f can take in σ and other forms!

A neural network
= running several logistic regressions at the same time



If we feed a vector of inputs through a bunch of logistic regression functions, then we get a vector of outputs, which we can feed into another logistic regression function

Mathematical notations

- Input layer: $x_1, \ldots, x_d$
- Hidden layer 1: $h_1^{(1)}, h_2^{(1)}, \ldots, h_{d_1}^{(1)}$

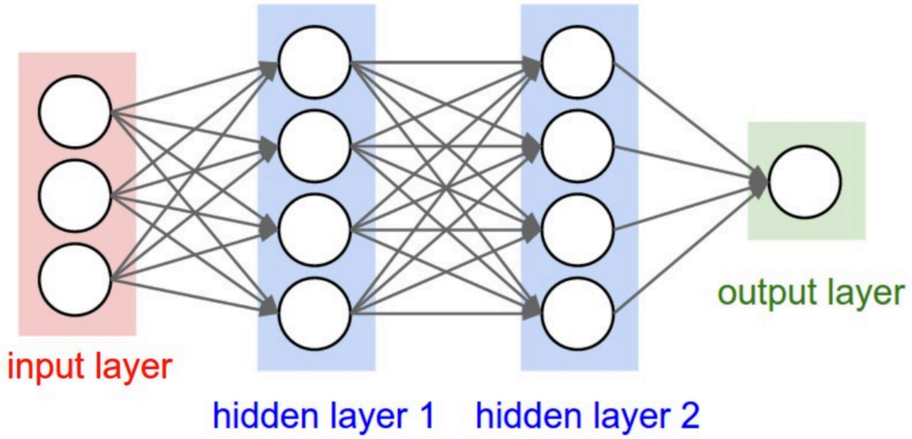
$$h_1^{(1)} = f(W_{1,1}^{(1)} + W_{1,2}^{(1)} x_2 + \ldots + W_{1,d}^{(1)} x_d + b_1^{(1)})$$
$$h_2^{(1)} = f(W_{2,1}^{(1)} + W_{2,2}^{(1)} x_2 + \ldots + W_{2,d}^{(1)} x_d + b_2^{(1)})$$
$$\ldots$$

- Hidden layer 2: $h_1^{(2)}, h_2^{(2)}, \ldots, h_{d_2}^{(2)}$

$$h_1^{(2)} = f(W_{1,1}^{(2)} h_1^{(1)} + W_{1,2}^{(2)} h_2^{(1)} + \ldots + W_{1,d_1}^{(2)} h_{d_1}^{(1)} + b_1^{(2)})$$
$$h_2^{(2)} = f(W_{2,1}^{(2)} h_1^{(1)} + W_{2,2}^{(2)} h_2^{(1)} + \ldots + W_{2,d_1}^{(2)} h_{d_1}^{(1)} + b_2^{(2)})$$
$$\ldots$$

- Output layer:

$$y = \sigma(w_1^{(o)} h_1^{(2)} + w_2^{(o)} h_2^{(2)} + \ldots + w_{d_2}^{(o)} h_{d_2}^{(2)} + b^{(o)})$$



Matrix notations

- Input layer: $\mathbf{x} \in \mathbb{R}^d$
- Hidden layer 1:

$$\mathbf{h}_1 = f(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \in \mathbb{R}^{d_1}$$

$$\mathbf{W}^{(1)} \in \mathbb{R}^{d_1 \times d}, \mathbf{b}^{(1)} \in \mathbb{R}^{d_1}$$

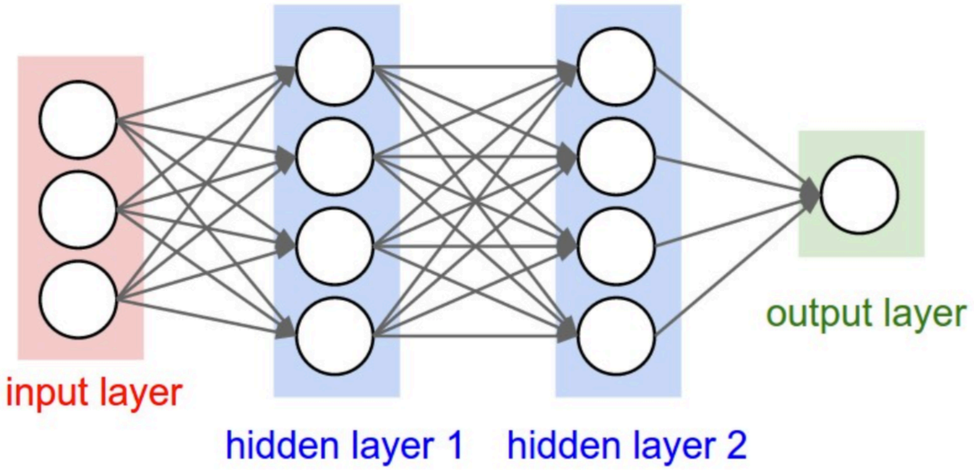
- Hidden layer 2:

