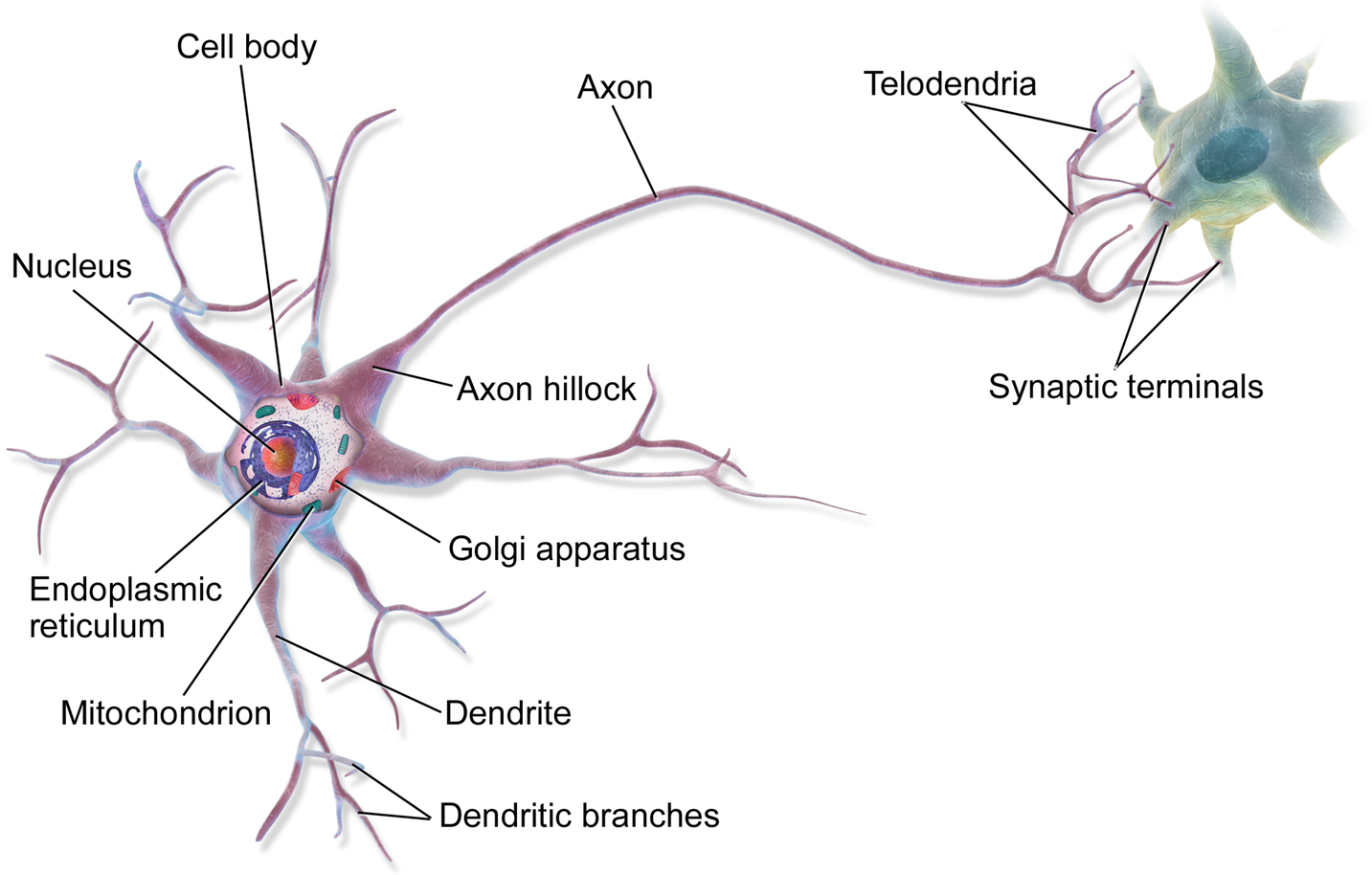
ORTA DOĞU TEKNİK ÜNİVERSİTESİ
MIDDLE EAST TECHNICAL UNIVERSITY

# CENG 463: Introduction to Natural Language Processing
# Neural Models

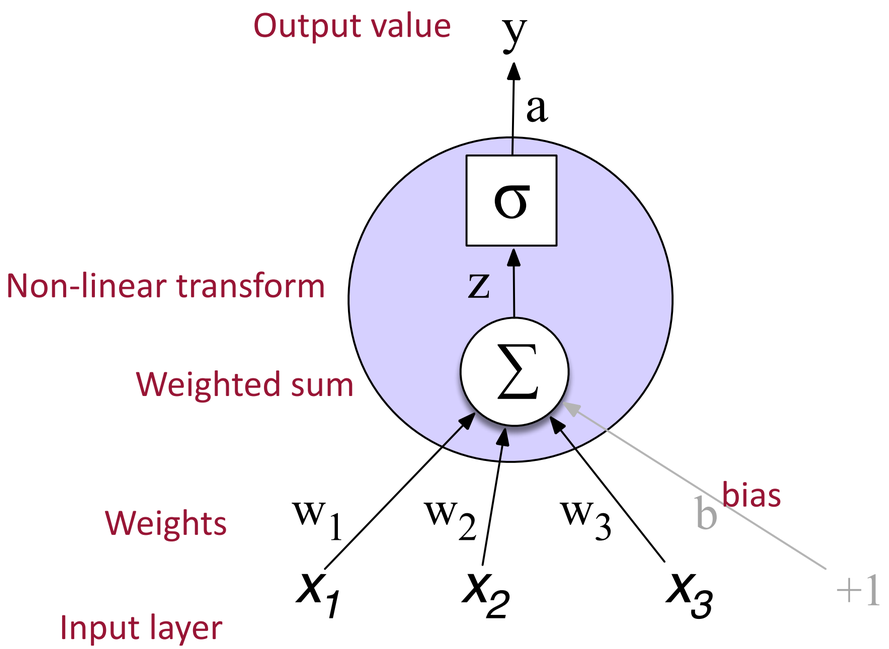
**Asst. Prof. Çağrı Toraman**
**Computer Engineering Department**
**ctoraman@ceng.metu.edu.tr**

