AIE1007: Natural Language Processing

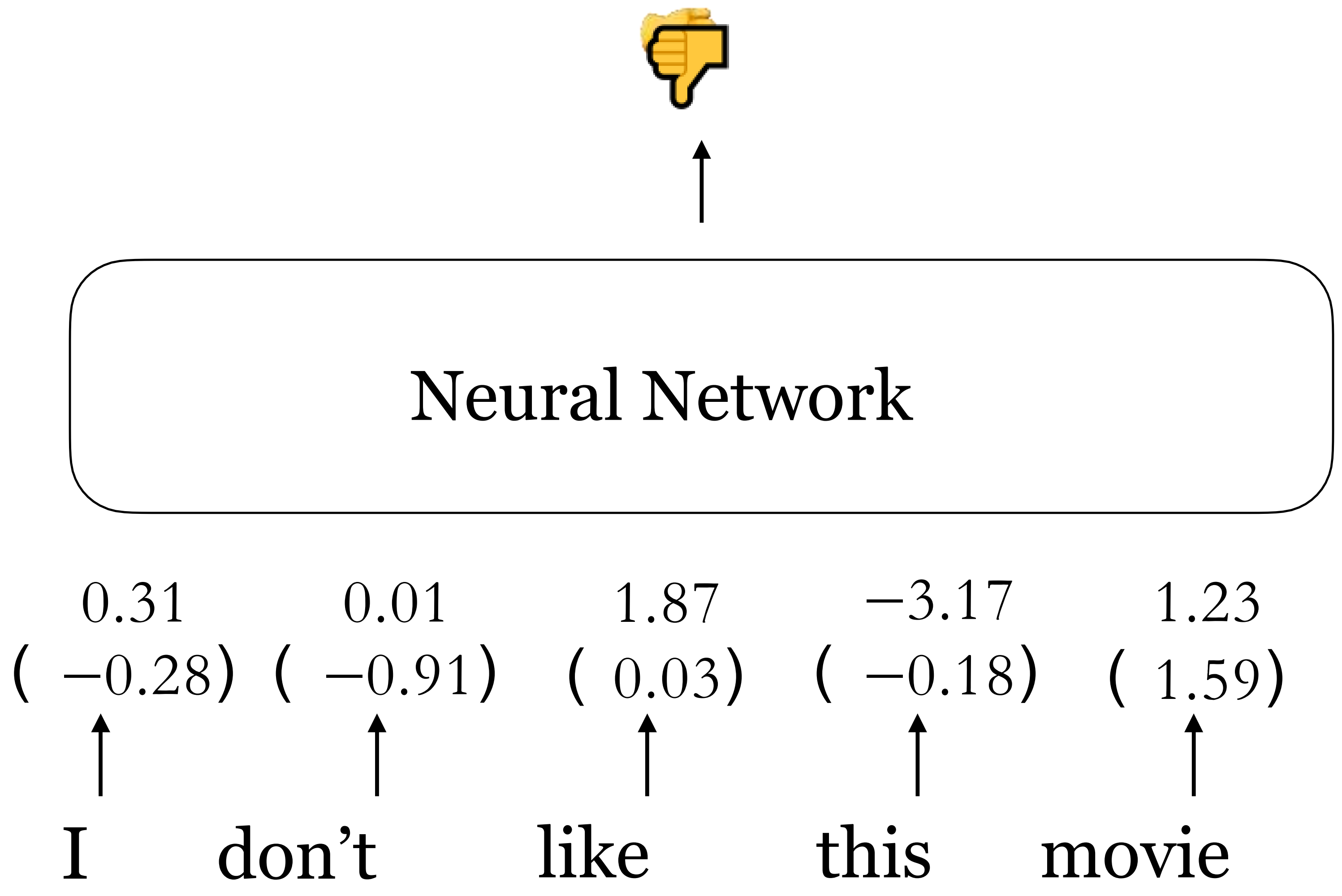
L8: Neural networks for NLP

Autumn 2024

From word embeddings to neural networks

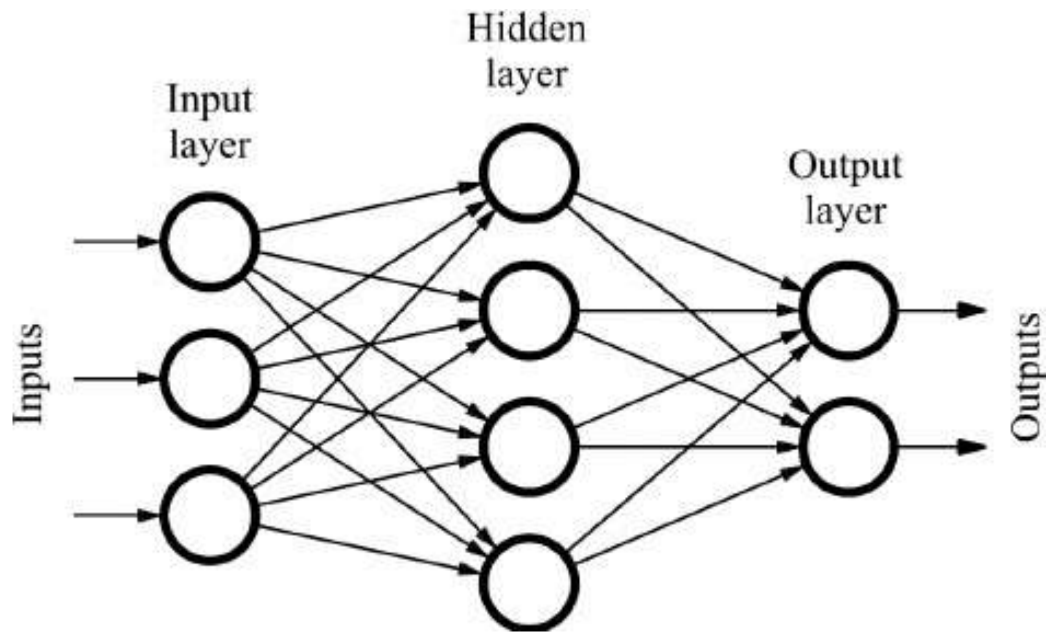
$$v_{\text{cat}} = \begin{pmatrix} -0.224 \\ 0.130 \\ -0.290 \\ 0.276 \end{pmatrix} \quad v_{\text{dog}} = \begin{pmatrix} -0.124 \\ 0.430 \\ -0.200 \\ 0.329 \end{pmatrix}$$