$$\mathbf{h}_2 = f(\mathbf{W}^{(2)}\mathbf{h}_1 + \mathbf{b}^{(2)}) \in \mathbb{R}^{d_2}$$

$$\mathbf{W}^{(2)} \in \mathbb{R}^{d_2 \times d_1}, \mathbf{b}^{(2)} \in \mathbb{R}^{d_2}$$

- Output layer:

$$y = \sigma(\mathbf{w}^{(o)} \cdot \mathbf{h}_2 + b^{(o)})$$

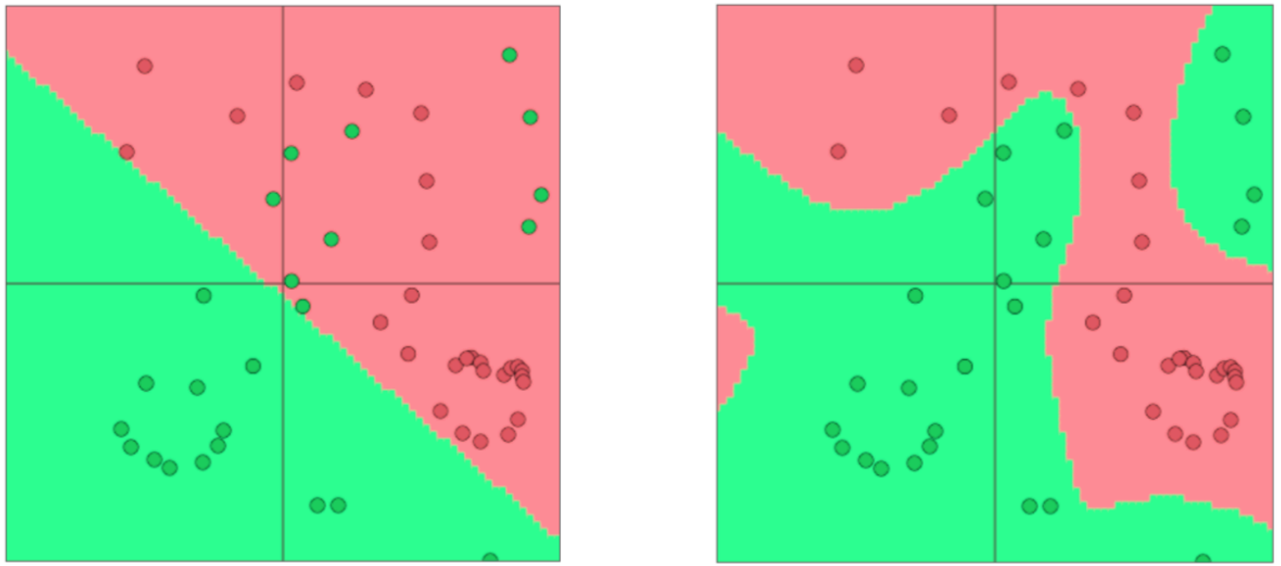


*: f is applied element-wise

$$f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$$

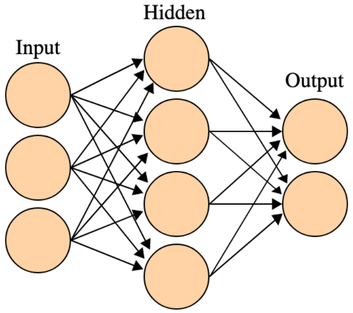
Why non-linearities?

