TTIC 31190: Natural Language Processing

Lecture 6: Neural Networks

Fall 2023

Announcement

- Final exam schedule out: Tuesday December 5, 3-5pm
- Pass/fail option available for this course

- Reminder: Assignment 1 due this Thursday

Schedule

Date	Topic	Instructor
W, 9/27	Introduction	Freda
M, 10/2	Word	Joe
W, 10/4	Distributional Semantics	Joe
M, 10/9	Dataset & Classification	Freda
W, 10/11	Classification	Freda
M, 10/16	Neural Networks	Freda
W, 10/18	Neural Networks & Sequence Labeling	Freda
M, 10/23	Sequence Labeling	Freda
W, 10/25	Language Modeling	Joe
M, 10/30	Seq2Seq	Freda
W, 11/1	Seq2Seq & Syntax	Freda

Date	Topic	Instructor
M, 11/6	Syntax	Freda
W, 11/8	Semantics	Joe
M, 11/13	Semantics	Joe
W, 11/15	Pragmatics	Freda
M, 11/20	Thanksgiving Break	
W, 11/22	Thanksgiving Break	
M, 11/27	LLM: Pretraining and Finetuning	Joe
W, 11/29	LLM: Prompting and Multilingualism	Freda
M, 12/4	Reading Period	
TBD	Final Exam	

Recap

Inference: solve arg max

Modeling: Define score function

$$\text{classify}(\mathbf{x}) = \arg\max_{y} \text{score}(\mathbf{x}, y; \mathbf{w})$$

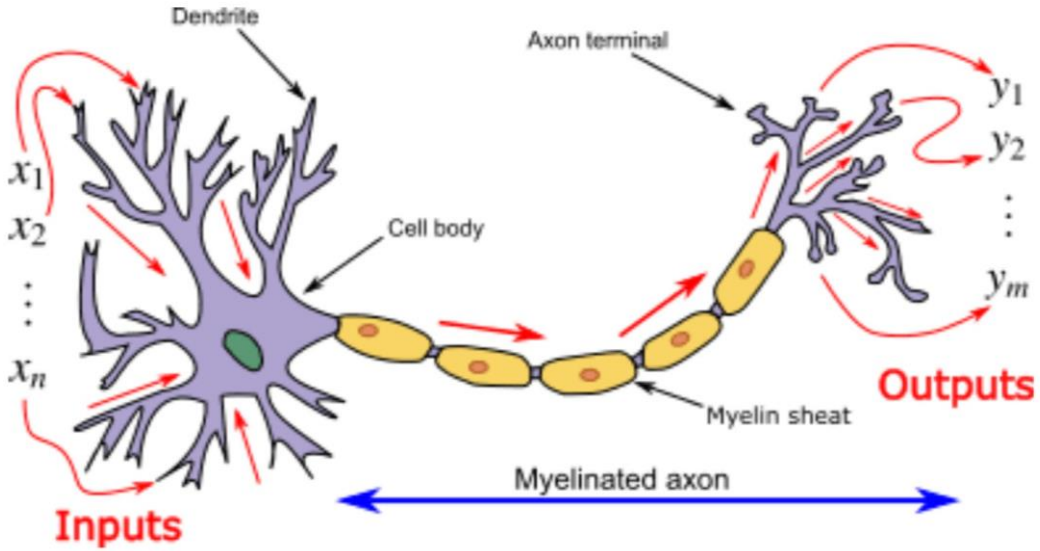
Learning: choose parameter

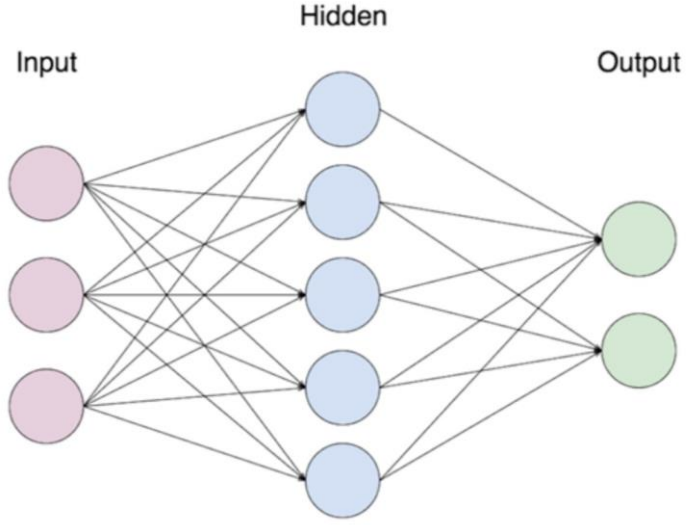
This Lecture (and the next)

- Neural networks
  - **Basics: Perceptron and multi-layer perceptron**
  - Convolutional neural networks
  - Recurrent and recursive neural networks
  - Attention
  - Transformers

What is a neural network?

- A neural network is a function
  - It has inputs and outputs
  - “Neural modeling” now is better thought of as dense representation learning





Classification with Neural Networks

Inference: solve arg max

Modeling: Neural Network

$$\text{classify}(\mathbf{x}) = \arg\max_{y} \text{score}(\mathbf{x}, y; \mathbf{w})$$

Learning: choose parameter

With a neural network—based function $f_{\mathbf{w}}$, we input $\mathbf{x}$ and collects a vector $\hat{\mathbf{y}} = f_{\mathbf{w}}(\mathbf{x})$; $\text{score}(\mathbf{x}, y; \mathbf{w})$ is defined by selecting the corresponding entry in $\hat{\mathbf{y}}$.

Notations

$\mathbf{u} =$ a vector

$u_i =$ entry i in the vector

$\mathbf{W} =$ a matrix

$w_{i,j} =$ entry (i, j) in the matrix

$\boldsymbol{x}/\mathcal{X} =$ a structured object

$x_i =$ Entry i in the structured object

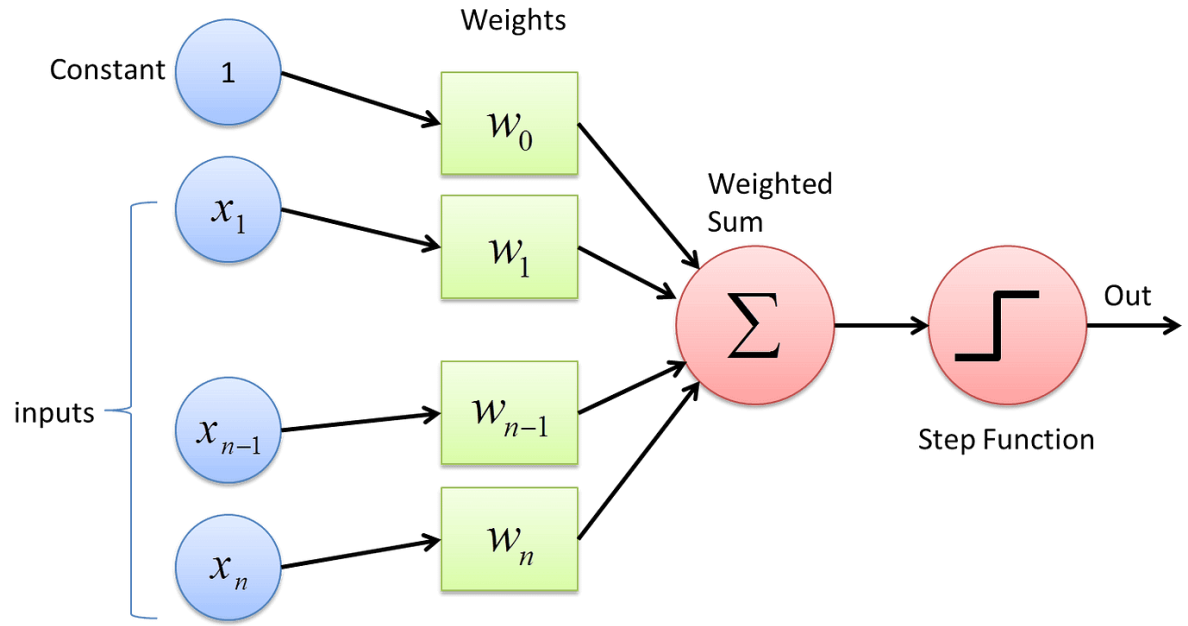
Perceptron

$$\text{perceptron}(\mathbf{x}) = \text{step}\left(\mathbf{w}^\top \mathbf{x} + b\right)$$

activation function

affine transform

$$\text{step}(z) = \begin{cases} 1 & if\ z \geq 0 \\ 0 & if\ z < 0 \end{cases}$$



Can be written as $\text{step}\left(\mathbf{w}^\top \mathbf{x}\right)$ if one entry of $\mathbf{x}$ is constant.