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

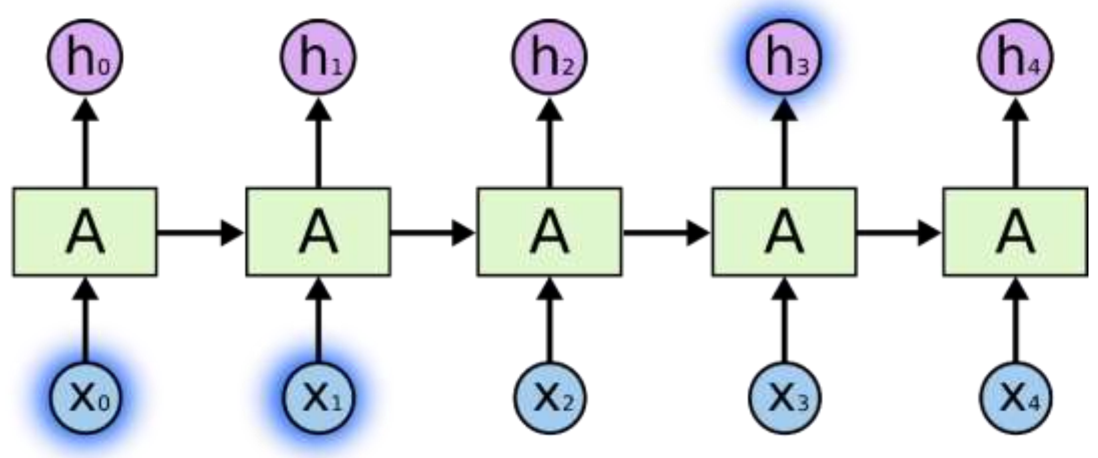


Neural networks in NLP

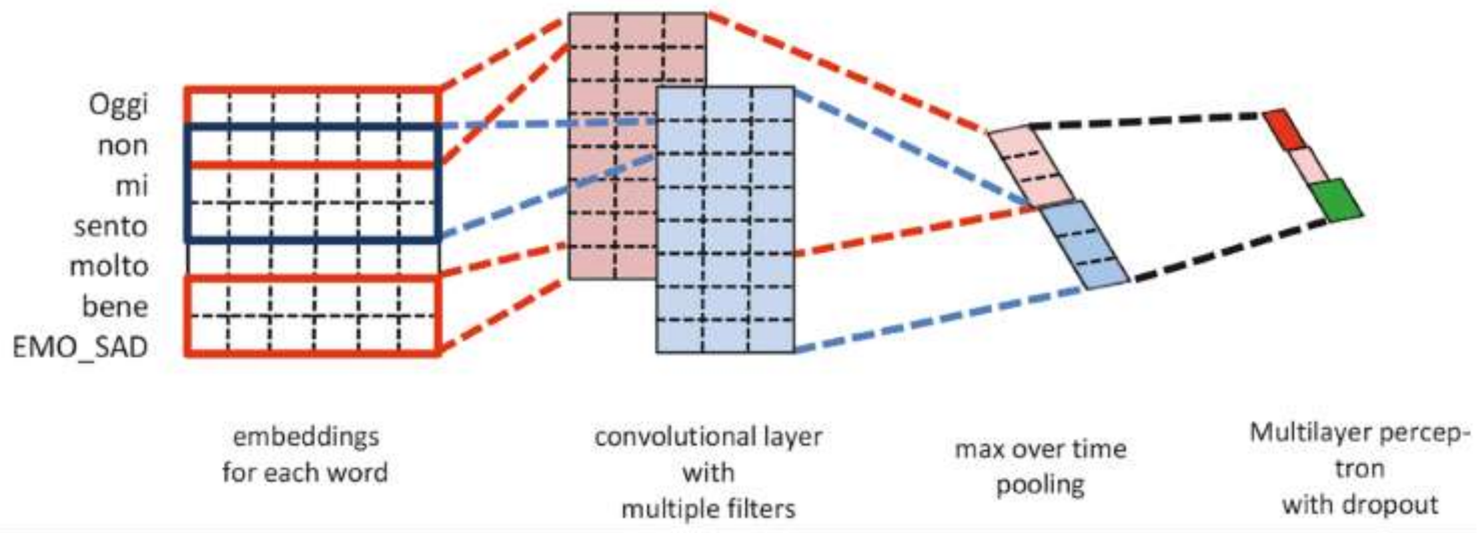
Feed-forward NNs



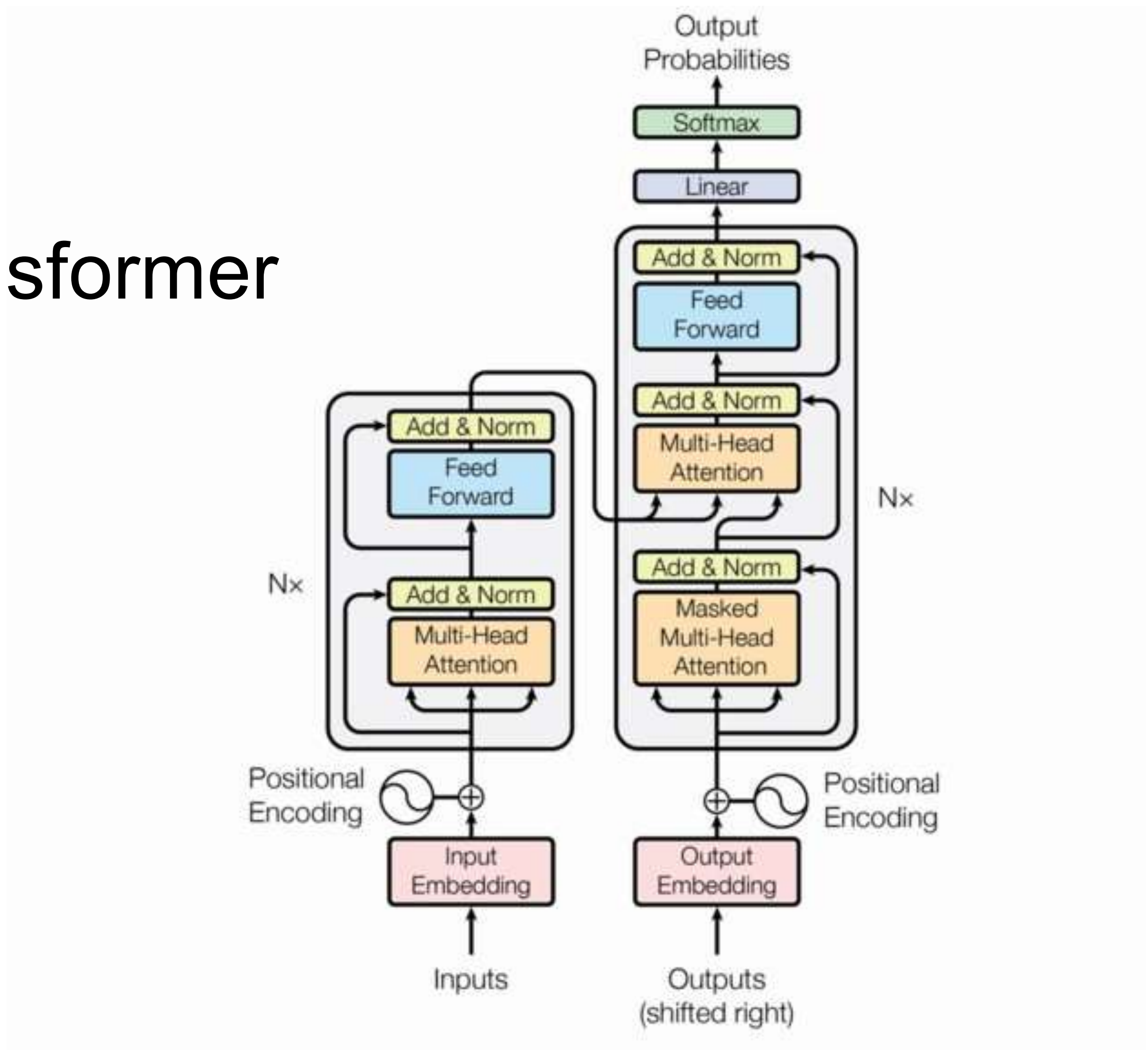
Recurrent NNs



Convolutional NNs



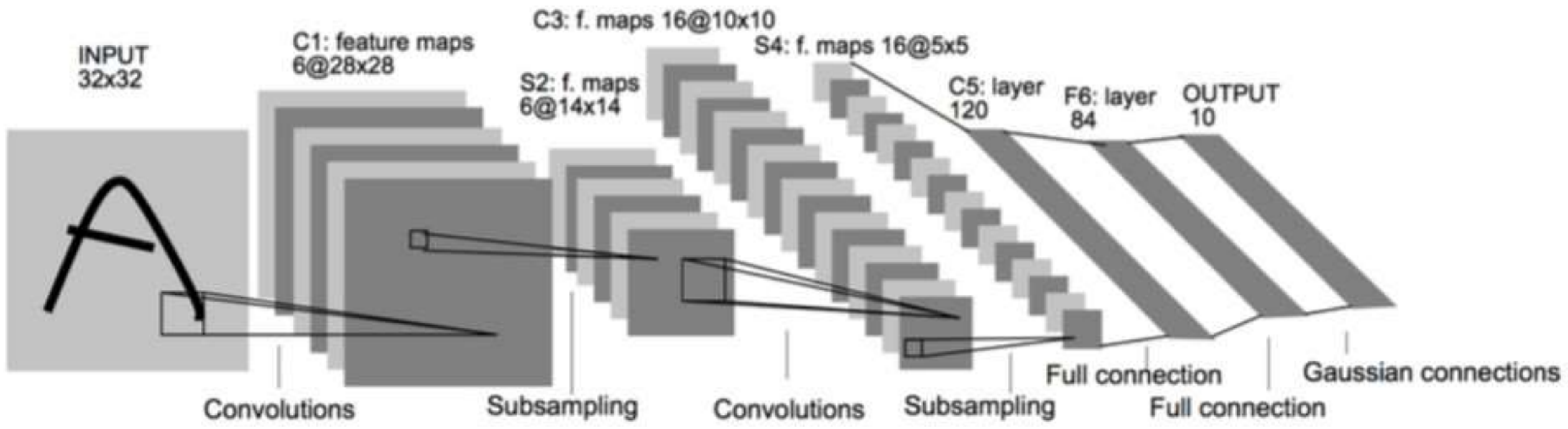
Transformer



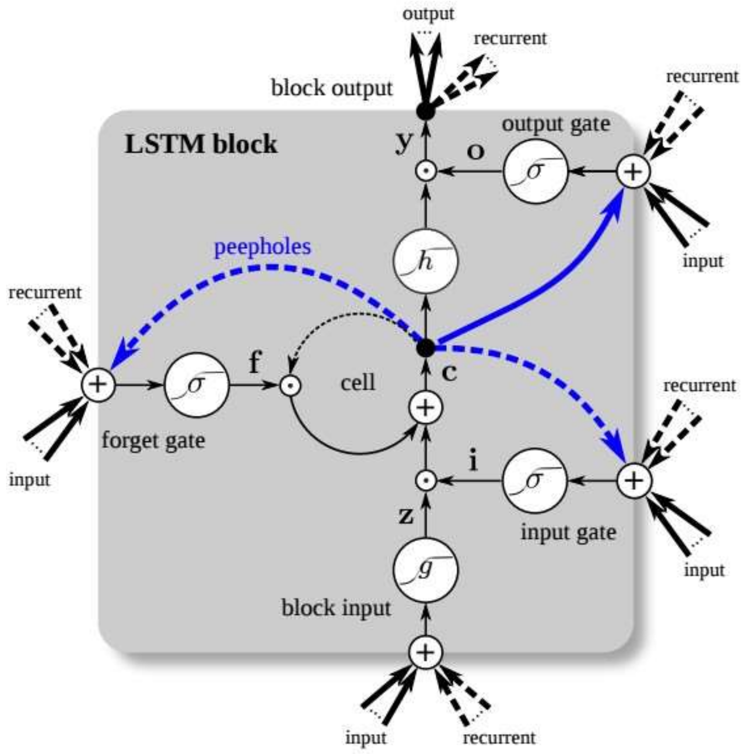
Neural networks for NLP: History

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



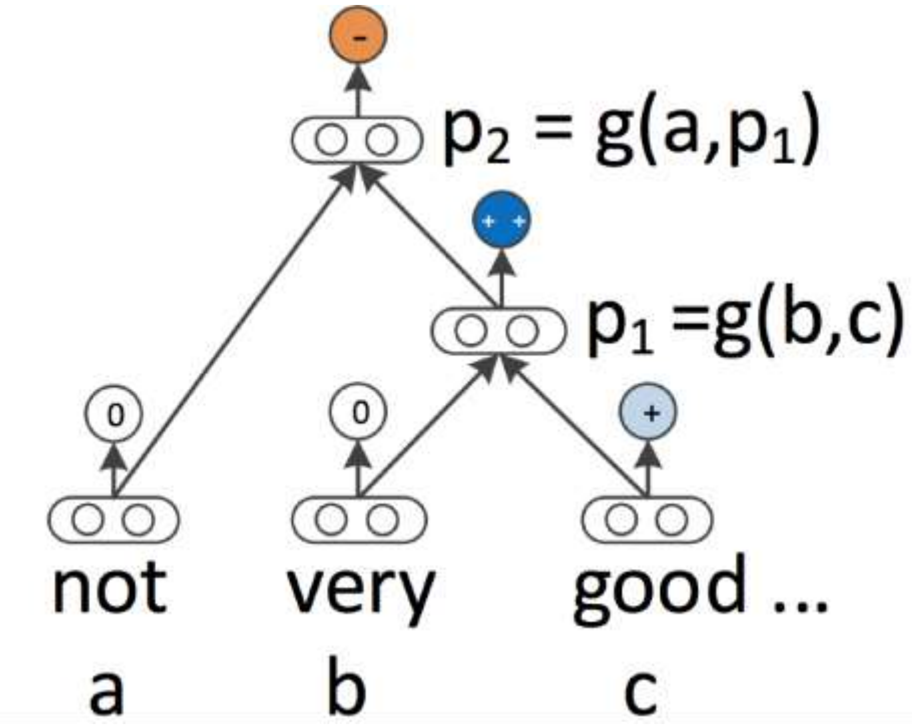
Slide credit: Greg Durrett

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, GPT, BERT)
- 2020+: the emergency of large language models (GPT-3, ChatGPT)