Neural networks can learn much more complex functions and nonlinear decision boundaries

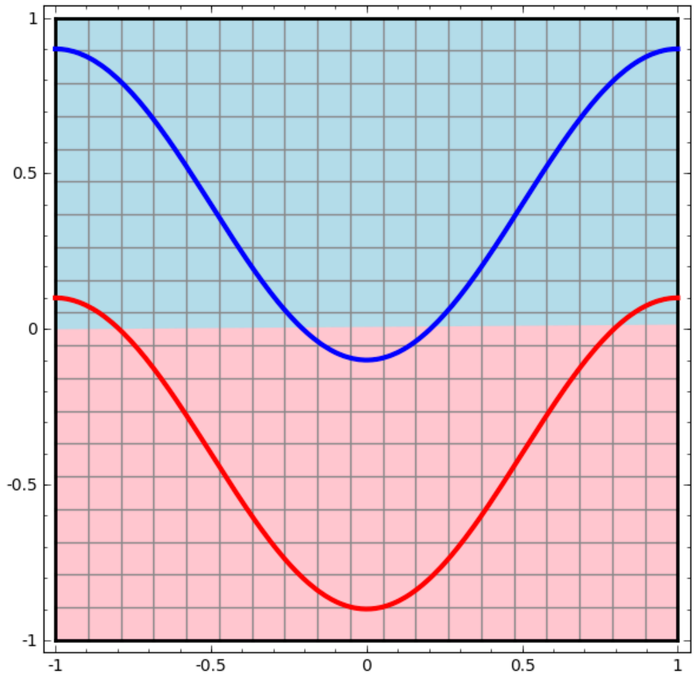


The capacity of the network increases with more hidden units and more hidden layers

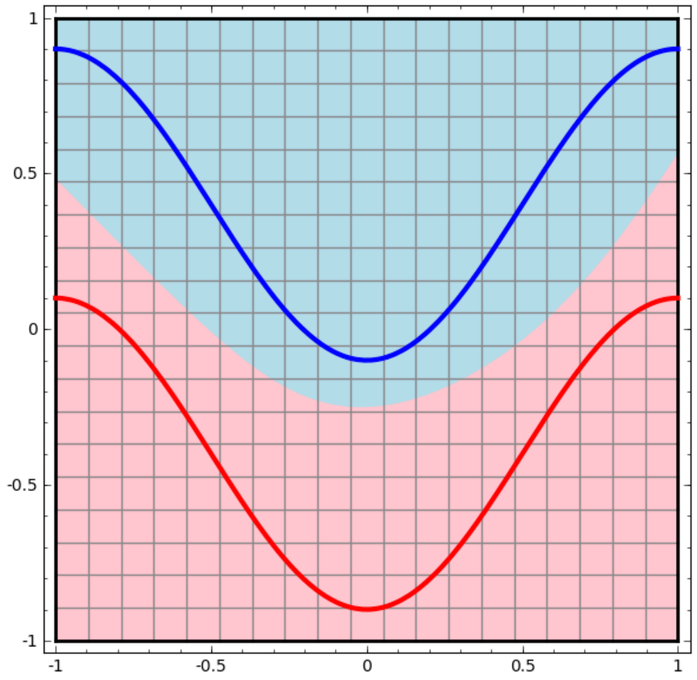
Q: How if we remove activation function f?



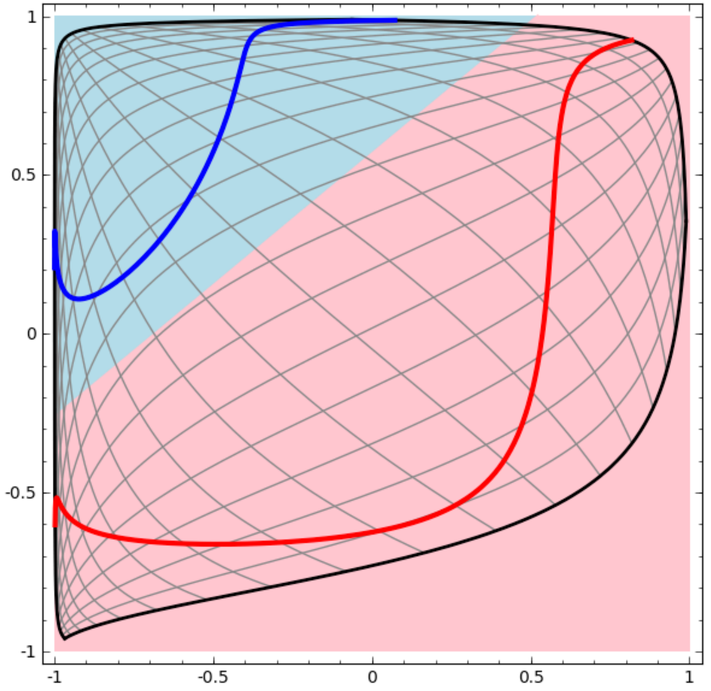
Why non-linearities?