**11.11.2025**

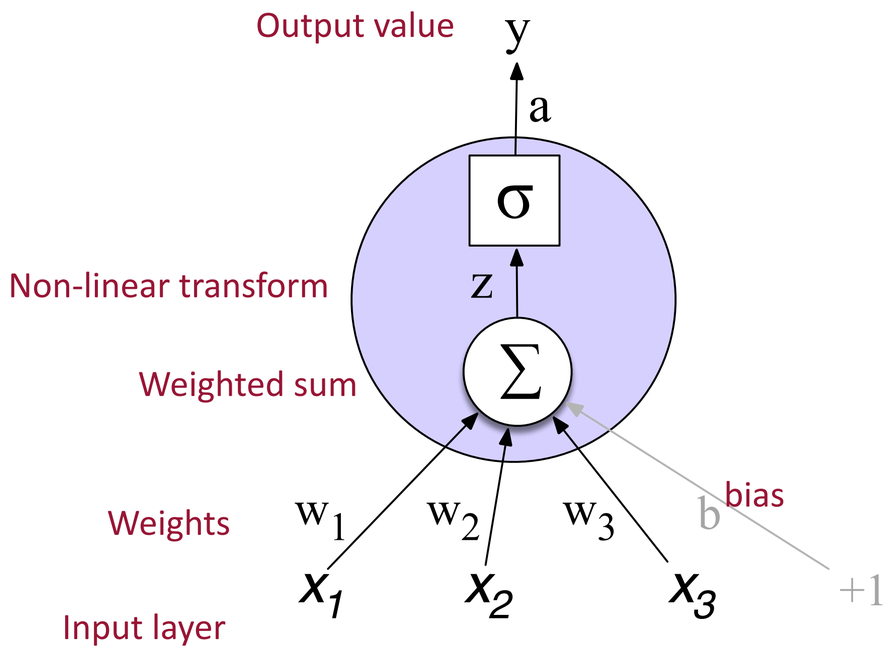
# Neural Networks



# Neural Networks



# Neural Networks




$y = a = f(z)$

**apply a nonlinear activation function f**
**(Why nonlinear? Check Perceptrons.)**

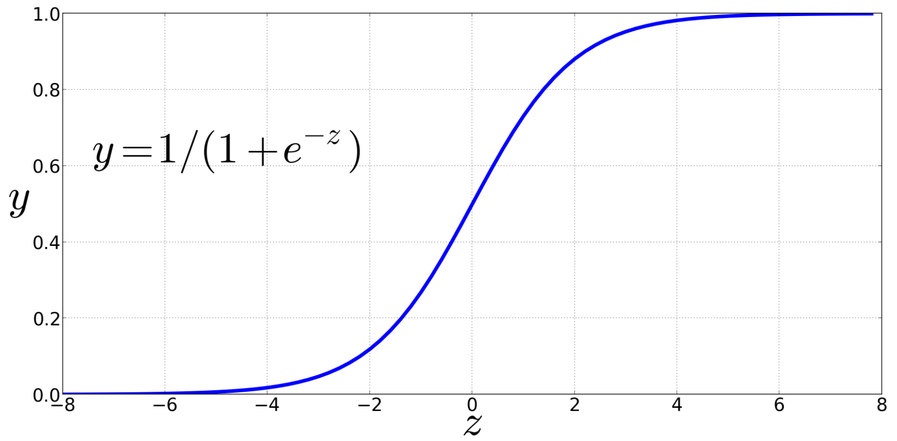
$z = w \cdot x + b$

**weighted sum of inputs, plus a bias**

# Neural Networks

**Sigmoid Non-Linear Activation Function**

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



# Neural Networks

$$y = s(w \cdot x + b) = \frac{1}{1 + \exp(-(w \cdot x + b))}$$

# Neural Networks

Suppose a unit has:

w = `[0.2,0.3,0.9]`

b = `0.5`

What happens with input x:

x = `[0.5,0.6,0.1]`

$$y = s(w \cdot x + b) = ?$$

# Neural Networks

Suppose a unit has:

w = [0.2,0.3,0.9]

b = 0.5

What happens with input x:

x = [0.5,0.6,0.1]

$$y = s(w \cdot x + b) =$$

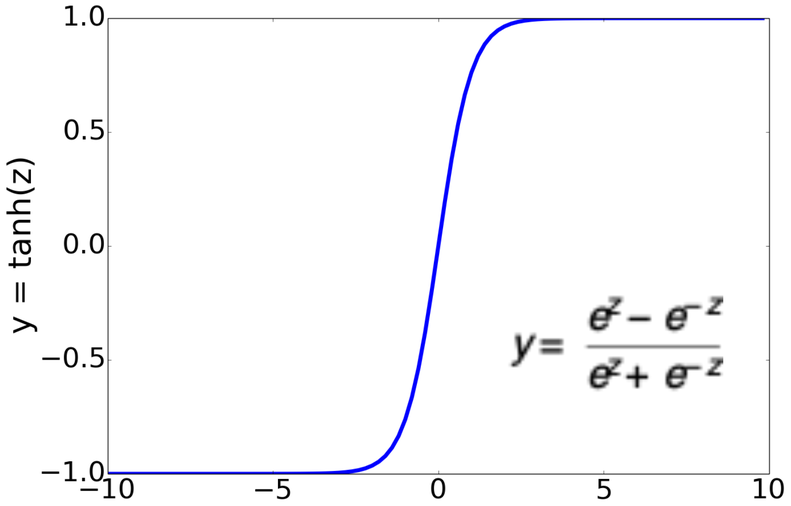
$$\frac{1}{1 + e^{-(w \cdot x + b)}} =$$

$$\frac{1}{1 + e^{-(.5*.2 + .6*.3 + .1*.9 + .5)}} =$$

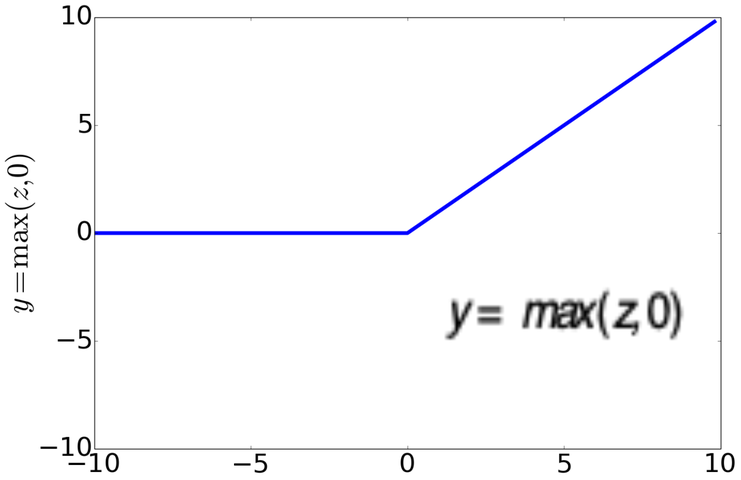
$$\frac{1}{1 + e^{-0.87}} = .70$$

# Neural Networks

Other non-linear activation functions:



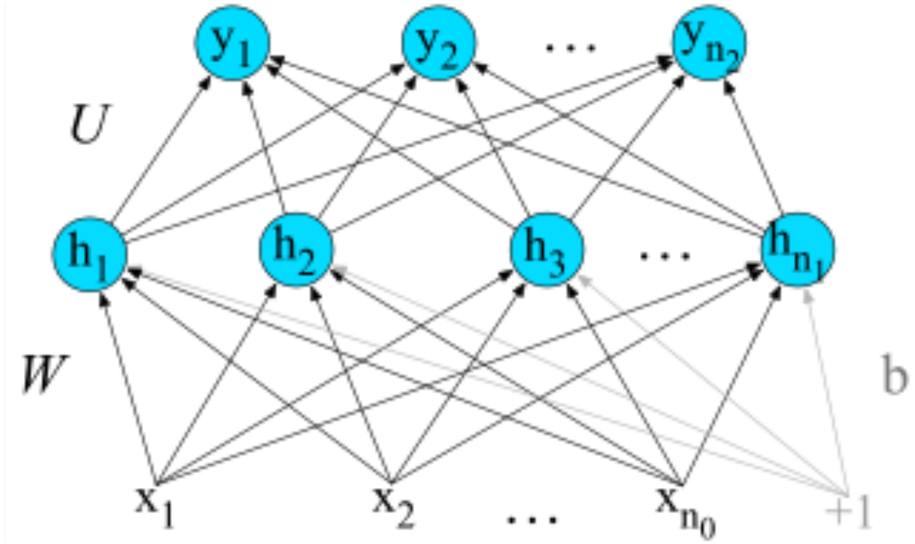

tanh




ReLU (Rectified Linear Unit)

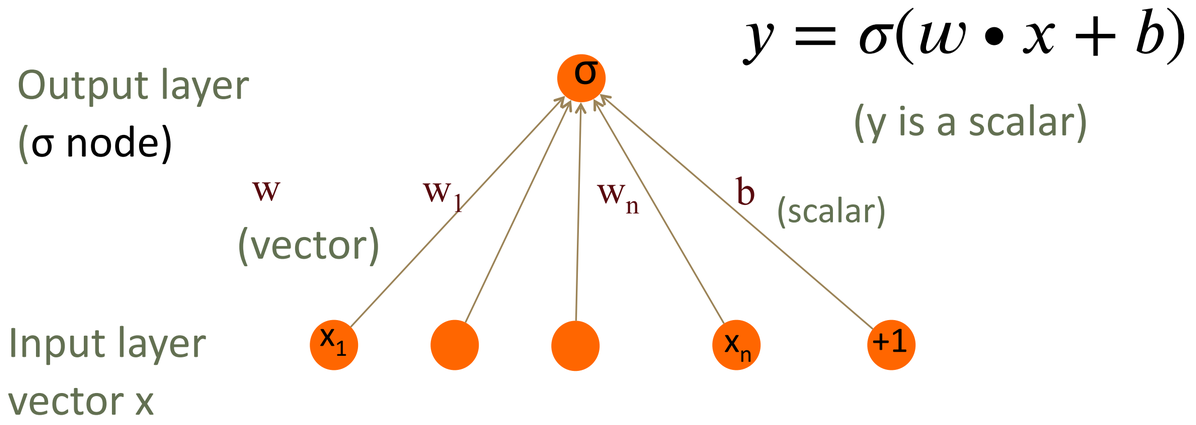
# Neural Networks

Feed-forward Neural Networks (or Multi-Layer Perceptrons):



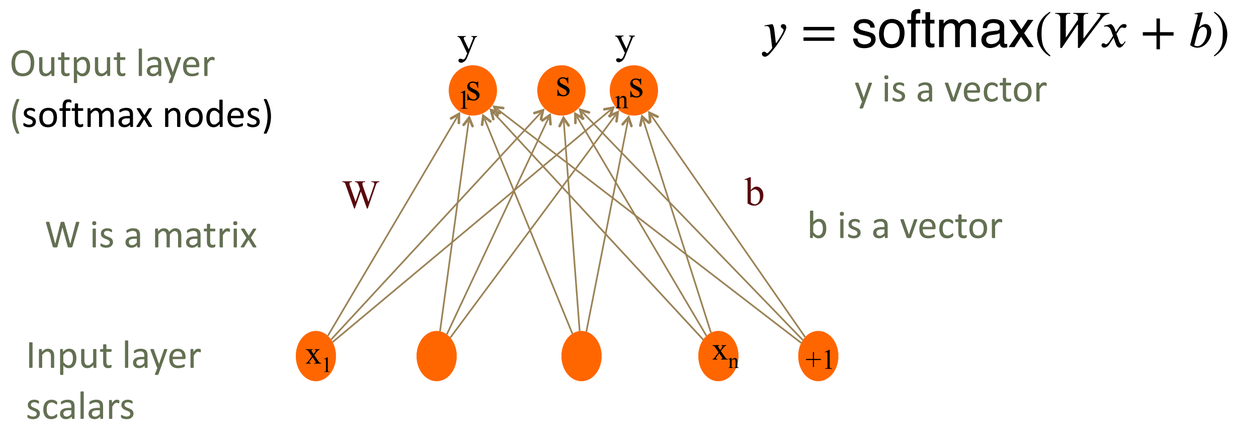
# Neural Networks

Binary Logistic Regression as 1-Layer Neural Network



# Neural Networks

Multinomial Logistic Regression as Fully Connected 1-Layer Neural Network



# Neural Networks

Softmax is a generalization of sigmoid activation function (binary vs. multi-class):

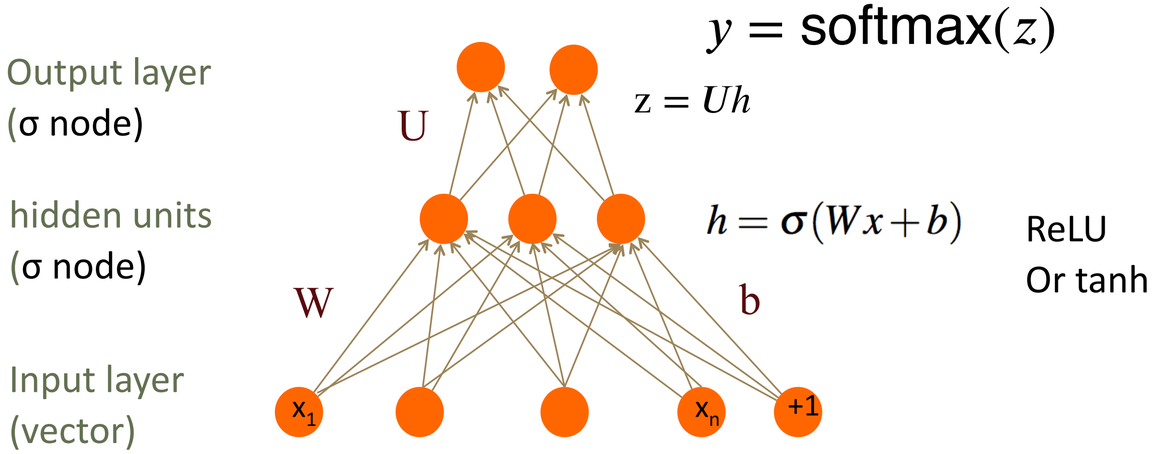
$$\text{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k}\exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k}\exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k}\exp(z_i)}\right] \qquad \text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad 1 \le i \le D$$