Perceptron: Learning

$$\text{perceptron}(\mathbf{x}) = \text{step}\left(\mathbf{w}^\top \mathbf{x}\right)$$

$$\text{step}(z) = \begin{cases} 1 & if\ z \geq 0 \\ 0 & if\ z < 0 \end{cases}$$

Predict the label $\hat{y}^{(i)} = \text{perceptron}(\mathbf{x}^{(i)})$

Update weights $\mathbf{w} = \mathbf{w} + \eta \cdot \left(y^{(i)} - \hat{y}^{(i)}\right) \mathbf{x}^{(i)}$

learning rate

gold standard

predicted label

Perceptron: Learning

$$\text{perceptron}(\mathbf{x}) = \text{step}\left(\mathbf{w}^\top \mathbf{x}\right)$$

$$\hat{y}^{(i)} = \text{perceptron}(\mathbf{x}^{(i)})$$

$$\mathbf{w} = \mathbf{w} + \eta \cdot \left(y^{(i)} - \hat{y}^{(i)}\right) \mathbf{x}^{(i)}$$

$$\text{loss}(\mathbf{w}; \mathbf{x}^{(i)}, y^{(i)}) = (\hat{y}^{(i)} - y^{(i)})\mathbf{w}^\top \mathbf{x}^{(i)}$$

$\hat{y}^{(i)}$	$y^{(i)}$	$\text{sgn}\left(\text{loss}(\mathbf{w})\right)$
0	0	

Perceptron: Learning

$$\text{perceptron}(\mathbf{x}) = \text{step}\left(\mathbf{w}^\top \mathbf{x}\right)$$

$$\hat{y}^{(i)} = \text{perceptron}(\mathbf{x}^{(i)})$$

$$\mathbf{w} = \mathbf{w} + \eta \cdot \left(y^{(i)} - \hat{y}^{(i)}\right) \mathbf{x}^{(i)}$$

$$\text{loss}(\mathbf{w}; \mathbf{x}^{(i)}, y^{(i)}) = (\hat{y}^{(i)} - y^{(i)})\mathbf{w}^\top \mathbf{x}^{(i)}$$

$\hat{y}^{(i)}$	$y^{(i)}$	$\text{sgn}\left(\text{loss}(\mathbf{w})\right)$
0	0	0
1	0	

Perceptron: Learning

$$\text{perceptron}(\mathbf{x}) = \text{step}\left(\mathbf{w}^\top \mathbf{x}\right)$$

$$\hat{y}^{(i)} = \text{perceptron}(\mathbf{x}^{(i)})$$

$$\mathbf{w} = \mathbf{w} + \eta \cdot \left(y^{(i)} - \hat{y}^{(i)}\right) \mathbf{x}^{(i)}$$

$$\text{loss}(\mathbf{w}; \mathbf{x}^{(i)}, y^{(i)}) = (\hat{y}^{(i)} - y^{(i)})\mathbf{w}^\top \mathbf{x}^{(i)}$$

$\hat{y}^{(i)}$	$y^{(i)}$	$\text{sgn}\left(\text{loss}(\mathbf{w})\right)$
0	0	0
1	0	1
0	1	

Perceptron: Learning

$$\text{perceptron}(\mathbf{x}) = \text{step}\left(\mathbf{w}^\top \mathbf{x}\right)$$

$$\hat{y}^{(i)} = \text{perceptron}(\mathbf{x}^{(i)})$$

$$\mathbf{w} = \mathbf{w} + \eta \cdot \underline{\left(y^{(i)} - \hat{y}^{(i)}\right) \mathbf{x}^{(i)}}$$

$$\text{loss}(\mathbf{w}; \mathbf{x}^{(i)}, y^{(i)}) = (\hat{y}^{(i)} - y^{(i)})\mathbf{w}^\top \mathbf{x}^{(i)}$$

$$-\frac{\partial \text{loss}(\mathbf{w}; \mathbf{x}^{(i)}, y^{(i)})}{\partial \mathbf{w}}$$

$\hat{y}^{(i)}$	$y^{(i)}$	$\text{sgn}\left(\text{loss}(\mathbf{w})\right)$
0	0	0
1	0	1
0	1	1

This is stochastic gradient descent!

Neural Layer: Generalized Perceptron

- A neural layer = affine transformation + nonlinearity

$$\text{perceptron}(\mathbf{x}) = \text{step}\left(\mathbf{w}^\top \mathbf{x} + b\right)$$

activation function

affine transform

$$\text{neural\_layer}(\mathbf{x}) = g\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right)$$

- Output is a vector (results from multiple independent perceptrons).
- Can have other activation functions for nonlinearity.

Neural Layer: Generalized Perceptron

$$\text{neural\_layer}(\mathbf{x}) = g\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right) \qquad \mathbb{R}^{d_1} \rightarrow \mathbb{R}^{d_2}$$

- $\mathbf{x} \in \mathbb{R}^{d_1}$ is the input
- The output is a vector with d_2 entries
- $\mathbf{W}$ and $\mathbf{b}$ are trainable parameters

- Multiple neural layers can be stacked together

$$\mathbf{z}^{(1)} = g\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

hidden units

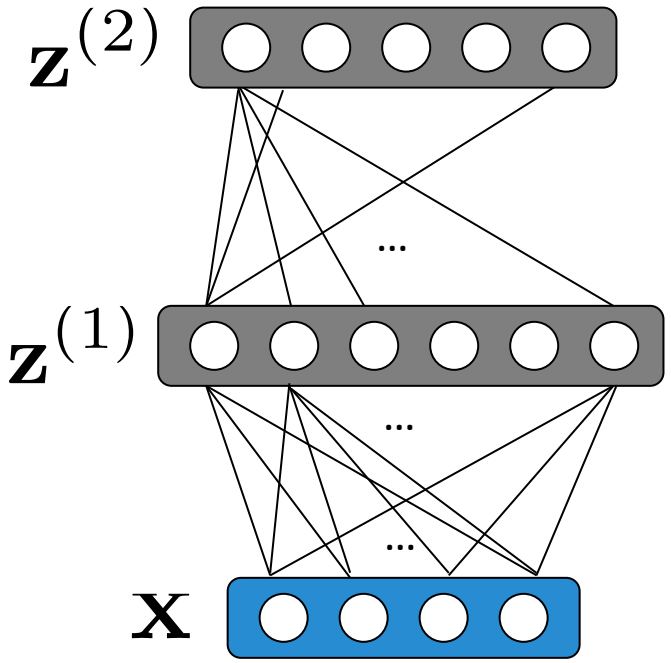
layer index

Stacking Neural Layers

$$\mathbf{z}^{(1)} = g\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

$$\mathbf{z}^{(2)} = g\left(\mathbf{W}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}\right)$$

$$\ldots$$



- Use output of one layer as input to the next
- *Feed-forward* and/or *fully-connected* layers
- Also called *multi-layer perceptron (MLP)*

Nonlinearities

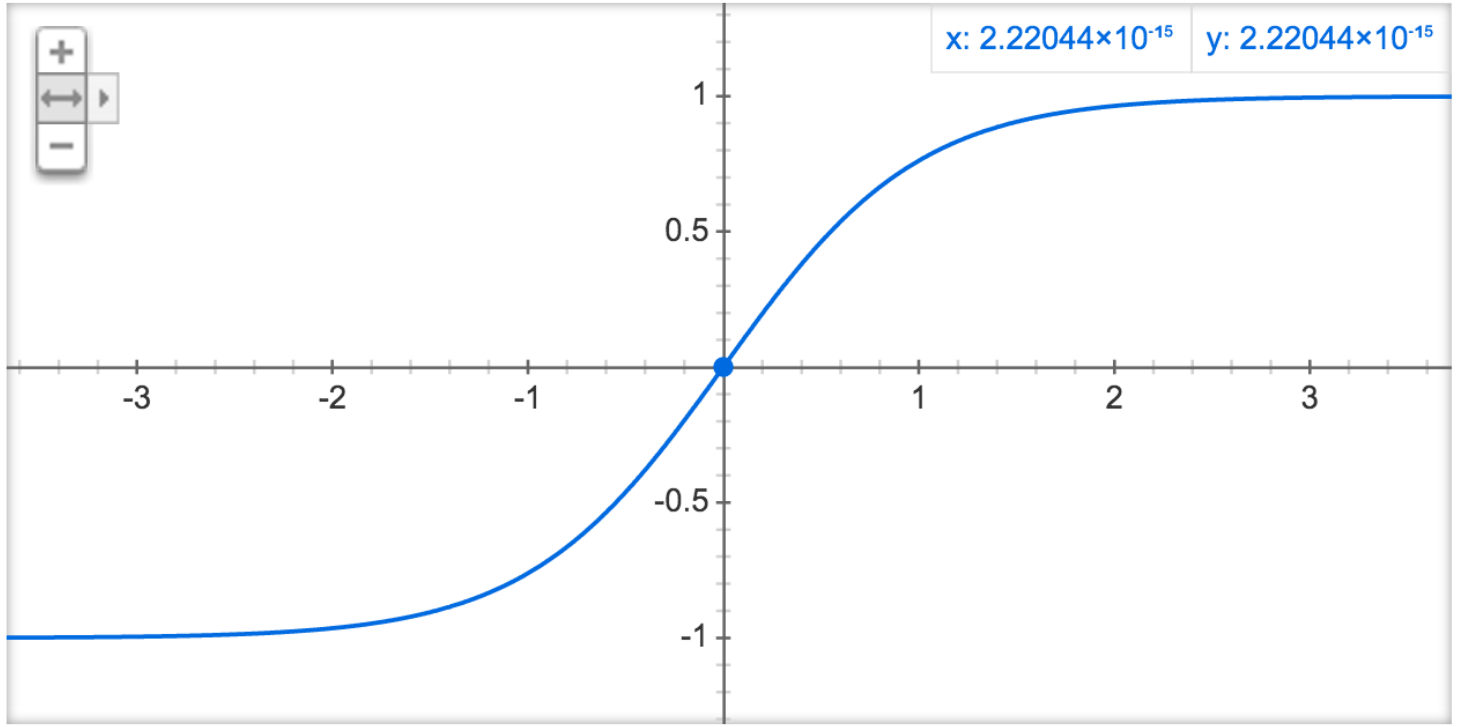
$$\mathbf{z}^{(1)} = g\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

- g can be applied to each entry in a vector in an element-wise manner
- Common activation functions: tanh, sigmoid, and ReLU

- Why nonlinearities?
- Otherwise stacking neural layers results in a simple affine transform.