Linear classifier



Neural Networks



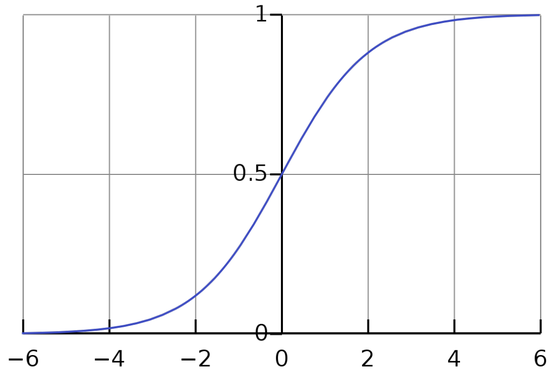
Hidden representations
are linearly separable!

https://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Activation functions

sigmoid

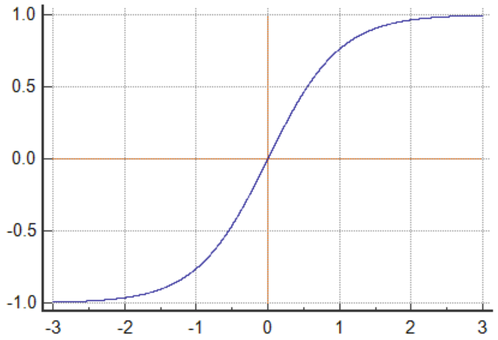
$$f(z) = \frac{1}{1 + e^{-z}}$$



$$f'(z) = f(z) \times (1 - f(z))$$

tanh

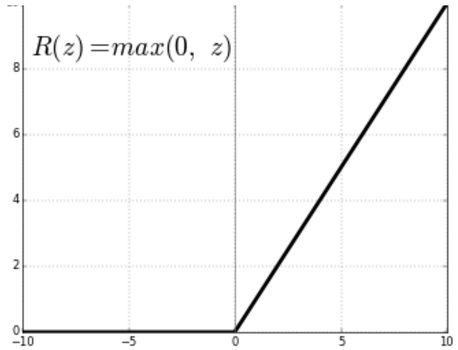
$$f(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$



$$f'(z) = 1 - f(z)^2$$

ReLU
(rectified linear unit)

$$f(z) = \max(0, z)$$



$$f'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$$

Q: Advantages of ReLU?

How to train neural networks?