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

$$\text{softmax}(z) = [0.055, 0.090, 0.006, 0.099, 0.74, 0.010]$$

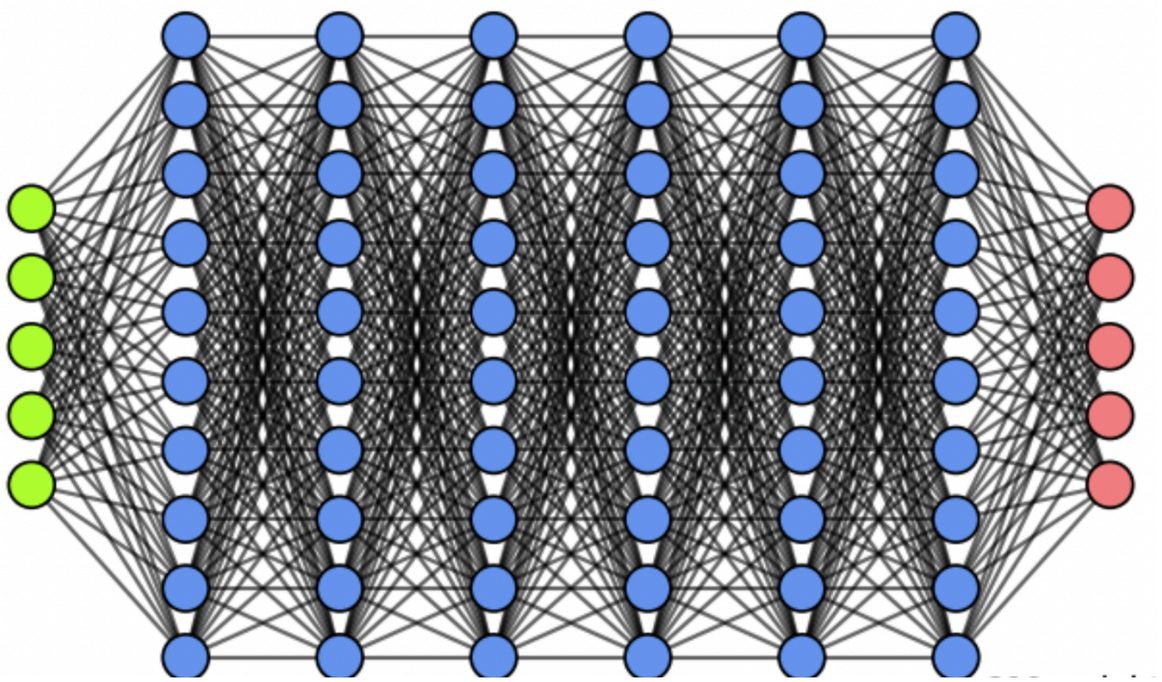
# Neural Networks

More layers:



# Neural Networks

More layers: Deep Learning



# Neural Networks for NLP

Recent models (BERT, GPT etc.) use more powerful neural networks.

Let’s consider a simple use case:

- Text Classification (Sentiment Analysis)

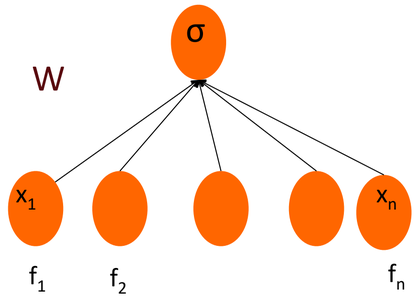
# Neural Networks for NLP

| Var | Definition |
|---|---|
| $x_1$ | count(positive lexicon) $\in$ doc) |
| $x_2$ | count(negative lexicon) $\in$ doc) |
| $x_3$ | $\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$ |
| $x_4$ | count(1st and 2nd pronouns $\in$ doc) |
| $x_5$ | $\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$ |
| $x_6$ | log(word count of doc) |

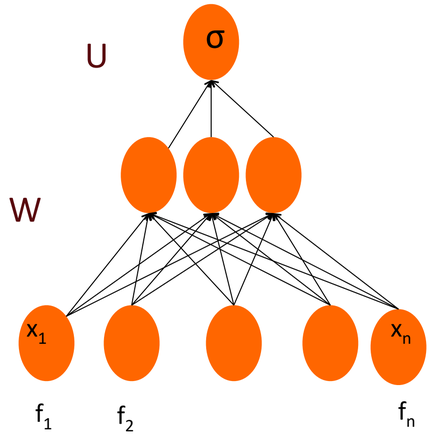
# Neural Networks for NLP

Same task (sentiment analysis with hand-crafted features), two solutions:

Logistic Regression

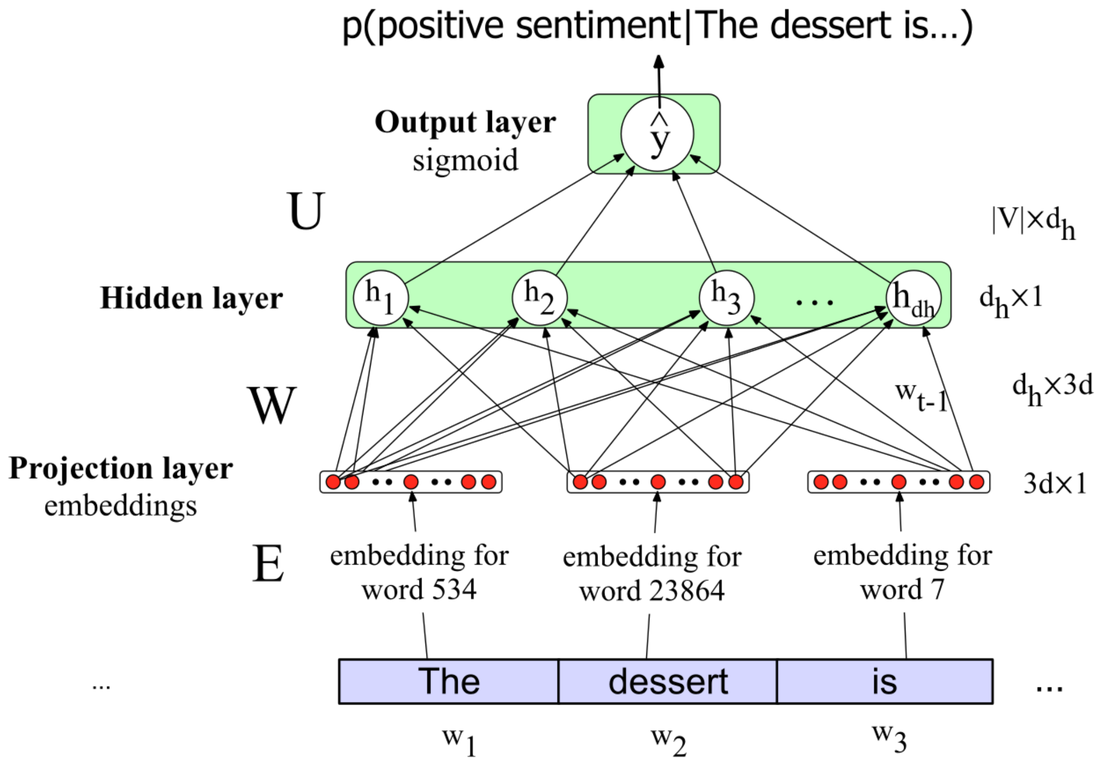



2-layer feedforward network



# Neural Networks for NLP

Real power of neural networks is utilized for NLP when …



# Neural Networks for NLP

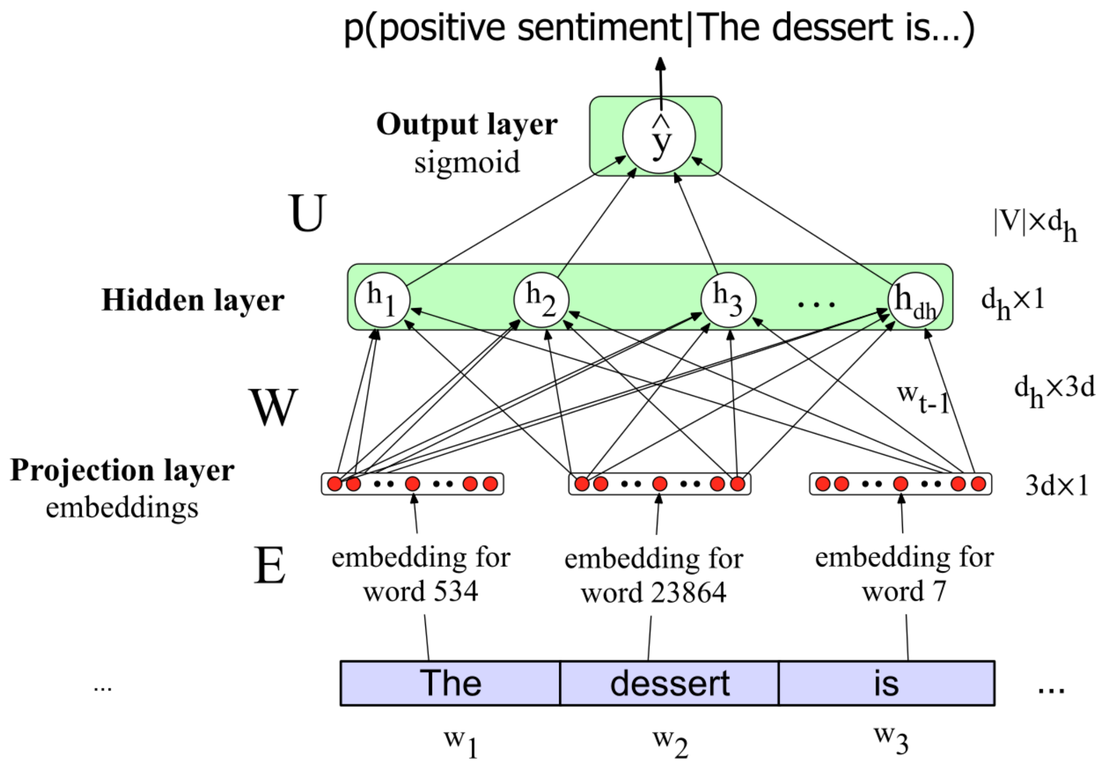
A **fixed size length** (3)

Some simple solutions:

1. Make the input the length of the longest review
   - If shorter then **pad** with zero embeddings
   - **Truncate** if you get longer reviews at test time

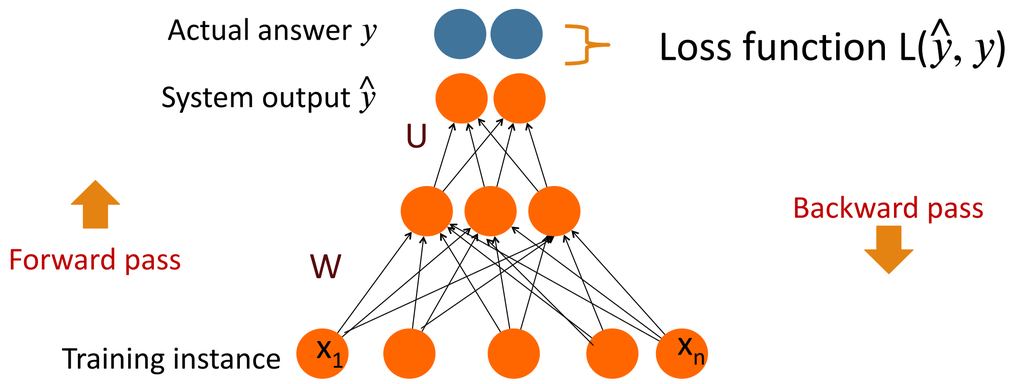
2. Create a single **sentence embedding** (the same dimensionality as word)

   a) Take the mean of all word embeddings
   b) Take the element-wise max of all the word embeddings.



# Neural Networks for NLP

How to train neural networks?




For every training tuple $(x, y)$

- Run *forward* computation to find our estimate $\hat{y}$
- Run *backward* computation to update weights:

For every output node
- Compute loss $L$ between $y$ and $\hat{y}$
- Update every weight $w$ in U

For every hidden node
- Assess how much blame it deserves for $\hat{y}$
- Update every weight $w$ in W

# Neural Networks for NLP

Cross-entropy Loss Function

$$L_{\text{CE}}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

$$= -[y \log \sigma(w \cdot x + b) + (1-y)\log(1-\sigma(w \cdot x + b))]$$

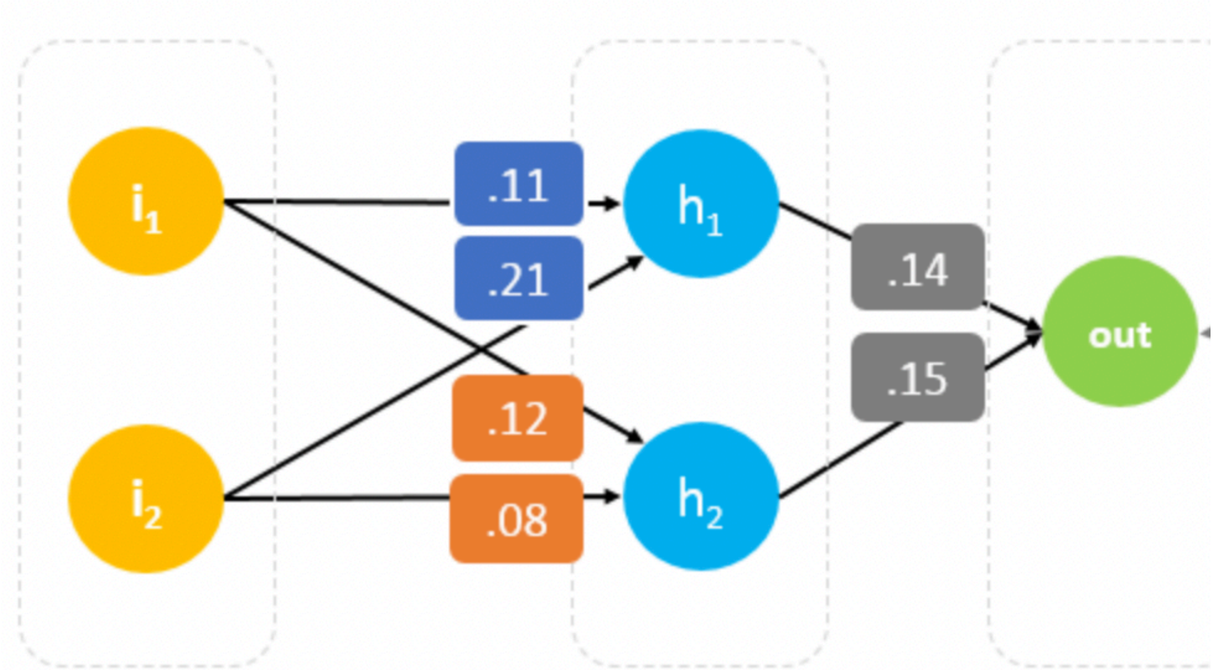
*Use the derivative of the loss function with respect to weights* $\frac{d}{dw} L(f(x; w), y)$