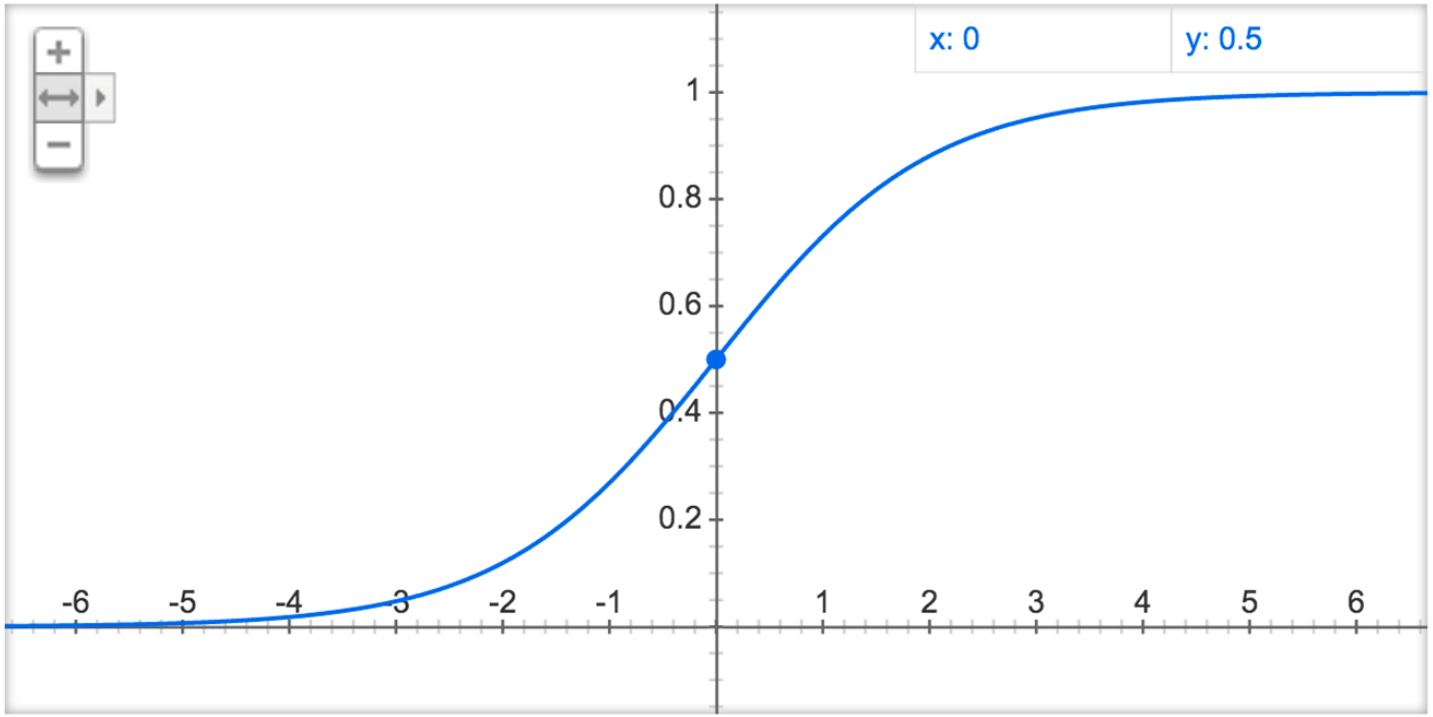
Nonlinearities: tanh

$$y = \tanh(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}}$$



Nonlinearities: sigmoid

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



Nonlinearities: sigmoid and tanh

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

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

$$= \frac{1-e^{-2x}}{1+e^{-2x}}$$

divide both sides by e^{x}

Nonlinearities: sigmoid and tanh

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

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

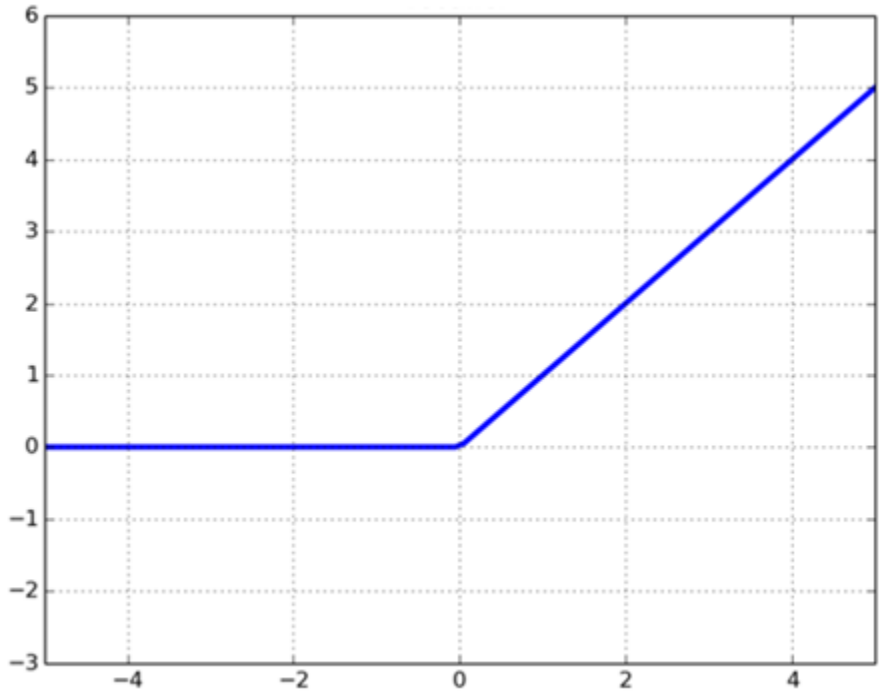
divide both sides by e^{x}

$$= \frac{1-e^{-2x}}{1+e^{-2x}}$$

$$= \sigma(2x) - (1-\sigma(2x)) = 2\sigma(2x) - 1$$

Nonlinearity: Rectified Linear Unit (ReLU)

$$y = \mathrm{ReLU}(x) = \max\{0, x\}$$



Sentiment Classification with Neural Network

- Two-layer perceptron

$$\mathbf{z}^{(1)} = g\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

$$\mathbf{s} = \mathbf{W}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}$$

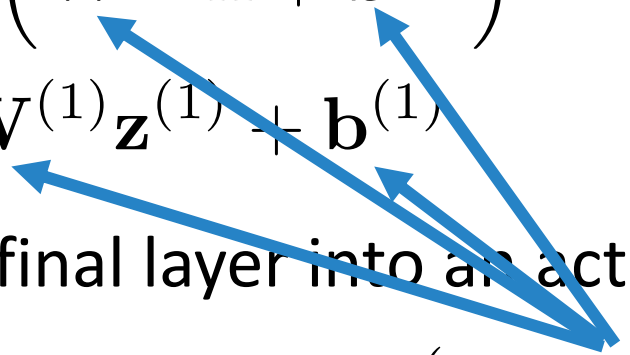
- We empirically don't pass the final layer into an activation function
- How can we get $\mathbf{x}$ for a sentence?
  - Average word embeddings
  - More complicated neural network structures

Sentiment Classification with Neural Network

- Two-layer perceptron

$$\mathbf{z}^{(1)} = g\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

$$\mathbf{s} = \mathbf{W}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}$$



- We empirically don't pass the final layer into an activation function

$$\text{classify}(\mathbf{x}) = \arg\max_{y} \text{score}(\mathbf{x}, y; \mathbf{w})$$

$$\mathbf{s} = \begin{bmatrix} \text{score}(\mathbf{x}, 0; \mathbf{w}) \\ \text{score}(\mathbf{x}, 1; \mathbf{w}) \end{bmatrix}$$

Sentiment Classification with Neural Network

$$\mathbf{s} = \begin{bmatrix} \text{score}(\mathbf{x}, 0; \mathbf{w}) \\ \text{score}(\mathbf{x}, 1; \mathbf{w}) \end{bmatrix} \overset{\text{softmax}}{\Rightarrow} \mathbf{p} = \begin{bmatrix} \frac{e^{\text{score}(\mathbf{x},0;\mathbf{w})}}{Z} \\ \frac{e^{\text{score}(\mathbf{x},1;\mathbf{w})}}{Z} \end{bmatrix}$$

$$Z = e^{\text{score}(\mathbf{x},0;\mathbf{w})} + e^{\text{score}(\mathbf{x},1;\mathbf{w})}$$

Training

$$\mathbf{s} = \begin{bmatrix} \text{score}(\mathbf{x}, 0) \\ \text{score}(\mathbf{x}, 1) \end{bmatrix} \overset{\text{softmax}}{\Rightarrow} \mathbf{p} = \begin{bmatrix} \frac{e^{\text{score}(\mathbf{x},0)}}{Z} \\ \frac{e^{\text{score}(\mathbf{x},1)}}{Z} \end{bmatrix}$$

Maximize the probability of gold standard label

$$\text{loss} = -\log P(y \mid \mathbf{x}) = -\log \left(\frac{e^{\text{score}(\mathbf{x},y;\mathbf{w})}}{Z} \right)$$

Also called *cross entropy* (between $\mathbf{p}$ and the 1-hot gold standard distribution) loss.