Binary classification: $y^* = \{0,1\}$

$$y = \sigma(\mathbf{w}^{(o)} \cdot \mathbf{h}_2 + b^{(o)})$$

$$L_{CE} = -\sum_{i=1}^{n} [y^* \log \hat{y} + (1 - y^*)\log(1 - y)]$$

Multi-class classification: $y^* = \{1,2,\ldots,C\}$

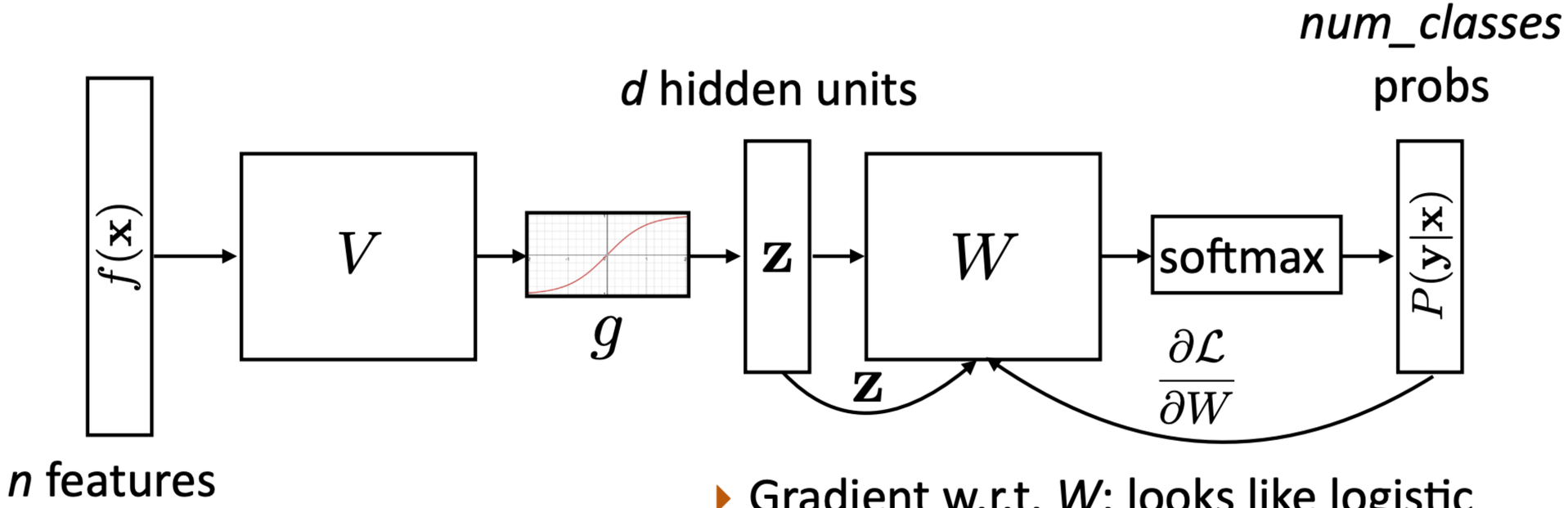
$$y = \text{softmax}(\mathbf{W}^{(o)}\mathbf{h}_2 + \mathbf{b}^{(o)}) \qquad \mathbf{W}^{(o)} \in \mathbb{R}^{C \times h}, \mathbf{b}^{(o)} \in \mathbb{R}^{C}$$

$x = h_2$ in this example

$$L_{CE}(y, y^*) = -\sum_{c=1}^{C} 1\{y^* = c\}\log \frac{e^{w_c \cdot x + b_c}}{\sum_{j=1}^{k} e^{w_j \cdot x + b_j}}$$

Backpropagation [advanced]