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

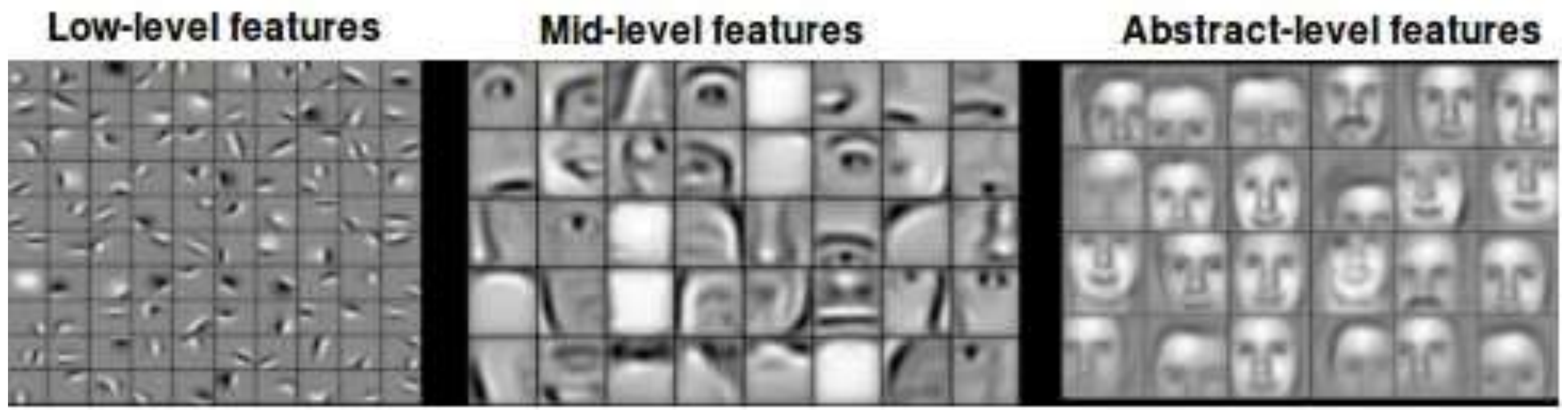
The “promise” of deep learning

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

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

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

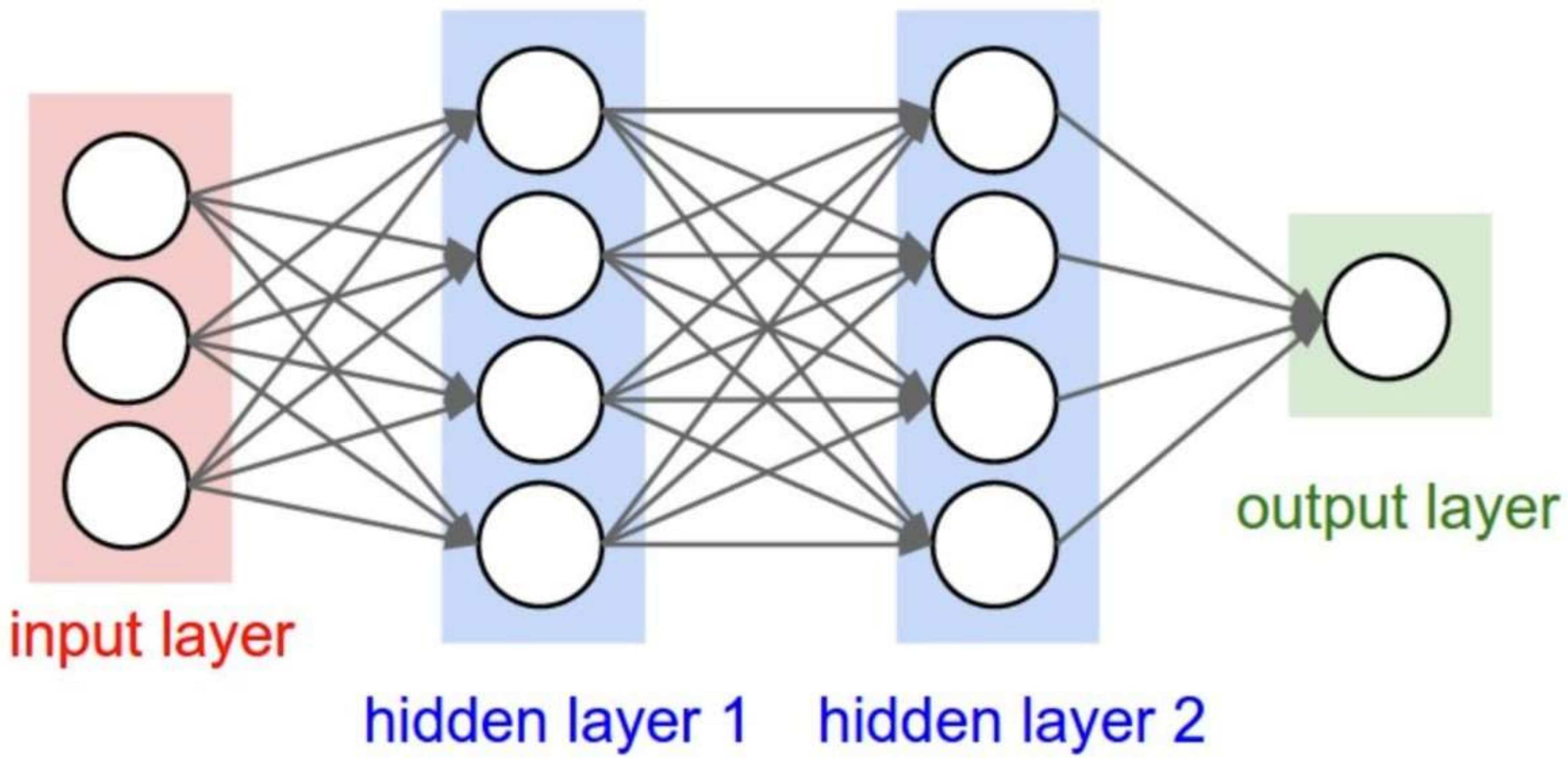
- **Deep learning** attempts to learn multiple levels of representations on increasing complexity/abstraction



Review: Feedforward neural networks

Feed-forward NNs

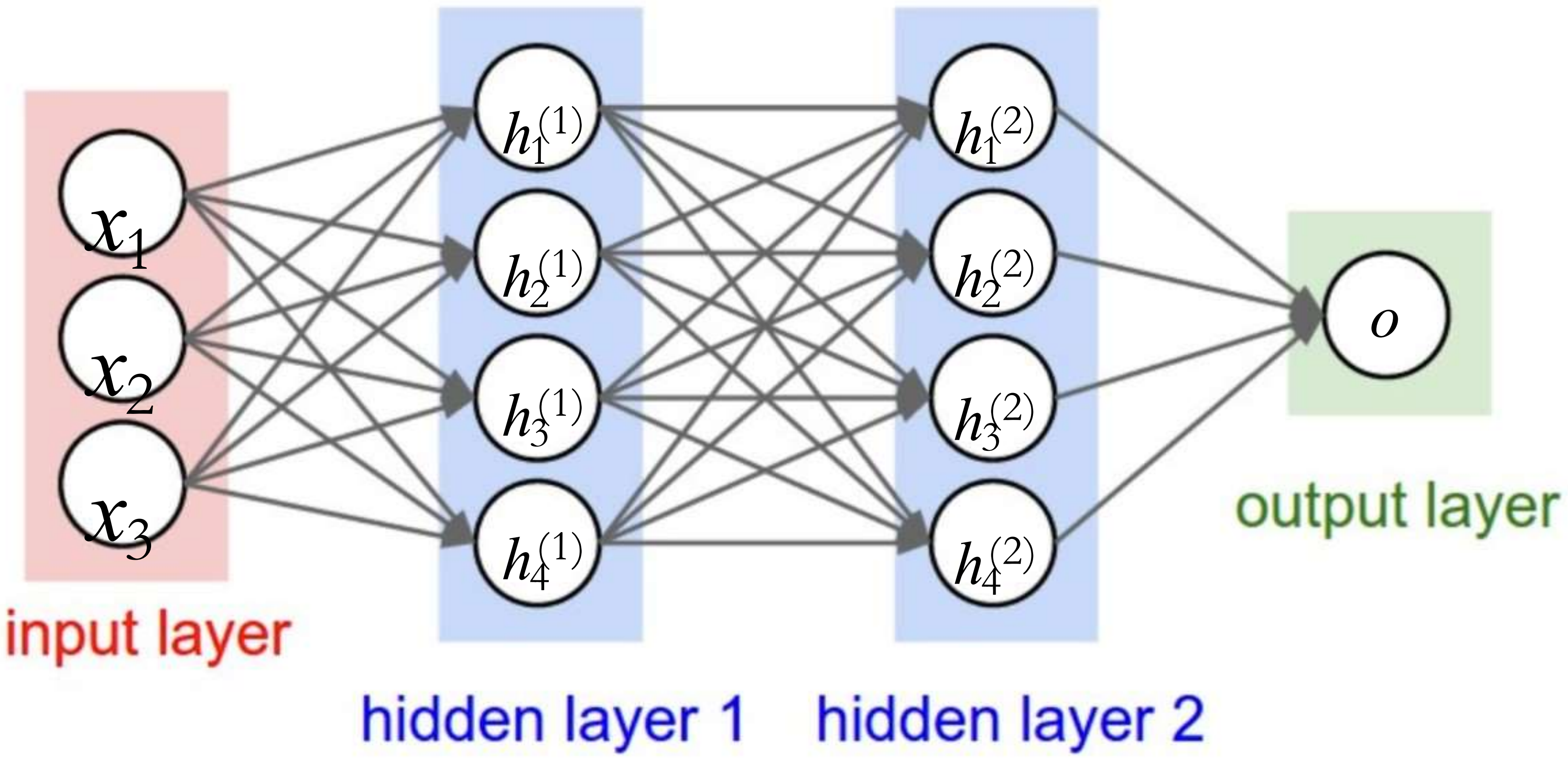
- The units are connected with no cycles
- The outputs from units in each layer are passed to units in the next higher layer
- No outputs are passed back to lower layers



Fully-connected (FC) layers:

All the units from one layer are fully connected to every unit of the next layer.

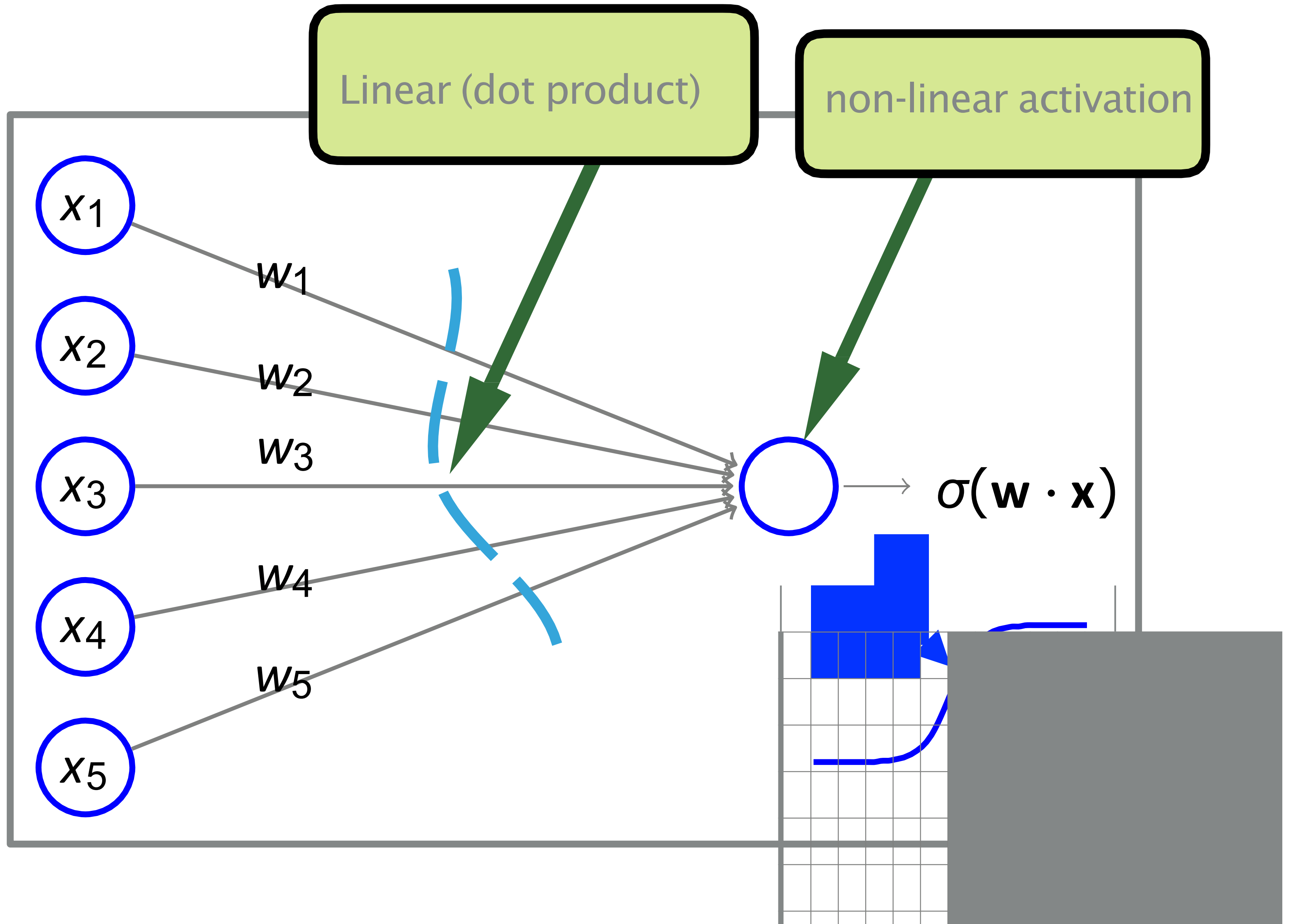
Feed-forward NNs



$$h_1^{(1)} = f(w_{1,1}^{(1)} x_1 + w_{1,2}^{(1)} x_2 + w_{1,3}^{(1)} x_3)$$

$$h_3^{(2)} = f(w_{3,1}^{(2)} h_1^{(1)} + w_{3,2}^{(2)} h_2^{(1)} + w_{3,3}^{(2)} h_3^{(1)} + w_{3,4}^{(2)} h_4^{(1)})$$