Backpropagation

- Chain rule: suppose $y = f(x), z = g(y)$, then $\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y} \cdot \frac{\mathrm{d}y}{\mathrm{d}x}$

$$\mathbf{z}^{(1)} = g\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

$$\mathbf{s} = \mathbf{W}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}$$

$$\mathrm{loss}(\mathbf{s}; \mathbf{x}, y) = -\log\left(\frac{e^{s_y}}{Z}\right)$$

Now we have $\frac{\partial \mathrm{loss}}{\partial \mathbf{s}}$, how should we update $\mathbf{W}^{(0)}$?

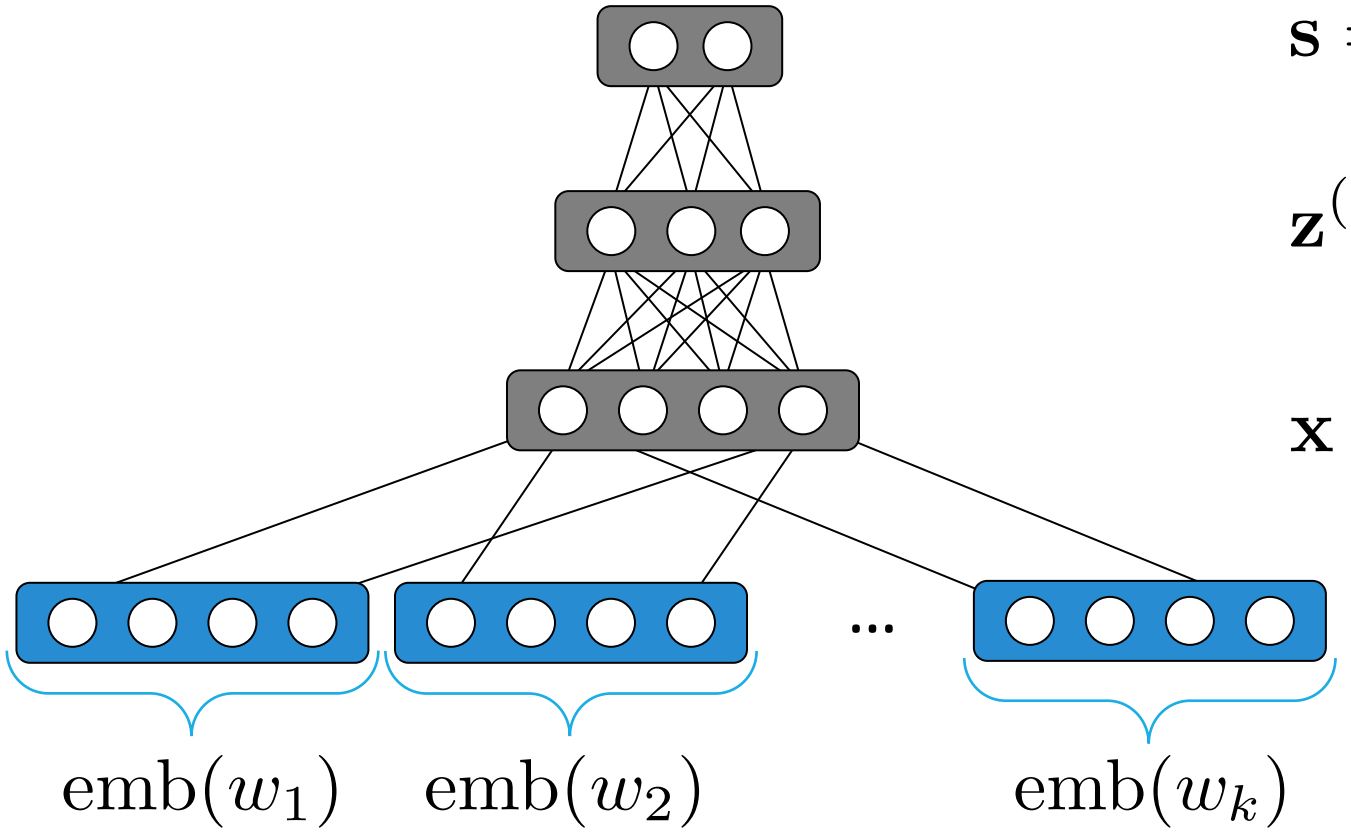
$$\frac{\partial \mathrm{loss}}{\partial \mathbf{W}^{(0)}} = \frac{\partial \mathrm{loss}}{\partial \mathbf{s}} \frac{\partial \mathbf{s}}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{W}^{(0)}}$$

Backpropagation

- Caveat: after adding nonlinearity, there's no guarantee on the convexity of the MLP
  Using gradient-based methods can result in local optimum

- We are usually happy with the local optima in practice

Visualization of Model Architecture



$$\mathbf{s} = \mathbf{W}^{(1)}\mathbf{z}^{(1)} + \mathbf{b}^{(1)}$$

$$\mathbf{z}^{(1)} = g\left(\mathbf{W}^{(0)}\mathbf{x} + \mathbf{b}^{(0)}\right)$$

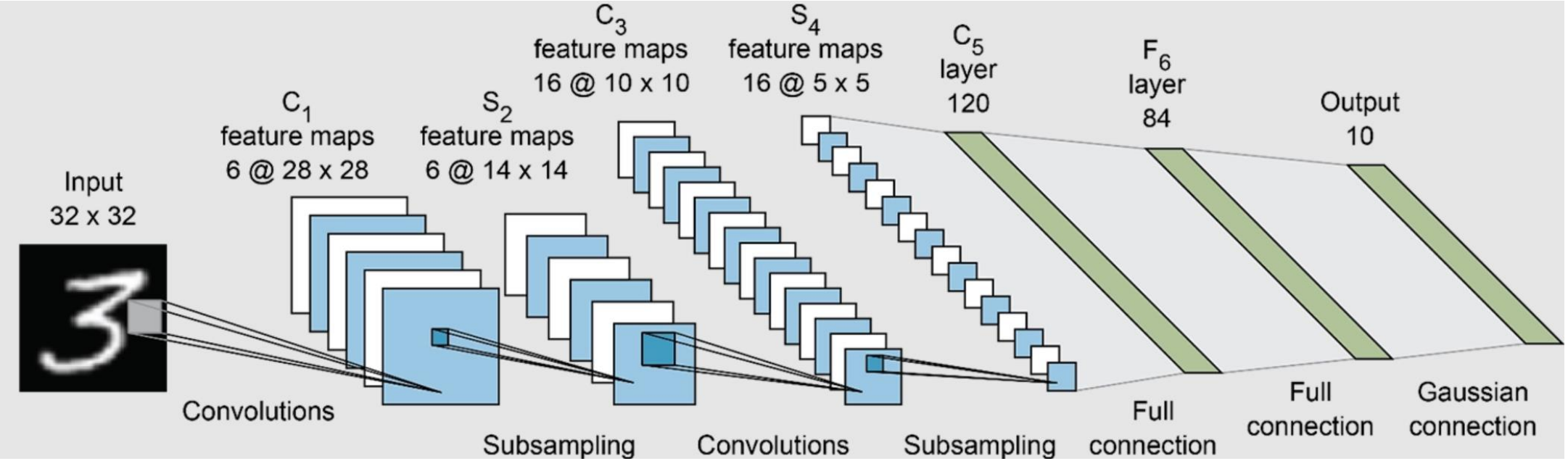
$$\mathbf{x} = \frac{1}{k}\sum_{i=1}^{k}\text{emb}(w_k)$$

This Lecture (and the next)

- Neural networks
  - Basics: Perceptron and multi-layer perceptron
  - **Convolutional neural networks**
  - Recurrent and recursive neural networks
  - Attention
  - Transformers