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

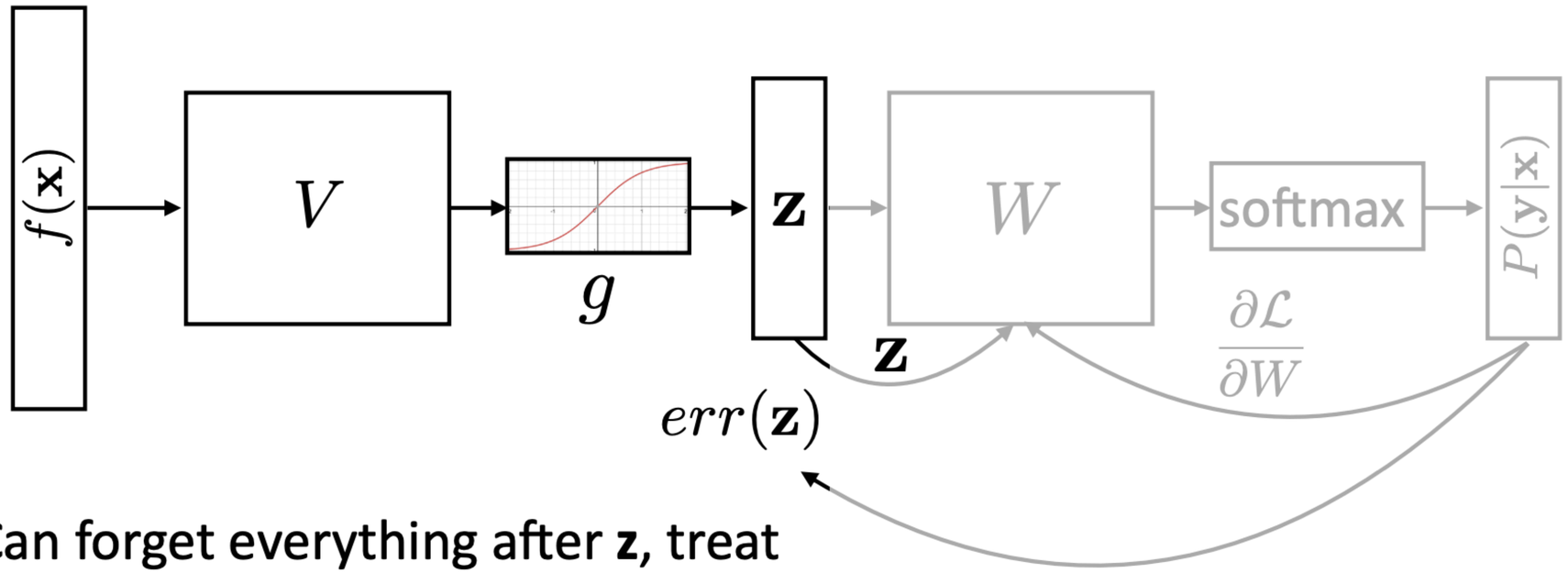


- Gradient w.r.t. *W*: looks like logistic regression, can be computed treating **z** as the features

Slide credit: Greg Durrett

Backpropagation [advanced]

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



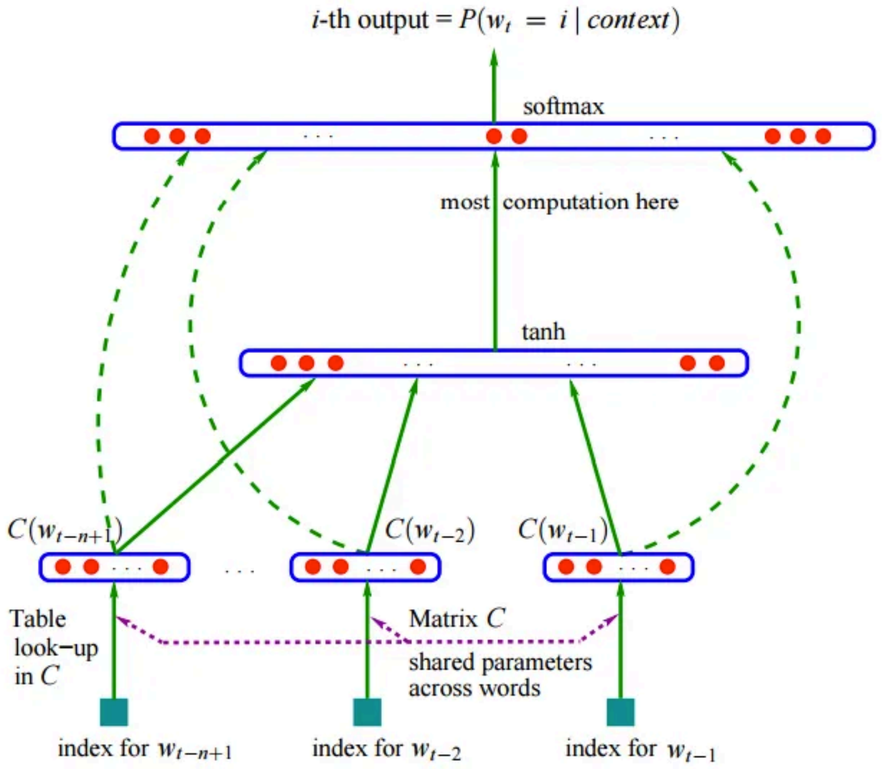
- Can forget everything after **z**, treat it as the output and keep backpropping

Good news is that modern automatic differentiation tools did all for you!