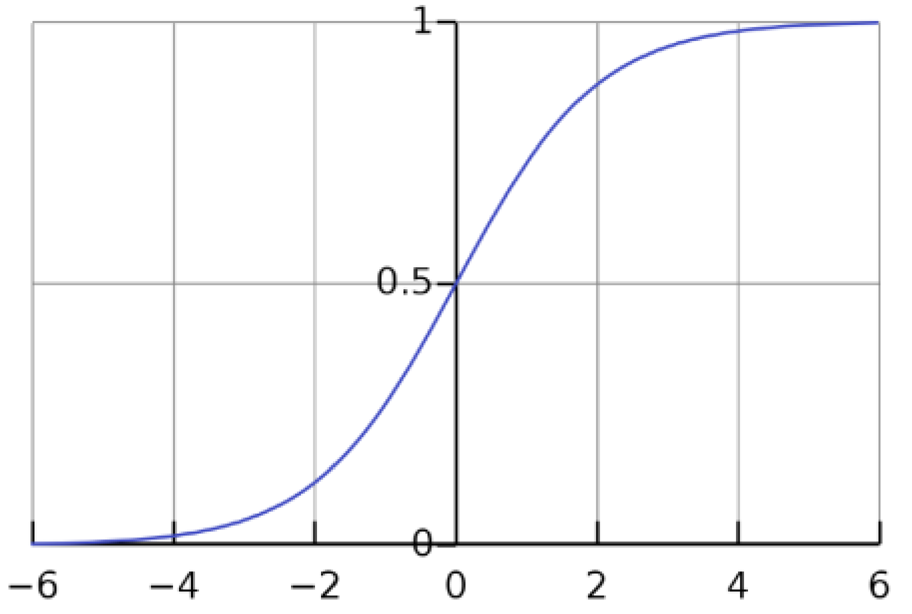
non-linearity f: , $\tanh$ *or ReLU.*



Activation functions

sigmoid

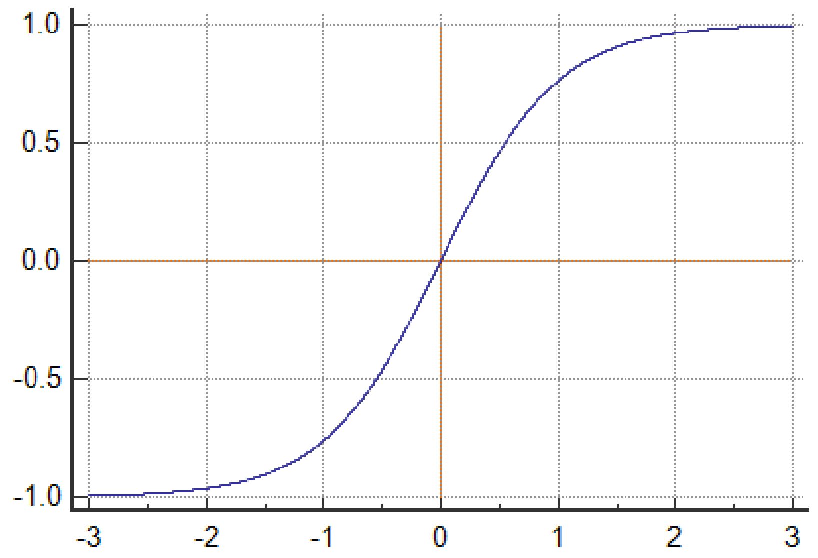
$$f(z) = \frac{1}{1 + e^{-z}}$$



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

tanh

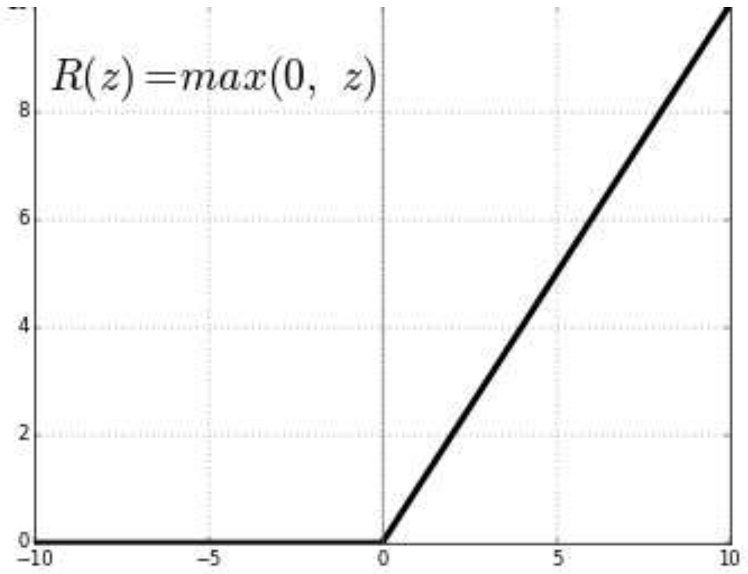
$$f(z) = \frac{e^{2z} - 1}{e^{2z} + 1}$$



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

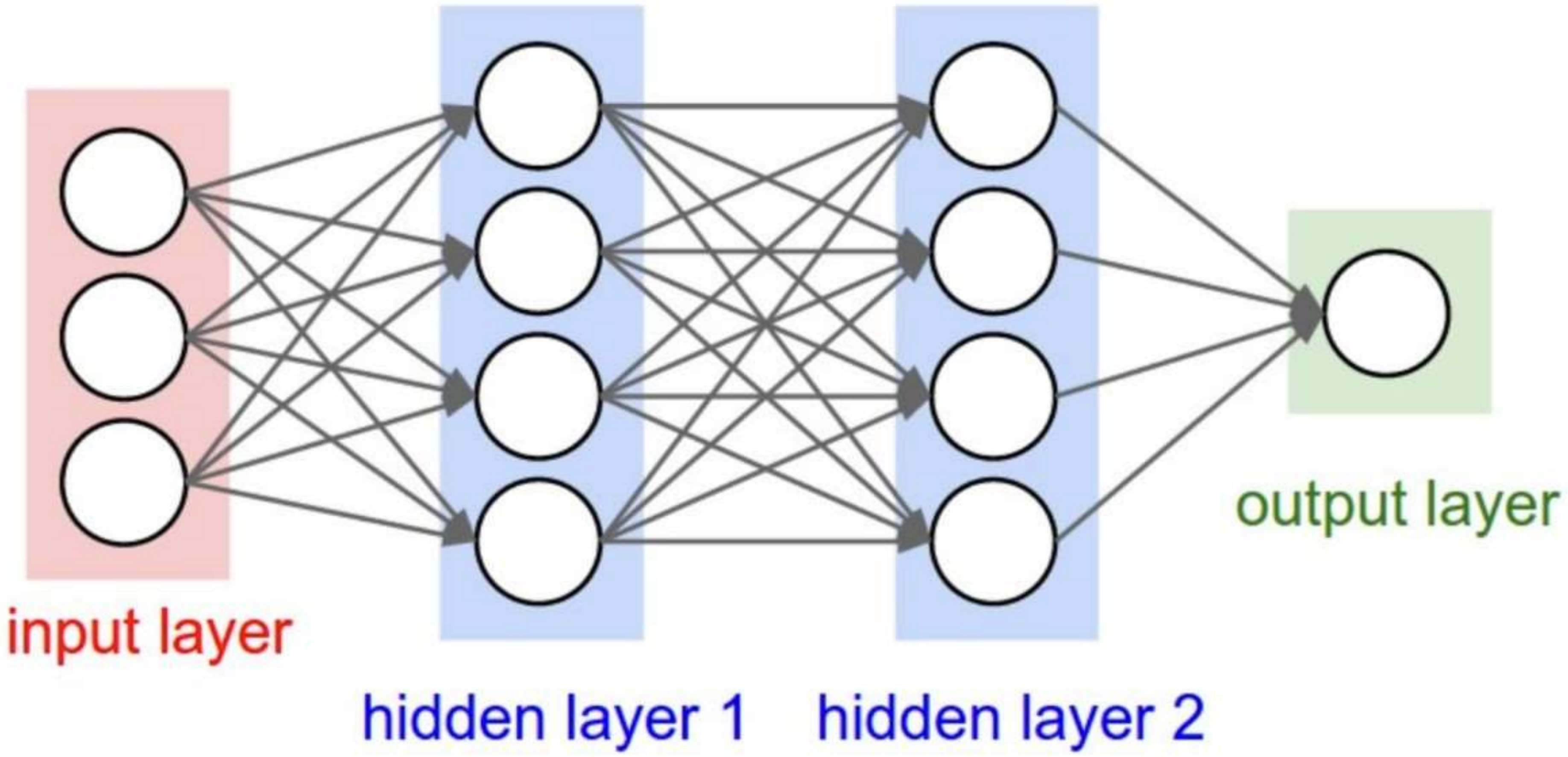
ReLU
(rectified linear unit)

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



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

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:

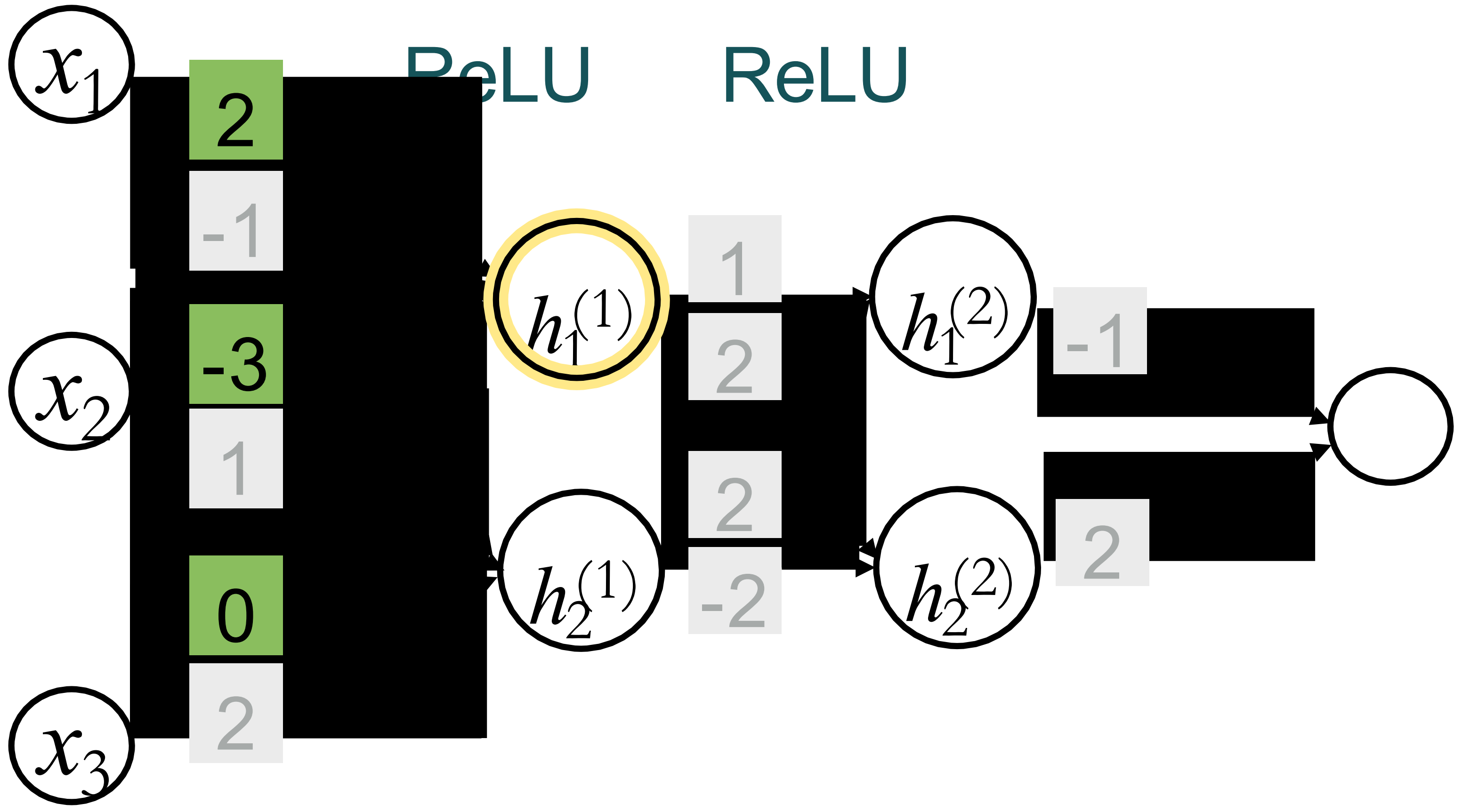
$$\mathbf{y} = \mathbf{W}^{(o)}\mathbf{h}_2, \mathbf{W}^{(o)} \in \mathbb{R}^{C \times d_2}$$

*: *f* is applied element-wise

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

C: number of classes
d: input dimension, d_1, d_2: hidden dimensions

Feedforward NNs



(Bias terms omitted in the next few slides)

For $x_1 = x_2 = x_3 = 1$, what is the value of $h_1^{(1)}$?