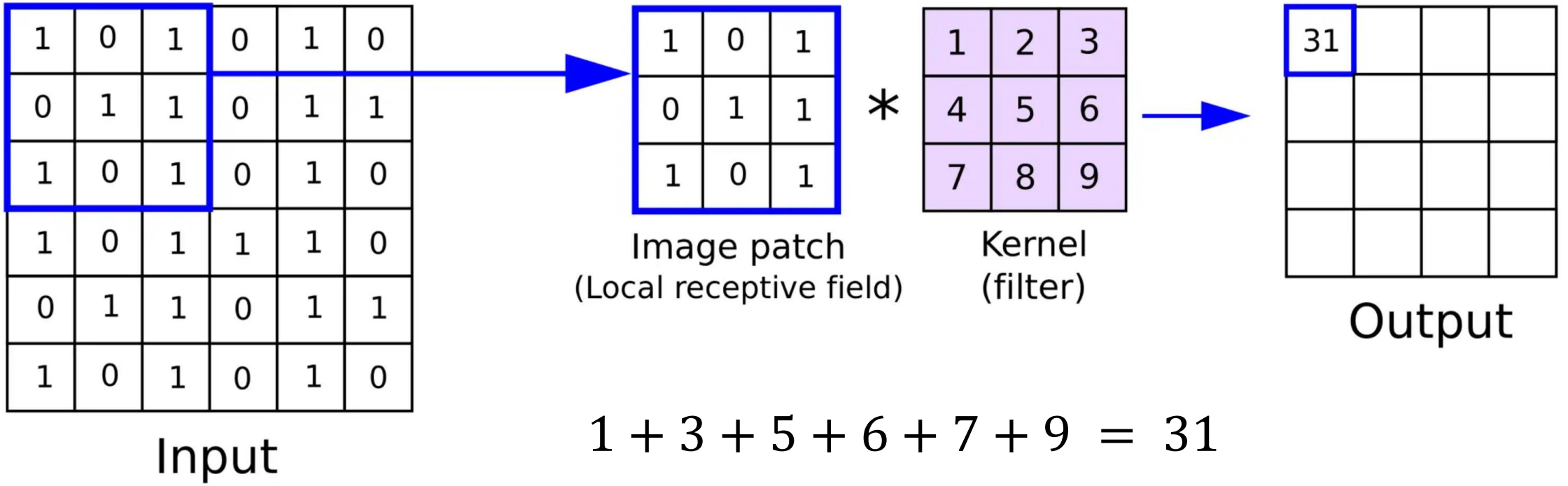
Convolutional Neural Networks

- Introduced for vision tasks; also used in NLP to extract feature vectors



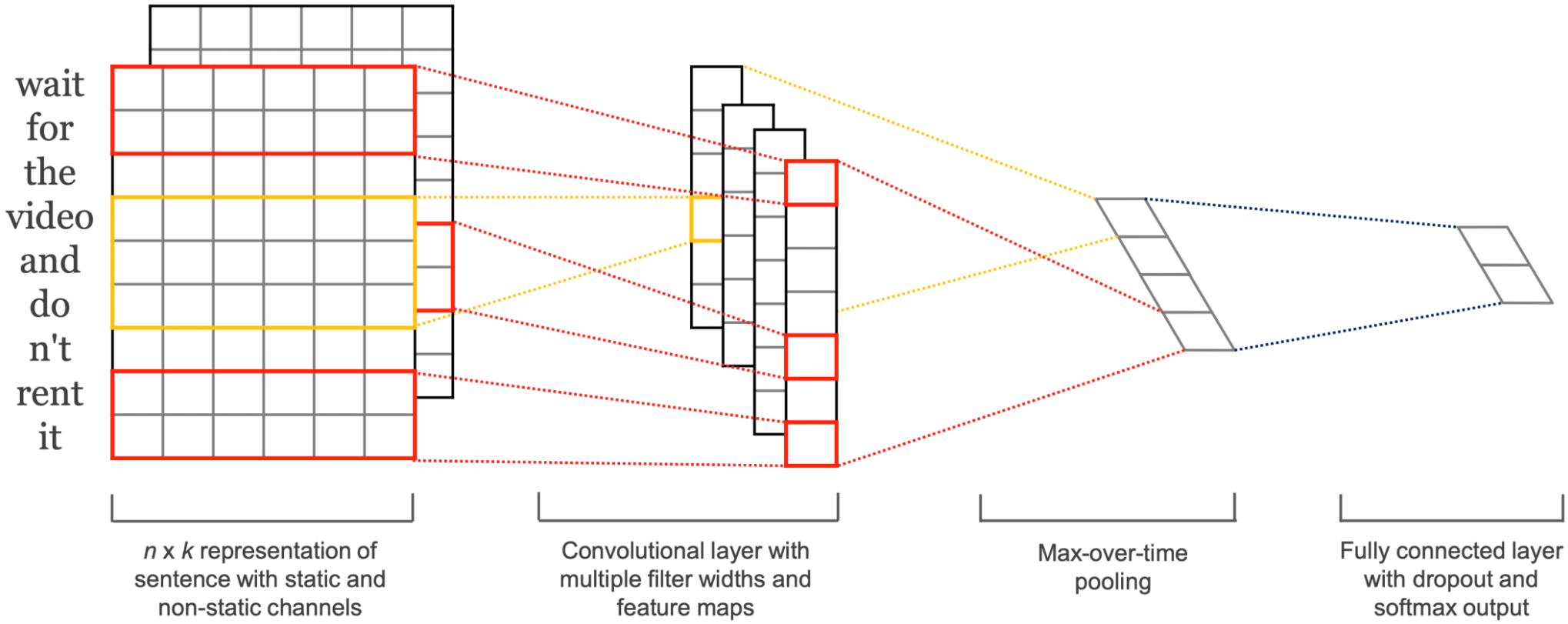
Convolutional Neural Networks

- Introduced for vision tasks; also used in NLP to extract feature vectors



$$1 + 3 + 5 + 6 + 7 + 9 = 31$$

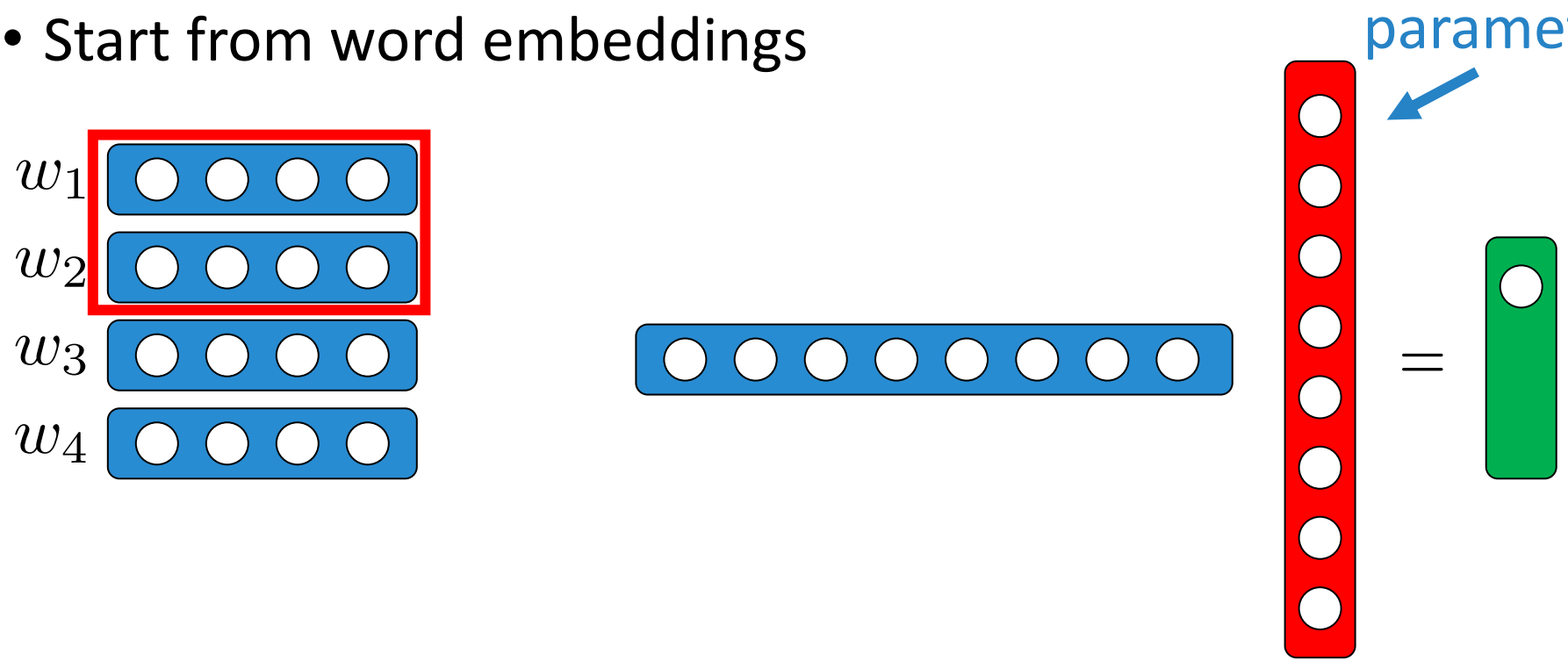
From 2D to 1D: Overview



Source: Y. Kim.(2014). Convolutional Neural Networks for Sentence Classification

Kernel/Filter

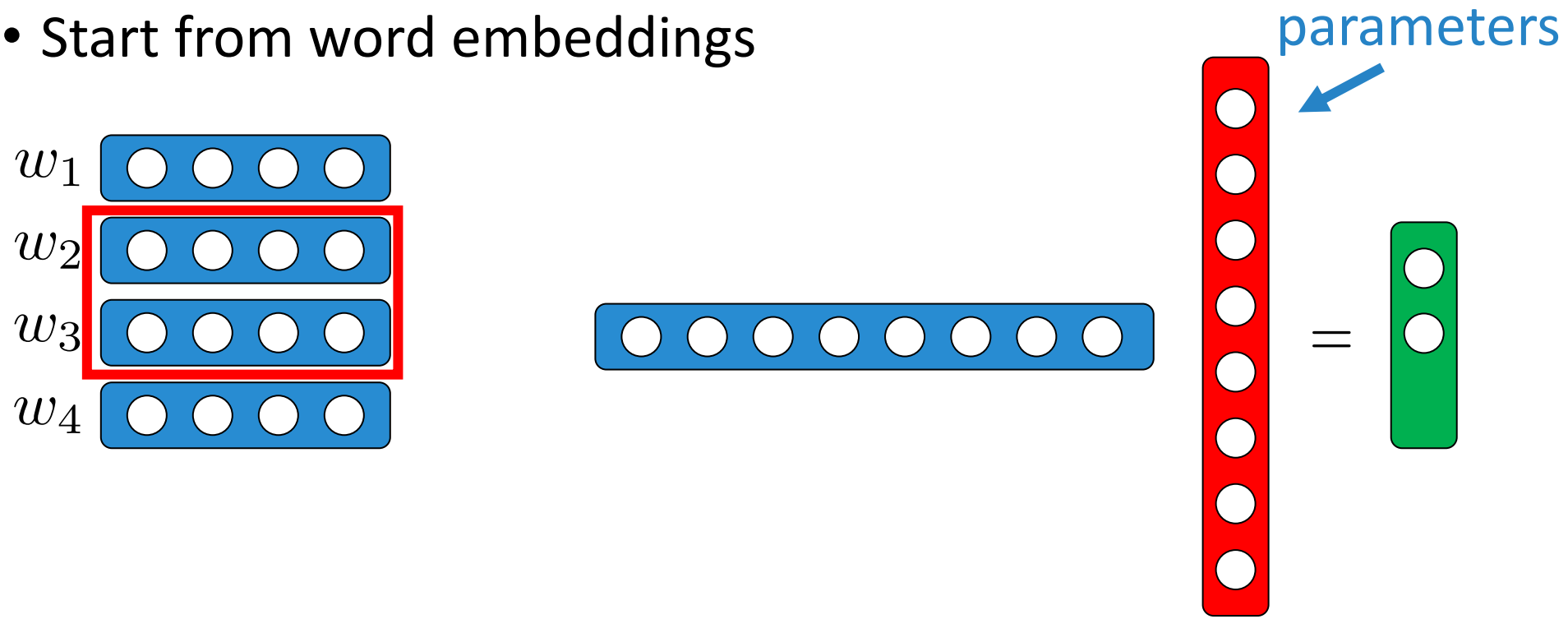
- Start from word embeddings



- Take dot product between filter and (stretched) word embeddings

Kernel/Filter

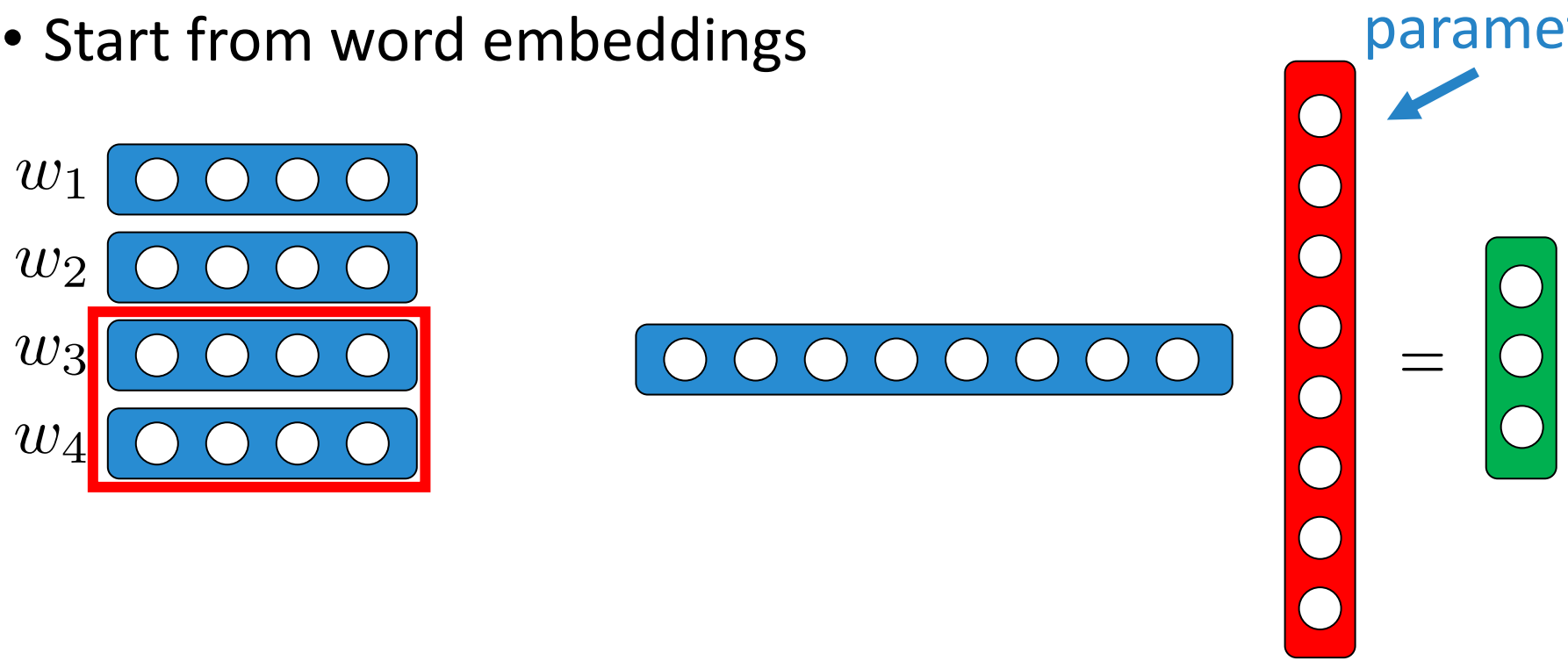
- Start from word embeddings



- Take dot product between filter and (stretched) word embeddings

Kernel/Filter

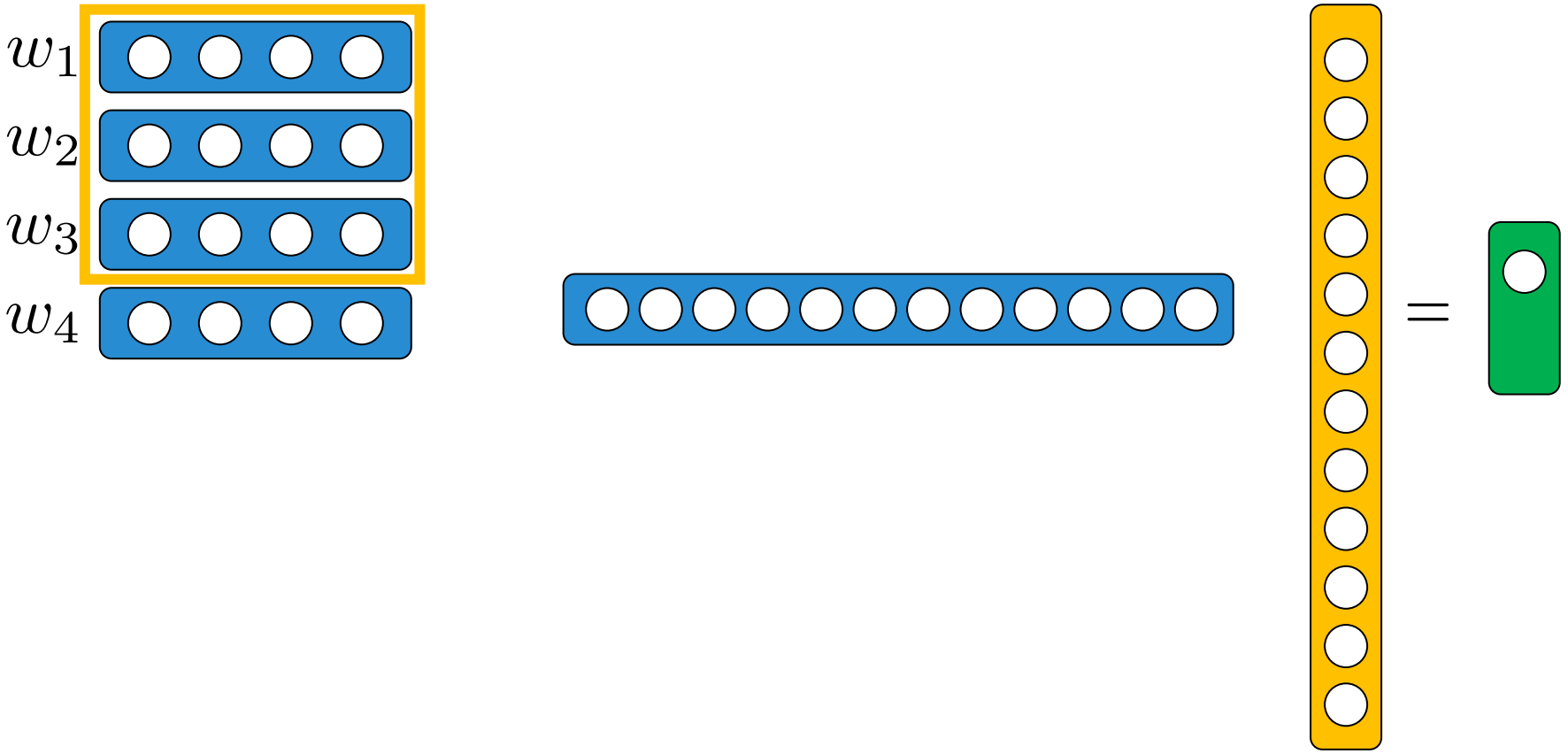
- Start from word embeddings



- Take dot product between filter and (stretched) word embeddings

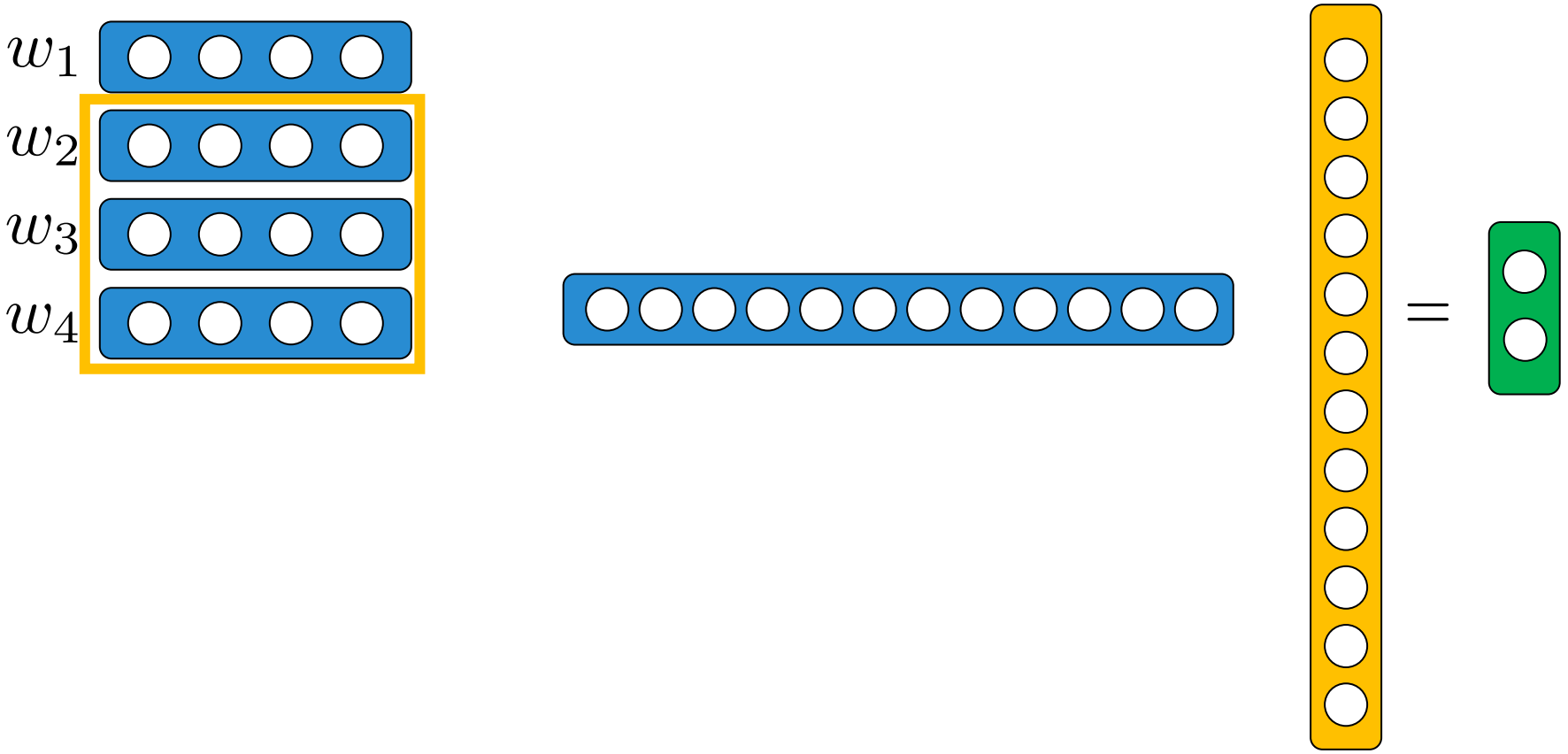
Kernel/Filter