Slide credit: Greg Durrett

Application: Feedforward neural language models

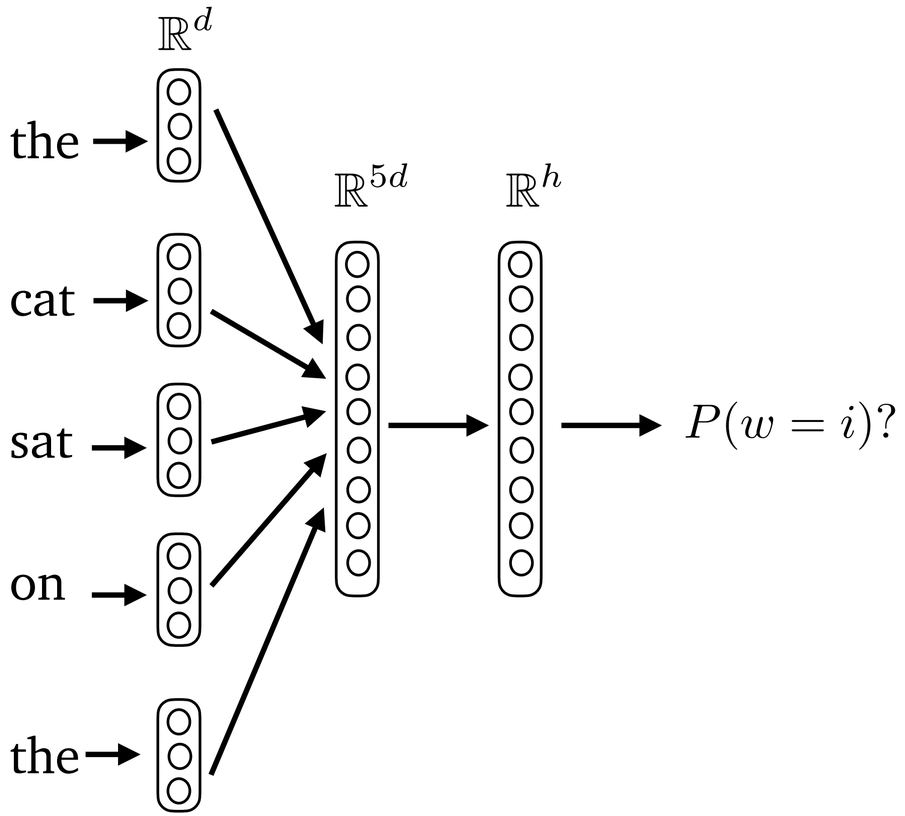
- P(mat | the cat sat on the) = ?



(Bengio et al., 2003): A Neural Probabilistic Language Model

Application: Feedforward Neural Language Models

- P(mat | the cat sat on the) = ?



- Input layer (n = 5):

$$\mathbf{x} = [\mathbf{e}_{\text{the}}; \mathbf{e}_{\text{cat}}; \mathbf{e}_{\text{sat}}; \mathbf{e}_{\text{on}}; \mathbf{e}_{\text{the}}] \in \mathbb{R}^{dn}$$

- Hidden layer

$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b}) \in \mathbb{R}^h$$

- Output layer (softmax)

$$\mathbf{z} = \mathbf{U}\mathbf{h} \in \mathbb{R}^{|V|}$$

$$P(w = i \mid \text{the cat sat on the})$$

$$= \text{softmax}_i(\mathbf{z}) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

(Bengio et al., 2003): A Neural Probabilistic Language Model

Zoom poll

What are the dimensions of W and U in this model?

d: word embedding size, h: hidden size

(a) $\mathbf{W} \in \mathbb{R}^{h \times d}, \mathbf{U} \in \mathbb{R}^{|V| \times h}$

(b) $\mathbf{W} \in \mathbb{R}^{h \times 5d}, \mathbf{U} \in \mathbb{R}^{|V| \times h}$

(c) $\mathbf{W} \in \mathbb{R}^{h \times 5d}, \mathbf{U} \in \mathbb{R}^{|V| \times d}$

(d) $\mathbf{W} \in \mathbb{R}^{h \times d}, \mathbf{U} \in \mathbb{R}^{d \times h}$

The answer is (b).

- Input layer (n = 5):

$$\mathbf{x} = [\mathbf{e}_{\text{the}}; \mathbf{e}_{\text{cat}}; \mathbf{e}_{\text{sat}}; \mathbf{e}_{\text{on}}; \mathbf{e}_{\text{the}}] \in \mathbb{R}^{dn}$$

- Hidden layer

$$\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b}) \in \mathbb{R}^{h}$$

- Output layer (softmax)

$$\mathbf{z} = \mathbf{U}\mathbf{h} \in \mathbb{R}^{|V|}$$

$$P(w = i \mid \text{context}) = softmax_i(\mathbf{z})$$