(a) 0 (b) -1 (c) 1 (d) 2

Correct: (a), because of the RELU:

max(2 x 1 + (-3) x 1 + 0 x 1, 0) = max(-1, 0) = 0

Feedforward NNs for multi-class classification

$$\mathbf{y} = \mathbf{W}^{(o)}\mathbf{h}_2, \mathbf{W}^{(o)}\ 2\ \mathrm{R}^{C \rightarrow d_z}$$

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{y}) \qquad \mathrm{softmax}(\mathbf{y})_k = \frac{\exp(y_k)}{\sum_{j=1}^{C} \exp(y_j)} \qquad \mathbf{y} = [y_1, y_2, \ldots, y_C]$$

Training loss:

$$\min_{\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \mathbf{W}^{(o)}} - \sum_{(\mathbf{x}, y) \in D} \log \hat{\mathbf{y}}_y$$

$$\mathbf{h}^{(1)} = \mathrm{ReLU}(\mathbf{W}^{(1)}\mathbf{x})$$

$$\mathbf{h}^{(2)} = \mathrm{ReLU}(\mathbf{W}^{(2)}\mathbf{h}^{(1)})$$

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{W}^{(o)}\mathbf{h}^{(2)})$$

Training feedforward NNs:
stochastic gradient descent!

Neural networks are difficult to optimize.
SGD can only converge to local minimum.
Initializations and optimizers matter a lot!

Back-propagation

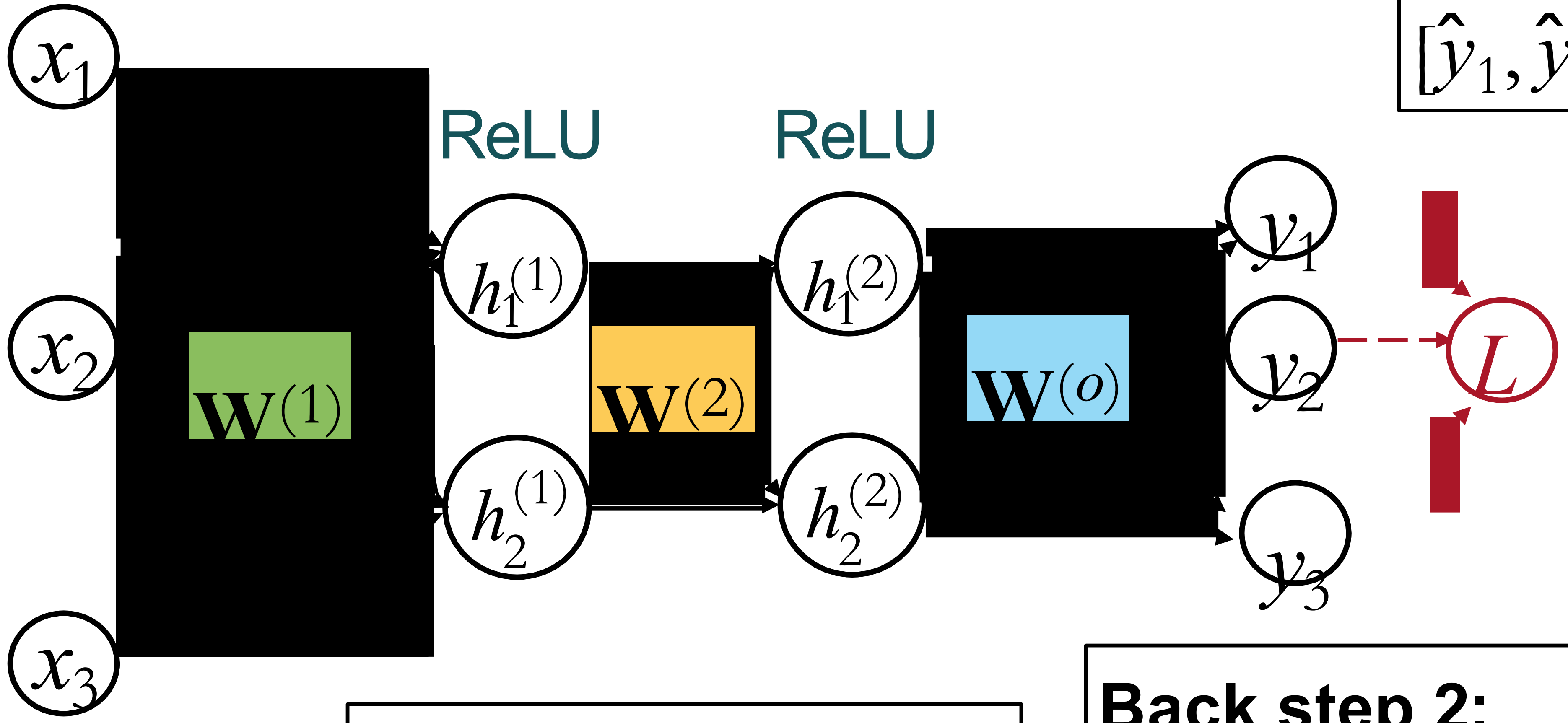
Forward propagation:
from input to output layer

Given: x_1, x_2, x_3
and the class label y
(a single training example)

Forward step 1:
Compute $h_1^{(1)}, h_2^{(1)}$

Forward step 2:
Compute $h_1^{(2)}, h_2^{(2)}$

Forward step 3:
Compute y_1, y_2, y_3 and
$[\hat{y}_1, \hat{y}_2, \hat{y}_3] = \mathrm{softmax}[y_1, y_2, y_3]$



Forward step 4:
Compute loss
$L = -\log \hat{\mathbf{y}}_y$

Goal:
$\frac{\partial L}{\partial W^{(1)}}, \frac{\partial L}{\partial W^{(2)}}, \frac{\partial L}{\partial W^{(o)}}$

Back propagation:
from output to input layer

Back step 1:
Compute $\frac{\partial L}{\partial y_1}, \frac{\partial L}{\partial y_2}, \frac{\partial L}{\partial y_3}$

Back step 2:
Compute $\frac{\partial L}{\partial h_1^{(2)}}, \frac{\partial L}{\partial h_2^{(2)}}, \frac{\partial L}{\partial W^{(o)}}$

Back step 3:
Compute $\frac{\partial L}{\partial h_1^{(1)}}, \frac{\partial L}{\partial h_2^{(1)}}, \frac{\partial L}{\partial W^{(2)}}$

Back step 4:
Compute $\frac{\partial L}{\partial W^{(1)}}$

Back-propagation in PyTorch

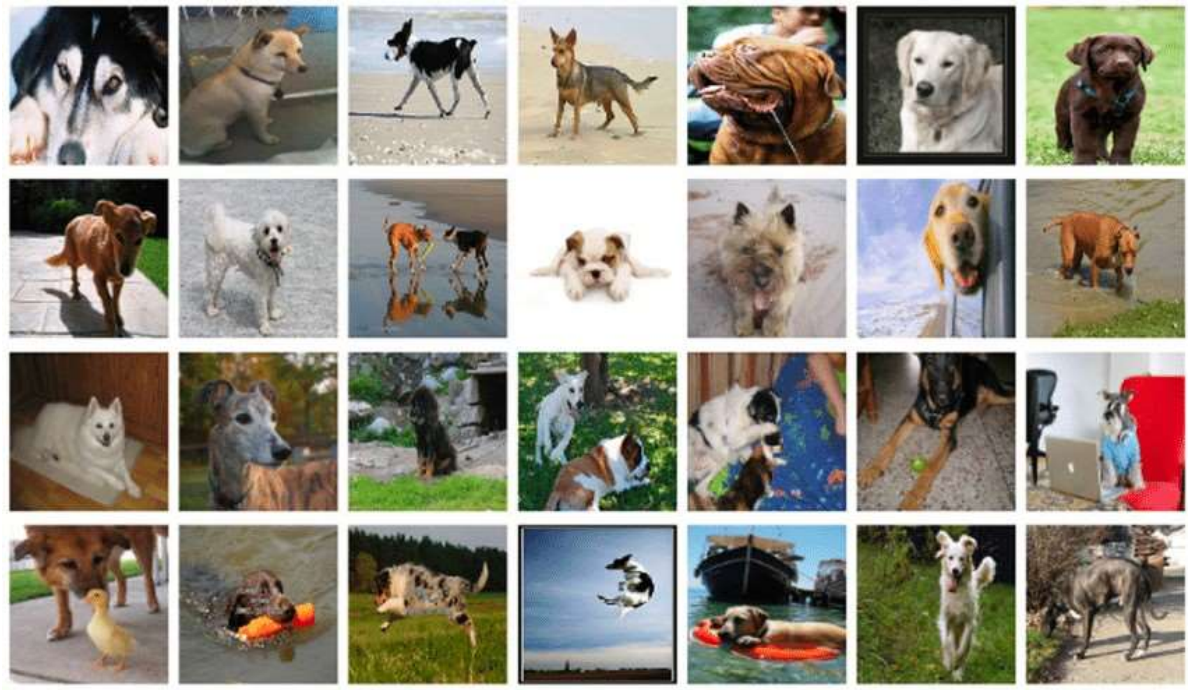
```python
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 10)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

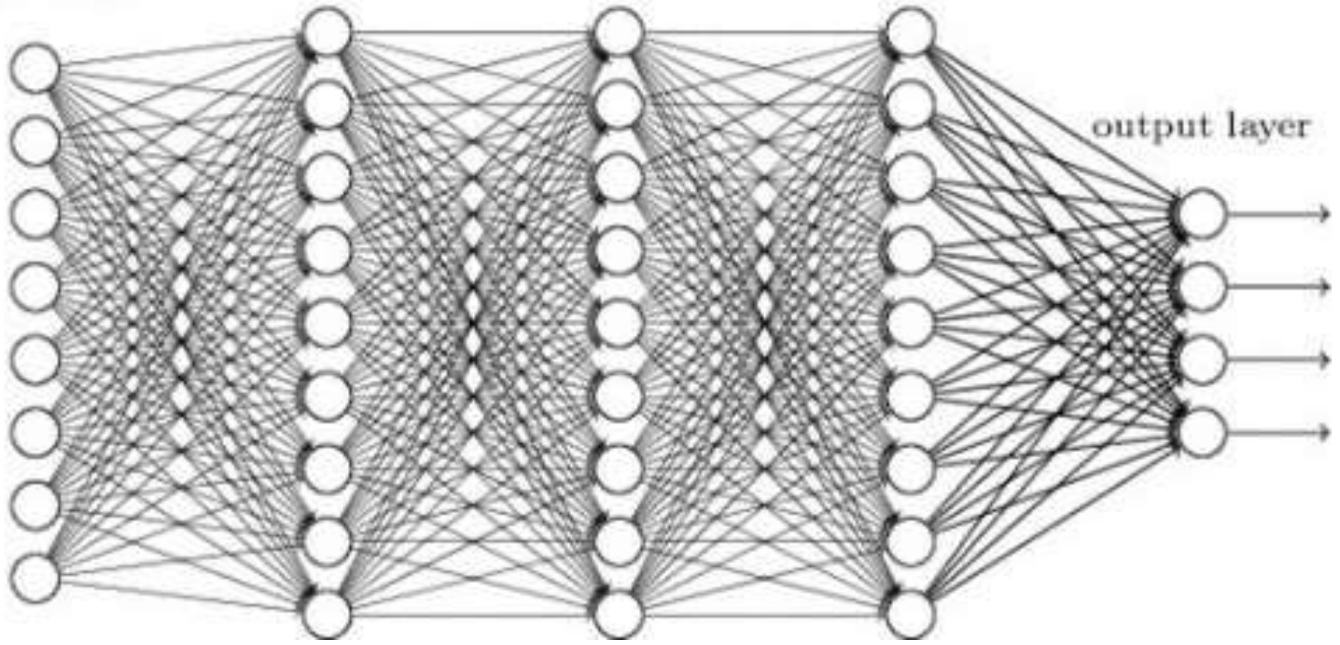
```python
import torch.optim as optim

net = Net()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

```python
outputs = net(inputs)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()
```

PyTorch did back-propagation for you in this one line of code!

Comparison: image vs text inputs





label = “dog”

a sometimes tedious film
i had to look away - this was god awful .
a gorgeous , witty , seductive movie .

label = positive

- Images: fixed-size input, continuous values
- Text: **variable-length** input, discrete words
  - need to convert into vectors - word embeddings!