- What about a kernel/filter with a different size?



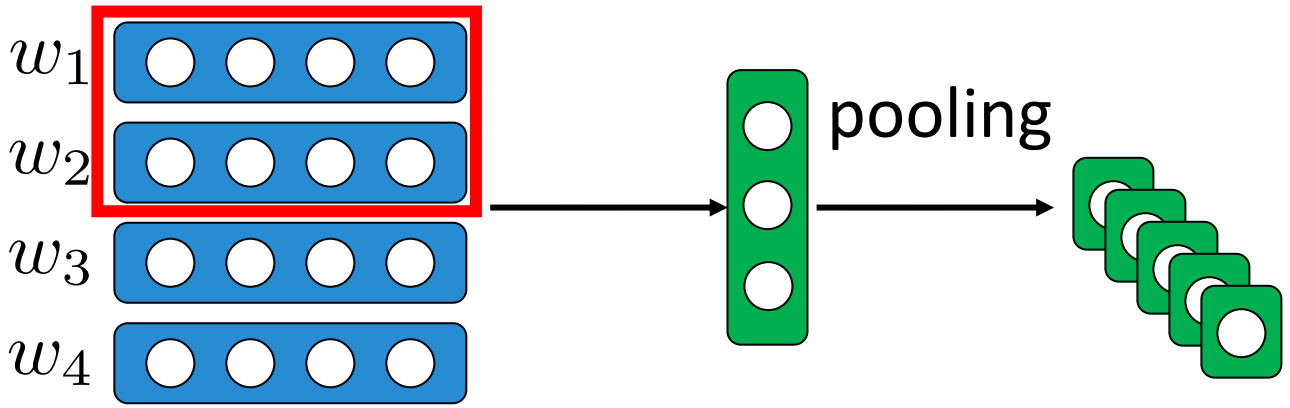
Kernel/Filter

- What about a kernel/filter with a different size?



Kernel/Filter: Pooling

- Each kernel/filter extracts one type of features
- However, a kernel's output size depends on sentence length
  A **fixed dimensional** vector is desirable for MLP inputs
- Solution: mean pooling/max pooling converts a vector to a scalar
- Final feature: concatenating pooling results of all filters



Convolutional Neural Networks

- Word order matters
  Example (kernel size = 2):
  a cat drinks milk → (a cat), (cat drinks), (drinks milk)
  a milk drinks cat → (a milk), (milk drinks), (drinks cat)
- An n-gram “matches” with a kernel when they have high dot product
- Cannot capture long-term dependency
- Often used for character-level processing: filters look at character n-grams

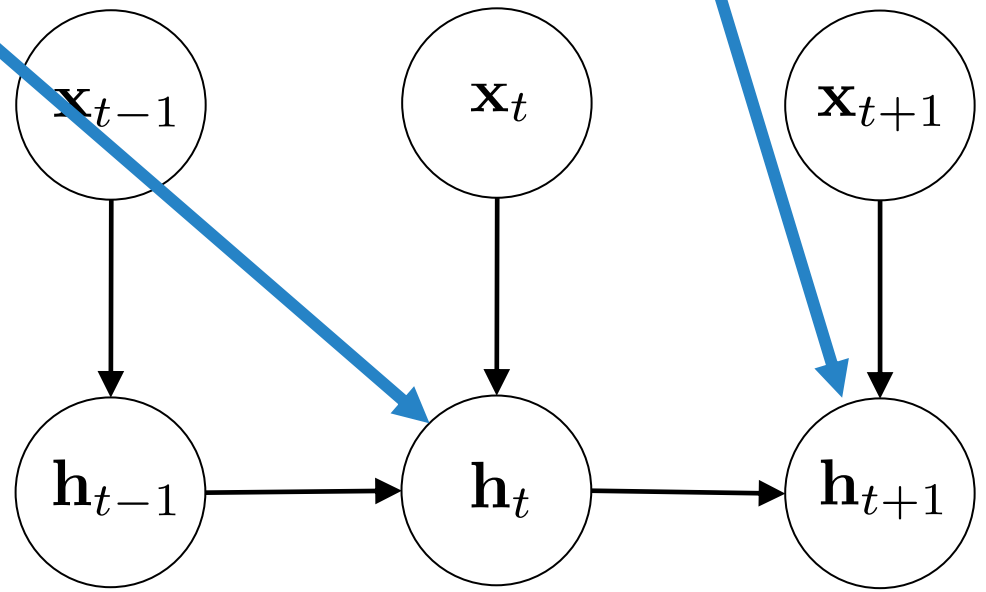
This Lecture (and the next)

- Neural networks
  - Basics: Perceptron and multi-layer perceptron
  - Convolutional neural networks
  - **Recurrent and recursive neural networks**
  - Attention
  - Transformers

Recurrent Neural Networks

- Idea: apply the same transformation to tokens in time order

$$\mathbf{h}_t = \mathbf{W}[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b} \qquad \mathbf{h}_{t+1} = \mathbf{W}[\mathbf{x}_{t+1}; \mathbf{h}_t] + \mathbf{b}$$



Recurrent Neural Networks

- Gradient update for $\mathbf{h}_t = \mathbf{W}[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}$
- Suppose $\mathbf{h}_k$ is the representation passed to the classifier

  We can easily calculate $\dfrac{\partial \text{loss}}{\partial \mathbf{h}_k}$

- What about $\dfrac{\partial \text{loss}}{\partial \mathbf{W}}$?

$$\frac{\partial \text{loss}}{\partial \mathbf{W}} = \sum_{t=1}^{k} \frac{\partial \text{loss}}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}}$$

$$\frac{\partial \text{loss}}{\partial \mathbf{h}_t} = \frac{\partial \text{loss}}{\partial \mathbf{h}_{t+1}} \frac{\partial \mathbf{h}_{t+1}}{\partial \mathbf{h}_t}$$