*Move them in the opposite direction of the gradient* $w^{t+1} = w^t - h \frac{d}{dw} L(f(x; w), y)$

*Apply chain rule and error backpropagation* $$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i}$$

# Neural Networks

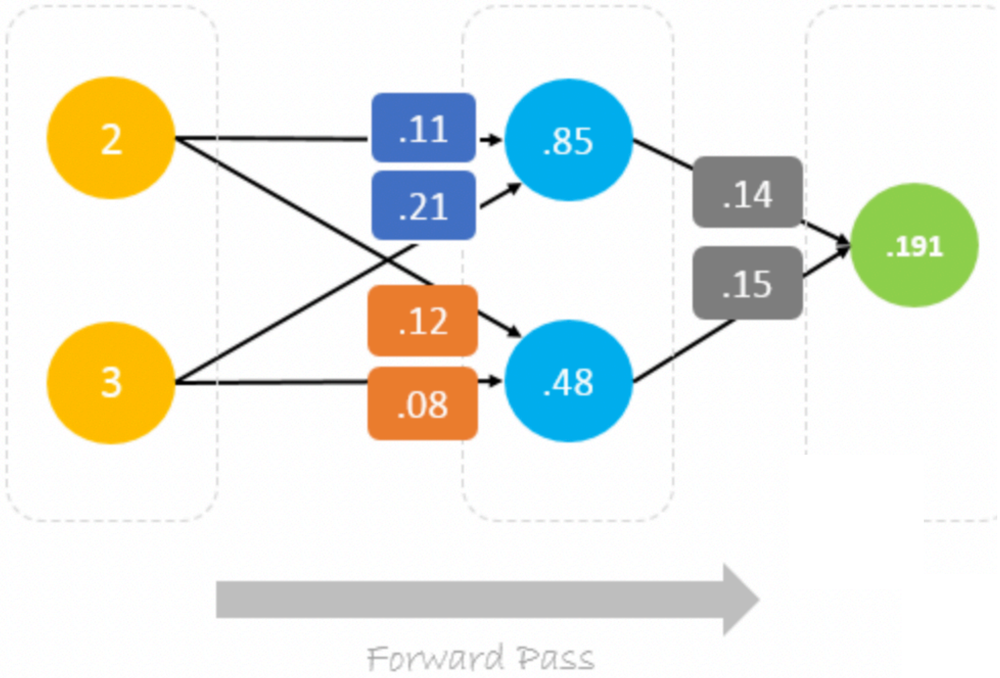
A toy example for training a single step of feed-forward neural network by backpropagation:




*Illustrations taken from: https://hmkcode.com/ai/backpropagation-step-by-step/

# Neural Networks

A toy example for training a single step of feed-forward neural network by backpropagation:
Train data: [2, 3] with label 1

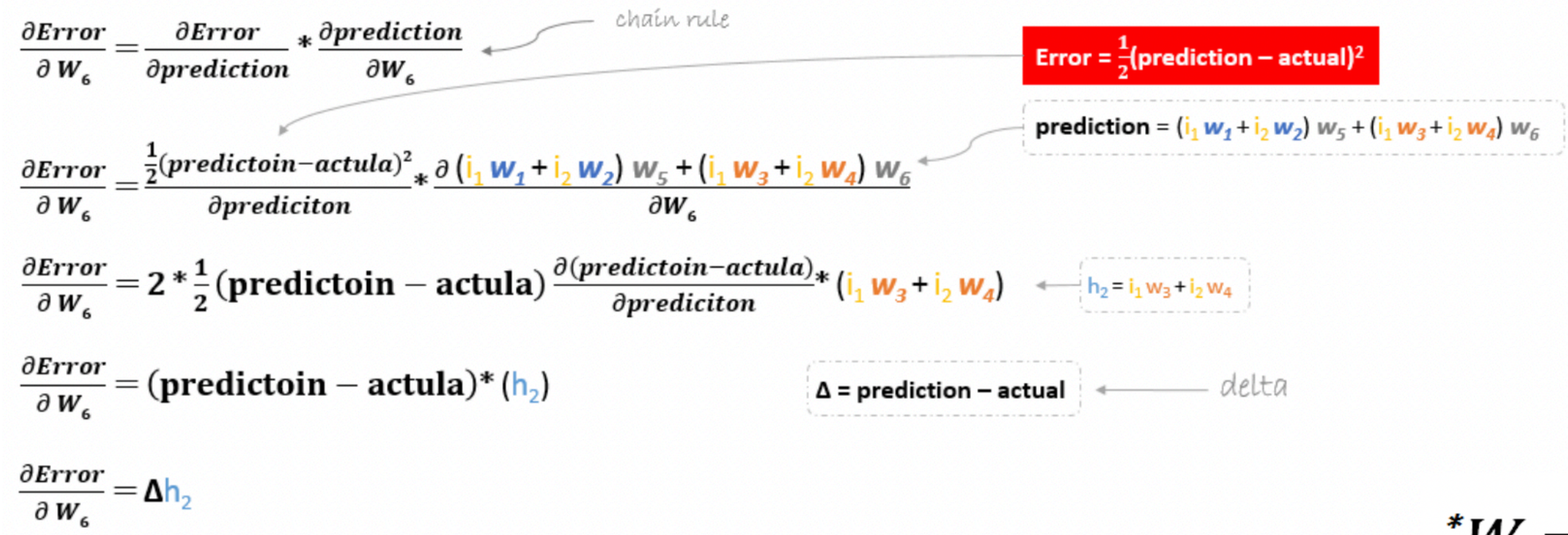



$$[2 \quad 3] \cdot \begin{bmatrix} 0.11 & 0.12 \\ 0.21 & 0.08 \end{bmatrix} = [0.85 \quad 0.48] \cdot \begin{bmatrix} 0.14 \\ 0.15 \end{bmatrix} = [0.191]$$

$$\text{Error} = \frac{1}{2}(0.191 - 1.0)^2 = 0.327$$

# Neural Networks

A toy example for training a single step of feed-forward neural network by backpropagation:
Train data: [2, 3] with label 1