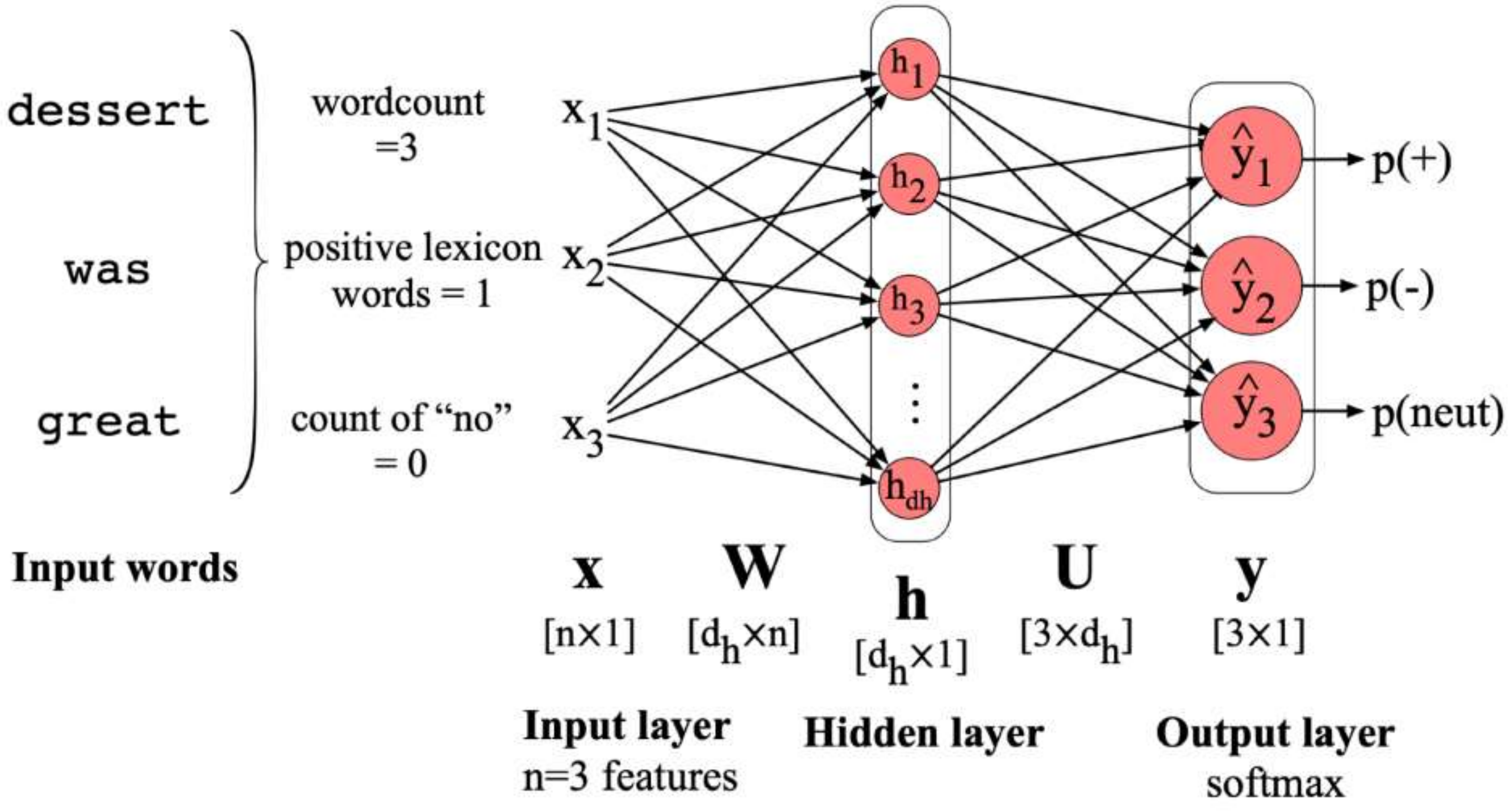
Neural “bag-of-words” models for text classification

Neural networks for text classification

- Input: $w_1, w_2, \ldots, w_K \in V$
- Output: $y \in C$

- Input: dessert was great
- Output: positive C = {positive, negative, neutral}

Solution #1: You can construct a feature vector from the input and simply feed the vector to a **neural network**, instead of a **logistic regression classifier**!



(each $\mathbf{x}_i$ is a hand-designed feature)

- $\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n]$
- $\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$
- $\mathbf{y} = \mathbf{U}\mathbf{h}$
- $\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{y})$

Deep learning has the promise to learn good features automatically..

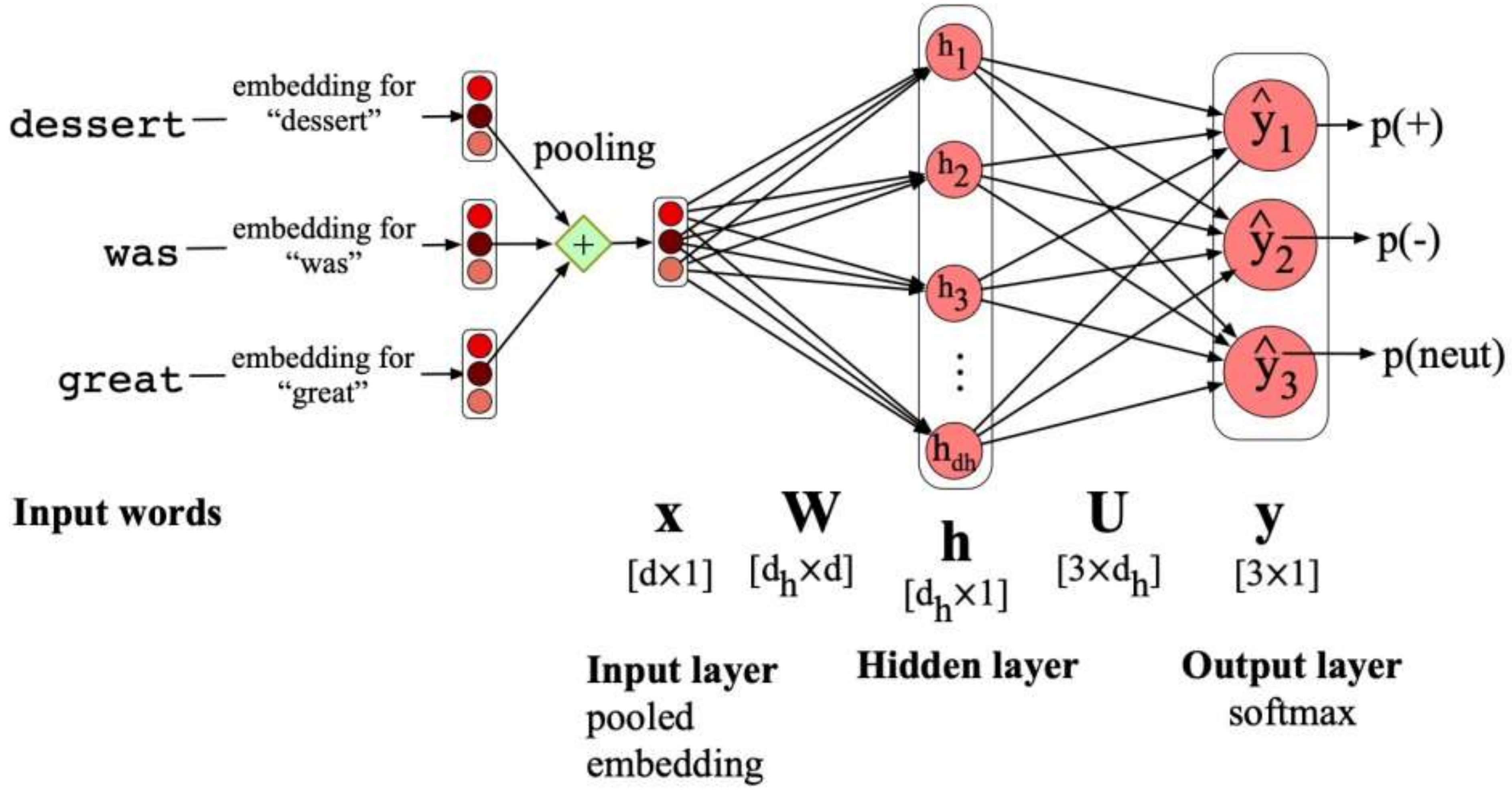
Neural networks for text classification

- How can we feed a **variable-length** input to a neural network classifier? $w_1, w_2, \ldots, w_K \in V$

Solution #2: Let's take the all the word embeddings of these words and aggregate them into a vector through some **pooling** function!

$$\mathbf{x}_{\text{mean}} = \frac{1}{K}\sum_{i=1}^{K} e(w_i)$$

pooling: sum, mean or max

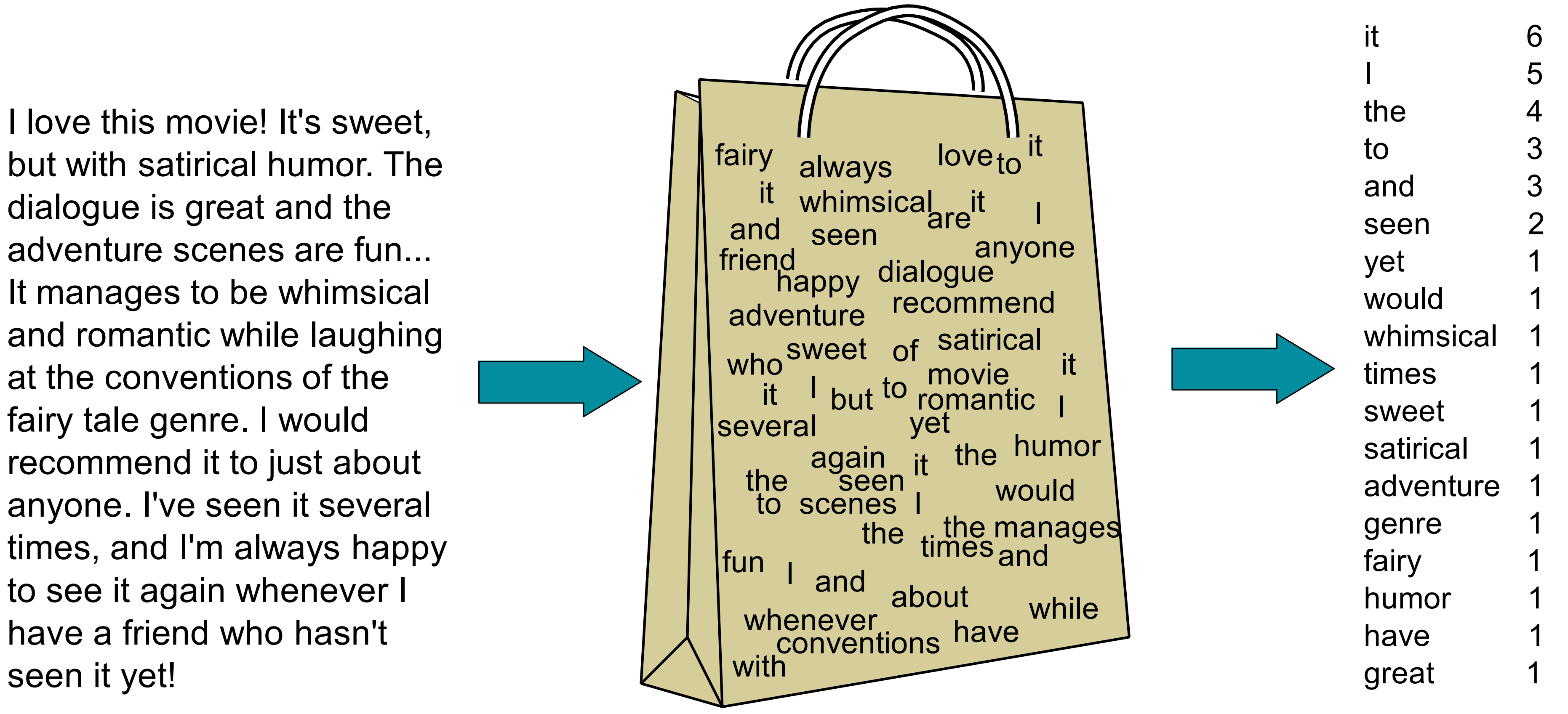


- $\mathbf{x} = \frac{1}{K}\sum_{i=1}^{K} e(w_i)$
- $\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$
- $\mathbf{y} = \mathbf{U}\mathbf{h}$
- $\hat{\mathbf{y}} = \text{softmax}(\mathbf{y})$

Neural networks for text classification

- (+): This provides a simple and flexible way to handle variable-length input
- (+): It learns feature representations automatically from the data
- (+): It can generalize to similar inputs through word embeddings
- (-): The model throws away any sequential information of the text

neural bag-of-words model (NBOW)



How to train this model?