Recurrent Neural Networks

- What's the problem with $\mathbf{h}_t = \mathbf{W}[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}$?

$$\begin{aligned}\mathbf{h}_t &= \mathbf{W}[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b} \\ &= \mathbf{W}[\mathbf{x}_t; (\mathbf{W}[\mathbf{x}_{t-1}; \mathbf{h}_{t-2}] + \mathbf{b})] + \mathbf{b}\end{aligned}$$

- Absolute value of entries grow exponentially w.r.t. sequence length

- What if we add nonlinearity (e.g., tanh/sigmoid)?
- Values (and therefore gradients) vanish exponentially

Long Short-Term Memory Networks

Designed to tackle the gradient vanishing problem
[Hochreiter and Schmidhuber, 1997]

- Forget gate: $\mathbf{f}_t = \sigma\left(\mathbf{W}_f[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_f\right)$
- Input gate: $\mathbf{i}_t = \sigma\left(\mathbf{W}_i[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_i\right)$
- Cell: $\tilde{\mathbf{c}}_t = \tanh\left(\mathbf{W}_c[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_c\right)$
- Update: $\mathbf{c}_t = \mathbf{f}_t * \mathbf{c}_{t-1} + \mathbf{i}_t * \tilde{\mathbf{c}}_t$
- Output gate: $\mathbf{o}_t = \sigma\left(\mathbf{W}_o[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_o\right)$
- Hidden state: $\mathbf{h}_t = \mathbf{o}_t * \tanh(\mathbf{c}_t)$
- Idea: keep entries in $\tilde{\mathbf{c}}_t$ and $\mathbf{h}_t$ in the range of $(-1, 1)$.

Gated Recurrent Units

Fewer parameters; generally works quite well

- Update gate: $\mathbf{z}_t = \sigma\left(\mathbf{W}_z[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_z\right)$
- Reset gate: $\mathbf{r}_t = \sigma\left(\mathbf{W}_r[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_r\right)$

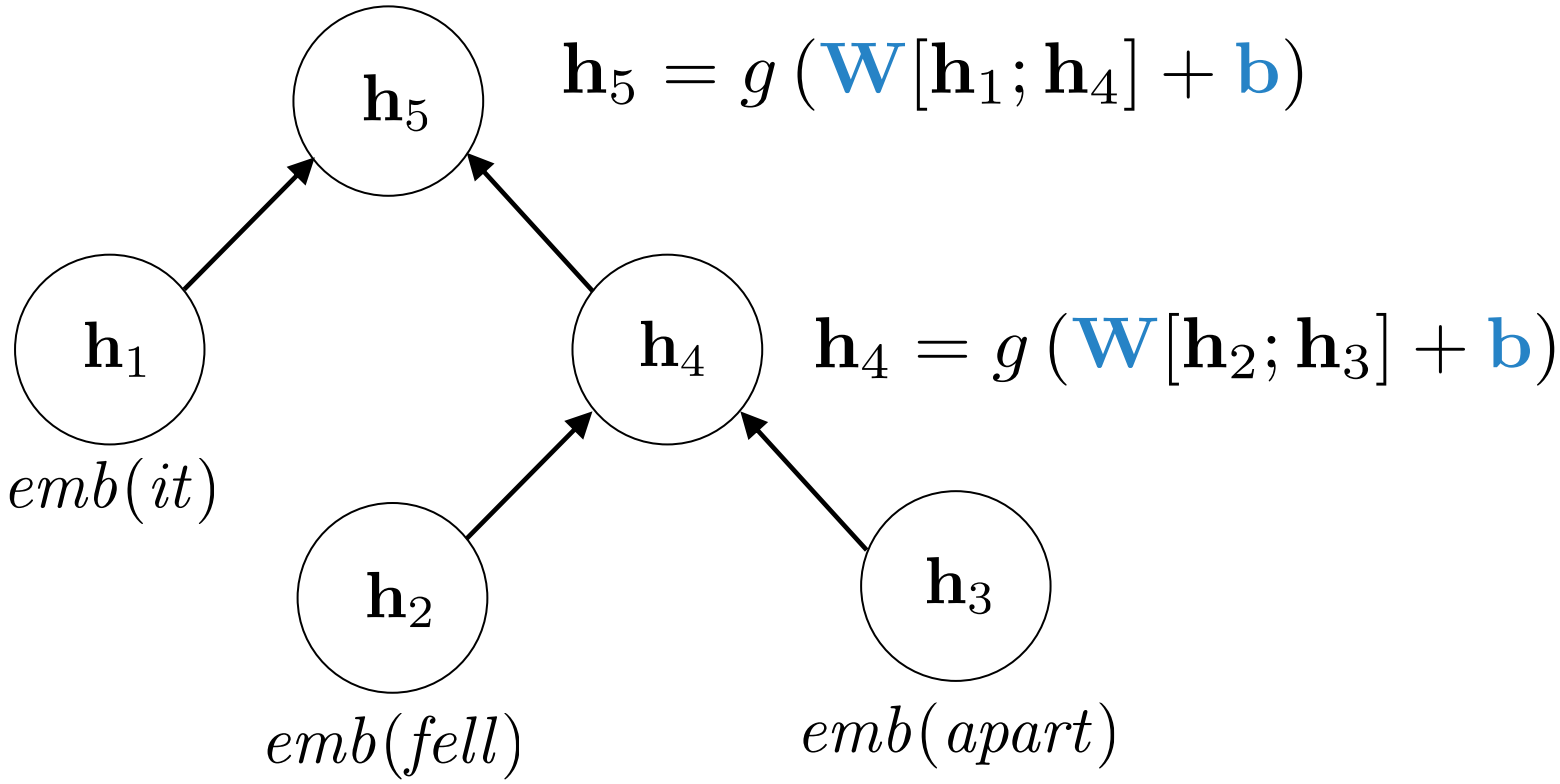
$$\mathbf{h}_t = (1 - \mathbf{z}_t) * \mathbf{h}_{t-1} + \mathbf{z}_t * \tanh\left(\mathbf{W}[\mathbf{x}_t; \mathbf{r}_t * \mathbf{h}_{t-1}] + \mathbf{b}\right)$$

RNN: Practical Approaches

- Gradient clip: gradient sometimes goes very large even with LSTMs. Empirical solution: After calculating gradients, require the L_2 norm to be at most C (set by hyperparameters)
- At time step t, what matters to $\mathbf{h}_t$ is mostly $\mathbf{x}_{t'}$ where $t'$is close to t [Khandelwal et al., ACL 2018]
- Bidirectional modeling typically results in more powerful features

Recursive Neural Networks

- Run constituency parser on sentence, and construct vector recursively
- All nodes share the same set of parameters [Socher et al., 2011&2013]



Recursive Neural Networks

- Tree LSTMs typically work well
  (slight modification of LSTM cells needed)

$$
\begin{aligned}
\mathbf{f}_t &= \sigma\left(\mathbf{W}_f[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_f\right) \\
\mathbf{i}_t &= \sigma\left(\mathbf{W}_i[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_i\right) \\
\tilde{\mathbf{c}}_t &= \tanh\left(\mathbf{W}_c[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_c\right) \\
\mathbf{c}_t &= \mathbf{f}_t * \mathbf{c}_{t-1} + \mathbf{i}_t * \tilde{\mathbf{c}}_t \\
\mathbf{o}_t &= \sigma\left(\mathbf{W}_o[\mathbf{x}_t; \mathbf{h}_{t-1}] + \mathbf{b}_o\right) \\
\mathbf{h}_t &= \mathbf{o}_t * \tanh(\mathbf{c}_t)
\end{aligned}
\quad \Longrightarrow \quad
\begin{aligned}
\mathbf{l}_n &= \sigma\left(\mathbf{W}_\ell[\mathbf{h}_\ell; \mathbf{h}_r] + \mathbf{b}_\ell\right) \\
\mathbf{r}_n &= \sigma\left(\mathbf{W}_r[\mathbf{h}_\ell; \mathbf{h}_r] + \mathbf{b}_r\right) \\
\tilde{\mathbf{c}}_n &= \tanh\left(\mathbf{W}_c[\mathbf{h}_\ell; \mathbf{h}_r] + \mathbf{b}_c\right) \\
\mathbf{c}_n &= \mathbf{l}_n * \mathbf{c}_\ell + \mathbf{r}_n * \mathbf{c}_r + \tilde{\mathbf{c}}_n \\
\mathbf{o}_n &= \sigma\left(\mathbf{W}_o[\mathbf{h}_\ell; \mathbf{h}_r] + \mathbf{b}_o\right) \\
\mathbf{h}_n &= \mathbf{o}_n * \tanh(\mathbf{c}_n)
\end{aligned}
$$

Recursive Neural Networks

- Tree LSTMs typically work well
  (slight modification of LSTM cells needed)
- Recursive neural networks with left-branching trees are basically equivalent to recurrent neural networks
- Syntactically meaningful parse trees are not necessary for good representations: instead, balanced trees work well for most tasks [Shi et al., EMNLP 2018]

This Lecture (and the next)

- Neural networks
  - Basics: Perceptron and multi-layer perceptron
  - Convolutional neural networks
  - Recurrent and recursive neural networks
  - **Attention**
  - **Transformers**