$$\frac{\partial Error}{\partial W_6} = \frac{\partial Error}{\partial prediction} * \frac{\partial prediction}{\partial W_6}$$

$$\frac{\partial Error}{\partial W_6} = \frac{\frac{1}{2}(predictoin - actula)^2}{\partial prediciton} * \frac{\partial\,(i_1 w_1 + i_2 w_2)\, w_5 + (i_1 w_3 + i_2 w_4)\, w_6}{\partial W_6}$$

$$\frac{\partial Error}{\partial W_6} = 2 * \frac{1}{2}(predictoin - actula)\frac{\partial(predictoin - actula)}{\partial prediciton} * (i_1 w_3 + i_2 w_4)$$

$$\frac{\partial Error}{\partial W_6} = (predictoin - actula) * (h_2)$$

$$\frac{\partial Error}{\partial W_6} = \Delta h_2$$

$$^*W_6 = W_6 - a\,\Delta h_2$$

# Neural Networks

A toy example for training a single step of feed-forward neural network by backpropagation:
Train data: [2, 3] with label 1

$$
\begin{aligned}
{}^{*}w_6 &= w_6 - a\,(h_2 \,.\, \Delta) \\
{}^{*}w_5 &= w_5 - a\,(h_1 \,.\, \Delta) \\
{}^{*}w_4 &= w_4 - a\,(i_2 \,.\, \Delta w_6) \\
{}^{*}w_3 &= w_3 - a\,(i_1 \,.\, \Delta w_6) \\
{}^{*}w_2 &= w_2 - a\,(i_2 \,.\, \Delta w_5) \\
{}^{*}w_1 &= w_1 - a\,(i_1 \,.\, \Delta w_5)
\end{aligned}
$$

# Neural Networks

A toy example for training a single step of feed-forward neural network by backpropagation:
Train data: [2, 3] with label 1

$$\begin{bmatrix} w_5 \\ w_6 \end{bmatrix} = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix} - a\,\Delta \begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix} - \begin{bmatrix} a h_1 \Delta \\ a h_2 \Delta \end{bmatrix}$$

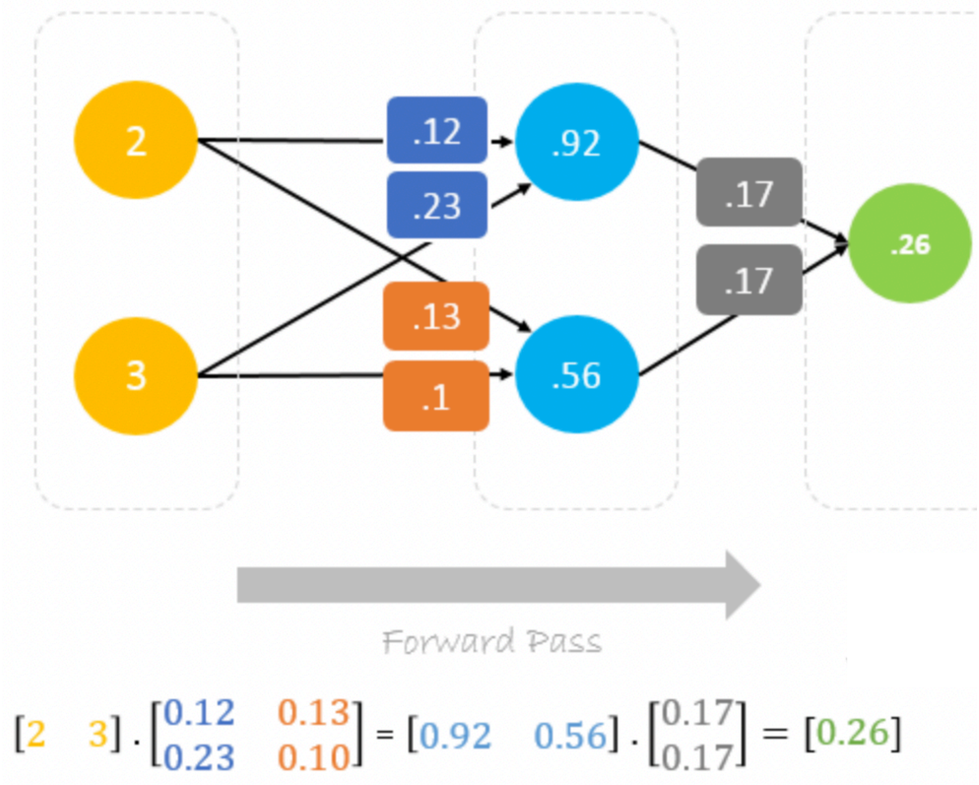
$$\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} - a\,\Delta \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} . [w_5 \quad w_6] = \begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} - \begin{bmatrix} a\, i_1 \Delta w_5 & a\, i_1 \Delta w_6 \\ a\, i_2 \Delta w_5 & a\, i_2 \Delta w_6 \end{bmatrix}$$

$$\begin{bmatrix} w_5 \\ w_6 \end{bmatrix} = \begin{bmatrix} 0.14 \\ 0.15 \end{bmatrix} - 0.05(-0.809) \begin{bmatrix} 0.85 \\ 0.48 \end{bmatrix} = \begin{bmatrix} 0.14 \\ 0.15 \end{bmatrix} - \begin{bmatrix} -0.034 \\ -0.019 \end{bmatrix} = \begin{bmatrix} 0.17 \\ 0.17 \end{bmatrix}$$

$$\begin{bmatrix} w_1 & w_3 \\ w_2 & w_4 \end{bmatrix} = \begin{bmatrix} .11 & .12 \\ .21 & .08 \end{bmatrix} - 0.05(-0.809) \begin{bmatrix} 2 \\ 3 \end{bmatrix} . [0.14 \quad 0.15] = \begin{bmatrix} .11 & .12 \\ .21 & .08 \end{bmatrix} - \begin{bmatrix} -0.011 & -0.012 \\ -0.017 & -0.018 \end{bmatrix} = \begin{bmatrix} .12 & .13 \\ .23 & .10 \end{bmatrix}$$

# Neural Networks

A toy example for training a single step of feed-forward neural network by backpropagation:
Train data: [2, 3] with label 1