- Training data: $\{(d^{(1)}, y^{(1)}), \ldots, (d^{(m)}, y^{(m)})\}$
- Parameters: $\{\mathbf{W}, \mathbf{b}, \mathbf{U}\}$
- Optimize these parameters using gradient descent!

- Word embeddings can be treated as parameters too!

$$\mathbf{E} \in \mathbb{R}^{|V|\times d}$$

- $\mathbf{x} = \frac{1}{K}\sum_{i=1}^{K} e(w_i)$
- $\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b})$
- $\mathbf{y} = \mathbf{U}\mathbf{h}$
- $\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{y})$

How to train this model?

- Common practice: initialize $\mathbf{E}$ using word embeddings (e.g. word2vec), and optimize them using SGD!

Why? $\mathbf{v}(\text{good}) \approx \mathbf{v}(\text{bad})$

	Most Similar Words for	
	Static	Non-static
bad	*good* *terrible* *horrible* *lousy*	*terrible* *horrible* *lousy* *stupid*
good	*great* *bad* *terrific* *decent*	*nice* *decent* *solid* *terrific*

(Kim 2014)

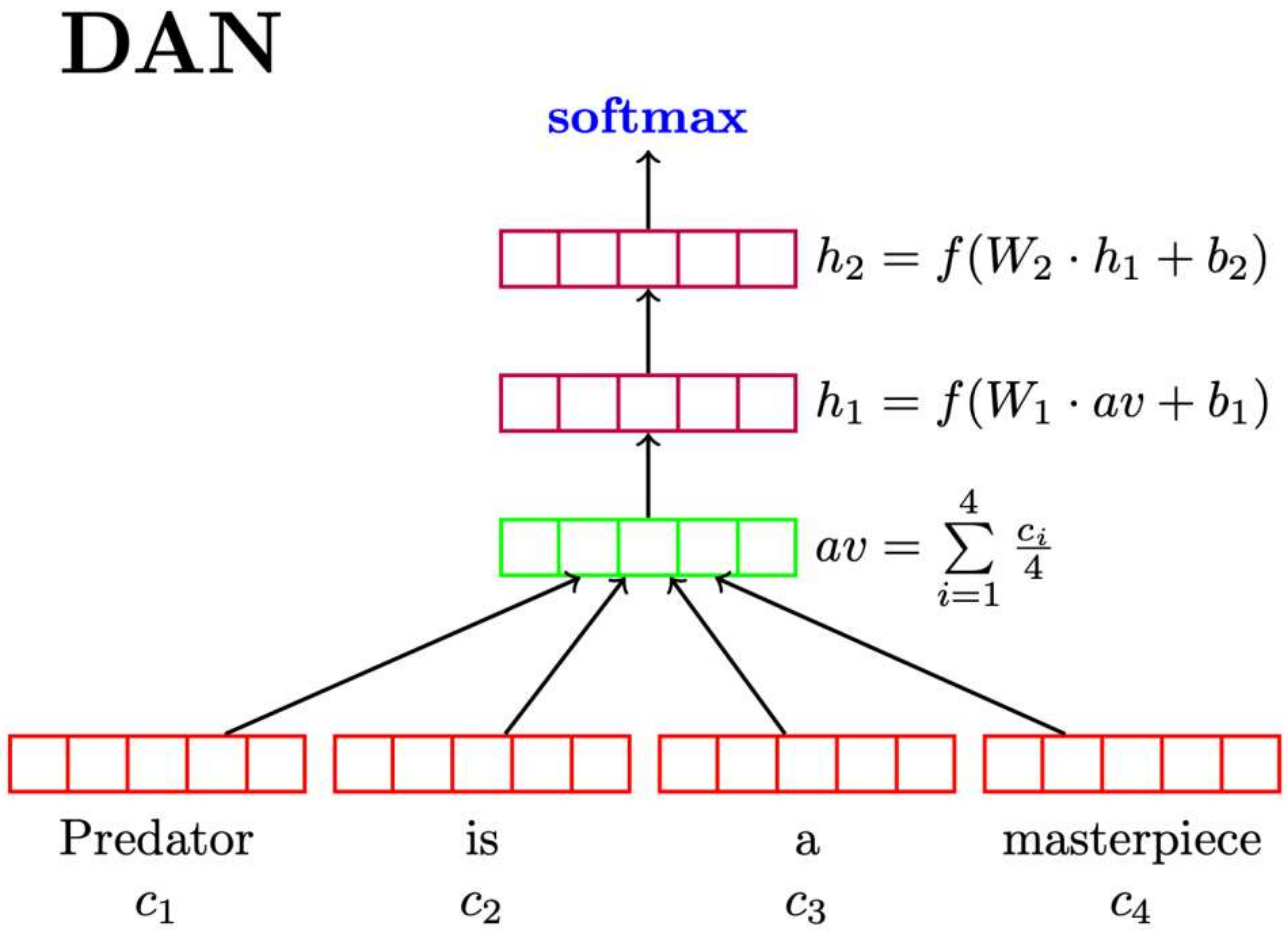
- When the training data is small, don't treat $\mathbf{E}$ as parameters!

- When the training data is very large (e.g., language modeling), initialization doesn't matter much either (= can use random initialization)

Deep Averaging Networks (DAN)

(Iyyer et al., 2015)

Deep Unordered Composition Rivals Syntactic Methods for Text Classification



Basically the same as NBOW but neural network is deeper!

f: non-linearity

Feedforward neural language models

N-gram vs neural language models

Language models: Given $x_1, x_2, \ldots, x_n \in V$, the goal is to model:

$$P(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid x_1, \ldots, x_{i-1})$$

Bigram: $P(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid x_{i-1})$

Trigram: $P(x_1, x_2, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid x_{i-2}, x_{i-1})$

Maximum likelihood estimate:

$$P(\text{sat}|\text{the cat}) = \frac{\text{count(the cat sat)}}{\text{count(the cat)}}$$

Limitations? Can't handle long histories!

As the proctor started the clock, the students opened their ______

The **keys** to the cabinet <u>is/are</u>

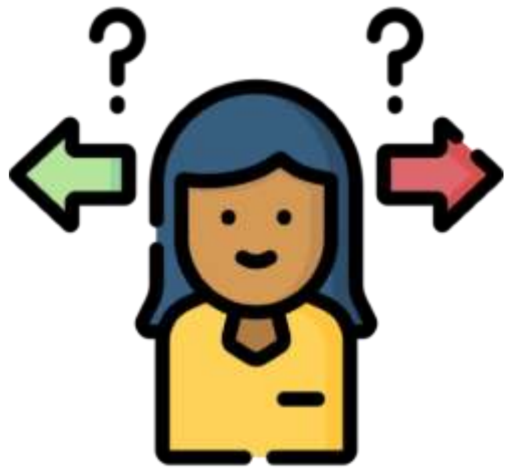
N-gram vs neural language models

- If we use a 4-gram, 5-gram, 6-gram language model, it will become too sparse to estimate the probabilities:

$$P(w \mid \text{students opened their}) = \frac{\text{count(students opened their } w)}{\text{count(students opened their)}}$$

Dilemma:

- We need to model bigger context!
- The # of probabilities that we need to estimate grow exponentially with window size!



- A lot of contexts are similar and simply counting them won't generalize

I am a **good** ______

I am a **great** ______

count(I am a good *w*)

count(I am a great *w*)

e(good) **e(great)**

Can we estimate the probabilities better?

Feedforward neural language models

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

Yoshua Bengio BENGIOY@IRO.UMONTREAL.CA
Réjean Ducharme DUCHARME@IRO.UMONTREAL.CA
Pascal Vincent VINCENTP@IRO.UMONTREAL.CA
Christian Jauvin JAUVINC@IRO.UMONTREAL.CA

Yoshua Bengio

Probabilistic models of sequences: In the 1990s, Bengio combined neural networks with probabilistic models of sequences, such as hidden Markov models. These ideas were incorporated into a system used by AT&T/NCR for reading handwritten checks, were considered a pinnacle of neural network research in the 1990s, and modern deep learning speech recognition systems are extending these concepts.

High-dimensional word embeddings and attention: In 2000, Bengio authored the landmark paper, "A Neural Probabilistic Language Model," that introduced high-dimension word embeddings as a representation of word meaning. Bengio's insights had a huge and lasting impact on natural language processing tasks including language translation, question answering, and visual question answering. His group also introduced a form of attention mechanism which led to breakthroughs in machine translation and form a key component of sequential processing with deep learning.

Generative adversarial networks: Since 2010, Bengio's papers on generative deep learning, in particular the Generative Adversarial Networks (GANs) developed with Ian Goodfellow, have spawned a revolution in computer vision and computer graphics. In one fascinating application of this work, computers can actually create original images, reminiscent of the creativity that is considered a hallmark of human intelligence.

https://awards.acm.org/about/2018-turing

Feedforward neural language models

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

Yoshua Bengio BENGIOY@IRO.UMONTREAL.CA
Réjean Ducharme DUCHARME@IRO.UMONTREAL.CA
Pascal Vincent VINCENTP@IRO.UMONTREAL.CA
Christian Jauvin JAUVINC@IRO.UMONTREAL.CA

Key idea: Instead of estimating raw probabilities, let's use a **neural network** to fit the **probabilistic distribution of language**!

$$P(w \mid \text{I am a good})$$

$$P(w \mid \text{I am a great})$$

Key ingredient: word embeddings **e(good)** **e(great)**

Hope: this would give us similar distributions for similar contexts!

Feedforward neural language models

- Feedforward neural language models approximate the probability based on the previous *m* (e.g., 5) words - *m* is a hyper-parameter!

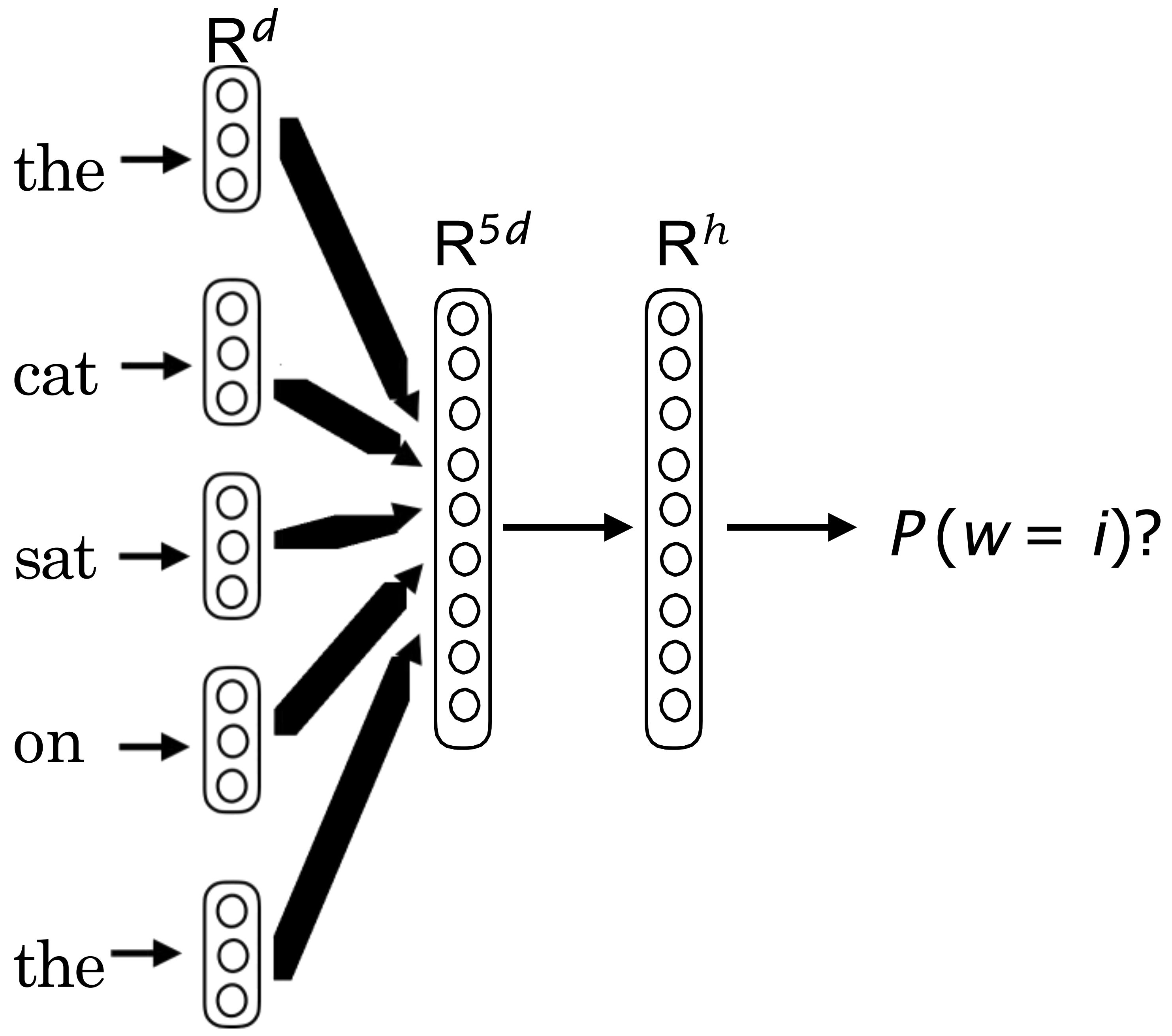
$$P(x_1, x_2, \ldots, x_n) \approx \prod_{i=1}^{n} P(x_i \mid x_{i-m+1}, \ldots, x_{i-1})$$

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

d: word embedding size

h: hidden size

It is a |V|-way classification problem!



Feedforward neural language models

P(mat | the cat sat on the) = ? d: word embedding size h: hidden size

- Input layer (m= 5):

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

- Hidden layer:

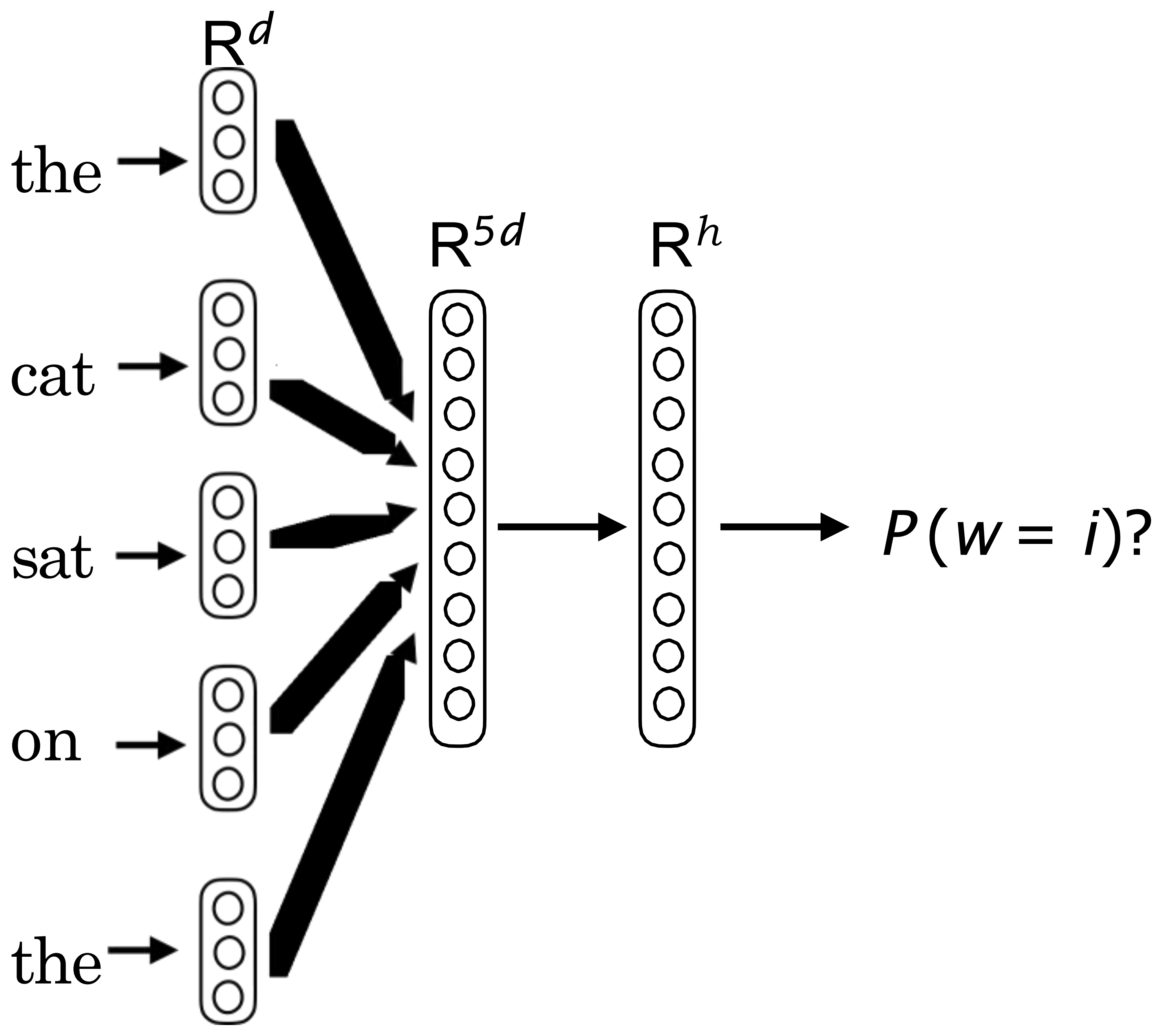
  $\mathbf{h} = \tanh(\mathbf{W}\mathbf{x} + \mathbf{b}) \in \mathbb{R}^{h}$

- Output layer

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

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

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



What are the dimensions of W and U?

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

Correct: (b)

- Input layer (m= 5):

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

- Hidden layer:

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

- Output layer

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

$$P(w = i \mid \text{the cat sat on the}) = \text{softmax}_i(\mathbf{z}) = \frac{e^{z_i}}{\sum_k e^{z_k}}$$

Feedforward neural language models

- How to train this model? A: Use a lot of raw text to create training examples and run gradient-descent optimization!

The Fat Cat Sat on the Mat is a 1996 children's book by Nurit Karlin. Published by Harper Collins as part of the reading readiness program, the book stresses the ability to read words of specific structure, such as -at.

→

the fat cat sat on	the
fat cat sat on the	mat
cat sat on the mat	is
sat on the mat is	a
…	

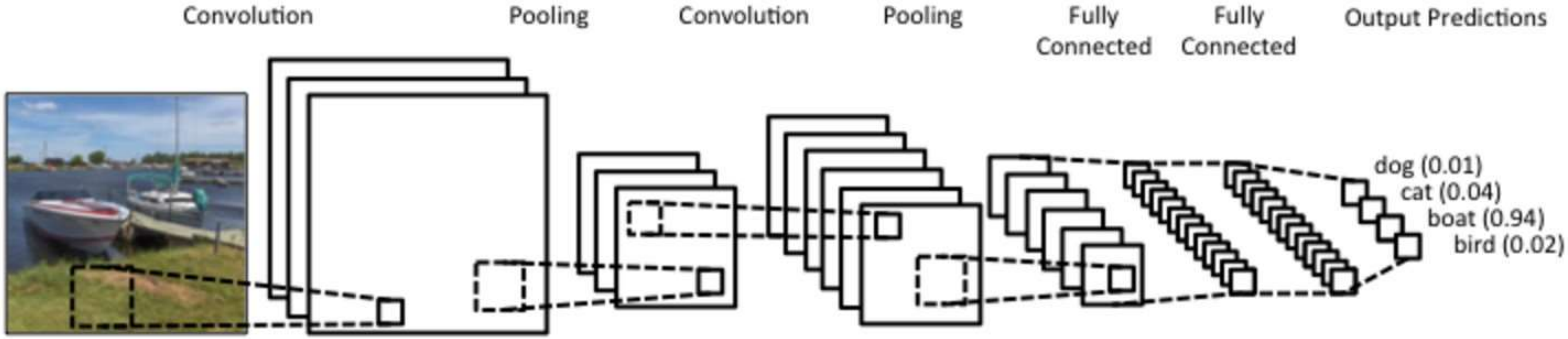
- Limitations?
  - $\mathbf{W}$ **linearly** scales with the context size m
  - The models learns separate patterns for different positions!
- Better solutions: recurrent NNs, Transformers..

the fat cat sat on	the
fat cat sat on the	mat
cat sat on the mat	is

"sat on" corresponds to different parameters in $\mathbf{W}$

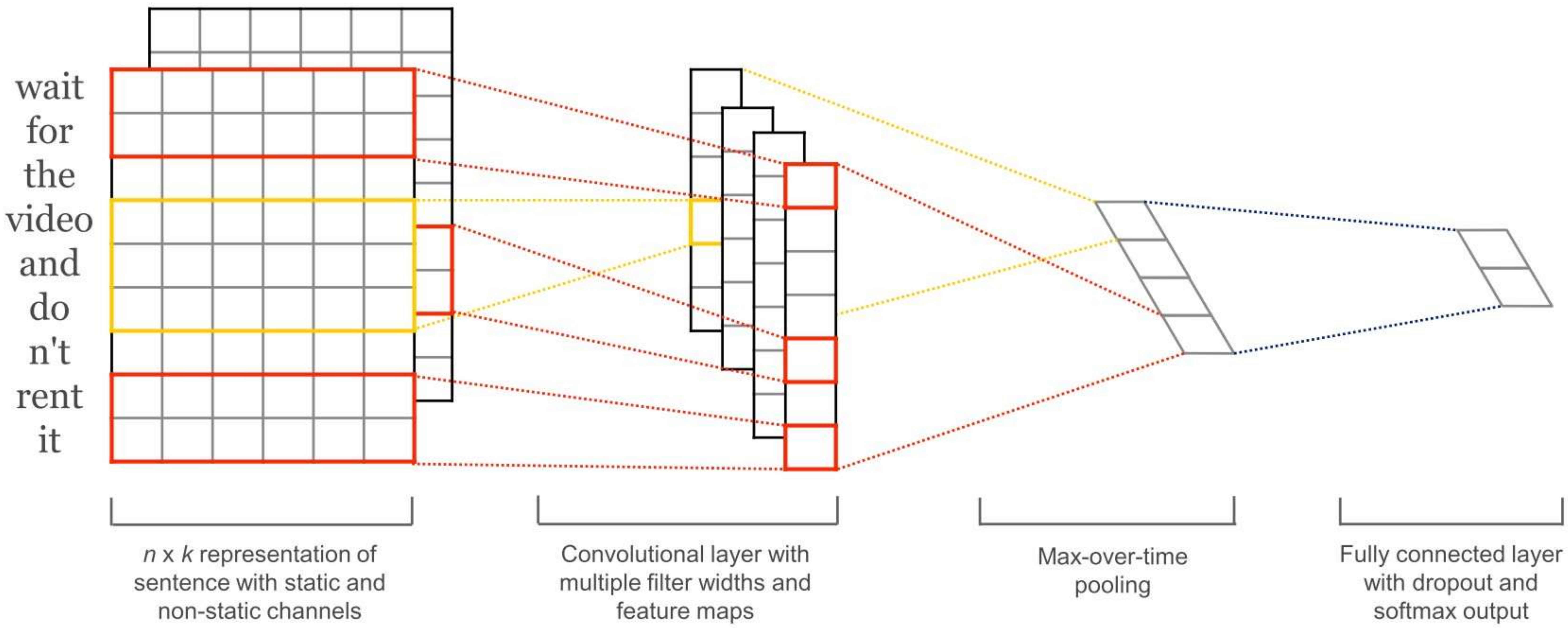
Convolutional NNs for text classification

Convolutional NNs in image classification



Key components: 1) convolution; 2) pooling; 3) multiple channels (feature maps)

Convolutional NNs for text classification



(Kim 2014): Convolutional Neural Networks for Sentence Classification