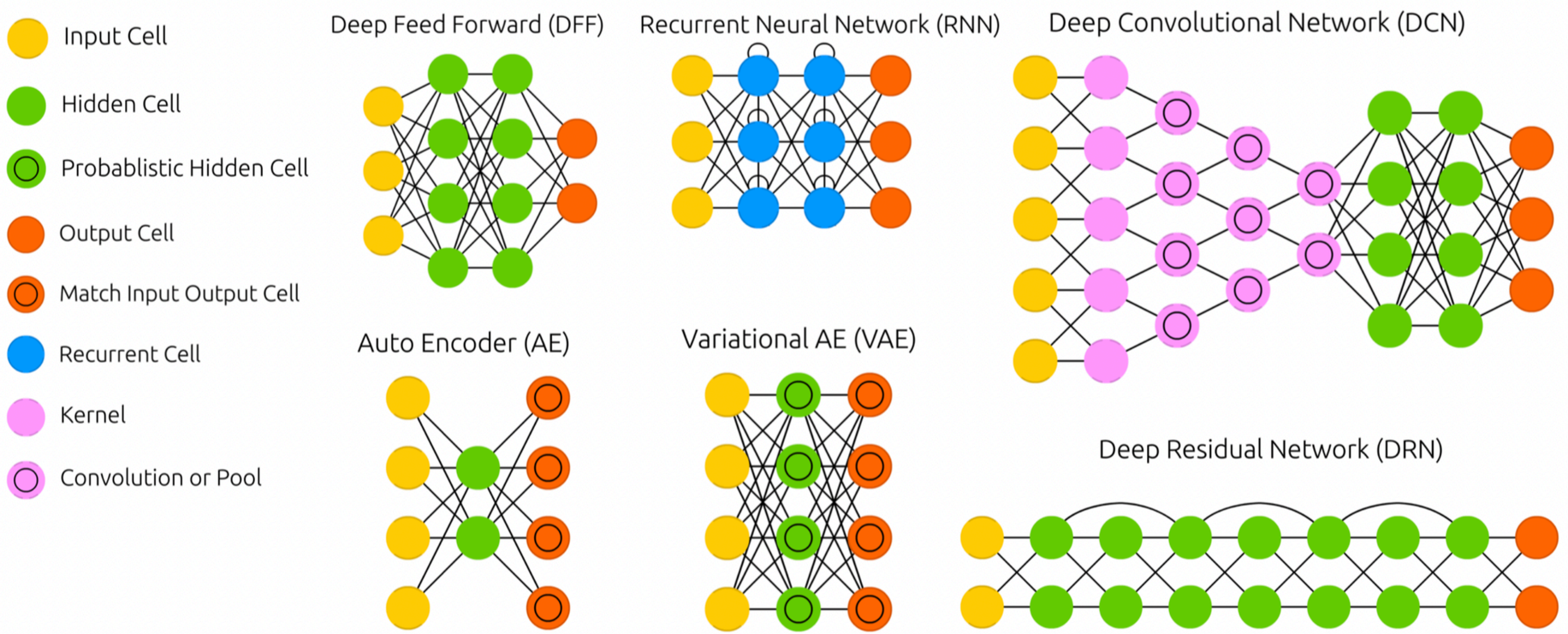
# Neural Networks

Further topics (take the Deep Learning course if you are interested):

- Advanced activation and loss functions
- Weight initialization
- Optimizers
- Momentum
- Weight decay
- Early stopping
- Hyperparameter tuning

# Neural Networks

- Other neural architectures: RNN, LSTM, GRU, CNN, etc.



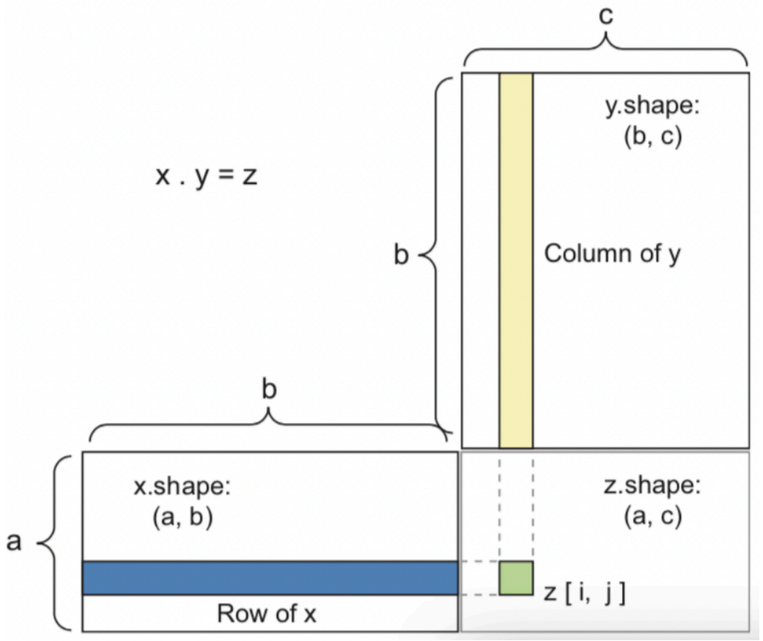
# Tensor Operations

Data operations on neural networks

Easier to run on GPUs

Example: A dense layer with sigmoid activation function (input X, weight W, bias b)

```
y = sigmoid(np.dot(X, W) + b)
```




$$\begin{bmatrix} A_1 & A_2 & A_3 \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ B_3 \end{bmatrix} = A_1B_1 + A_2B_2 + A_3B_3 = \vec{A}.\vec{B}$$

# Recurrent Neural Networks

Human thoughts have persistence; humans don’t start their thinking from scratch every second.

- As you read this sentence, you understand each word based on your prior knowledge.

Recurrent Neural Networks (RNNs) are a family of neural networks introduced to **learn sequential data**.

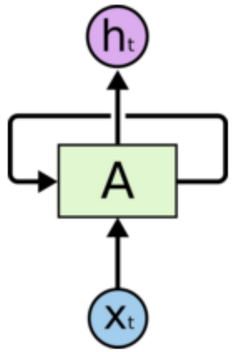
# Recurrent Neural Networks

- RNNs can be applied to various type of sequential data to learn the temporal patterns.
  - Time-series data (e.g., stock price)
  - Raw sensor data (e.g., signal, voice, handwriting)
  - Text to label (e.g., sentiment) or text sequence (e.g., translation, summary, answer)
  - Image and video to text description (e.g., captions, scene interpretation)

| **Task** | **Input** | **Output** |
|---|---|---|
| **Activity Recognition (Zhu et al. 2018)** | Sensor Signals | Activity Labels |
| **Machine translation (Sutskever et al. 2014)** | English text | French text |
| **Question answering (Bordes et al. 2014)** | Question | Answer |
| **Speech recognition (Graves et al. 2013)** | Voice | Text |
| **Handwriting prediction (Graves 2013)** | Handwriting | Text |
| **Opinion mining (Irsoy et al. 2014)** | Text | Opinion expression |

# Recurrent Neural Networks

RNNs are networks with loops, allowing information to persist.




Output is to predict a vector $h_t$, where $output\, y_t = \varphi(h_t)$ at some time steps ($t$)

Recurrent Neural Networks have loops.

In the above diagram, a chunk of neural network, ***A = $f_W$,*** looks at some input ***xt*** and outputs a value ***ht***. A loop allows information to be passed from one step of the network to the next.

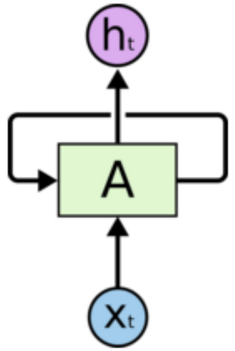
new state — old state

$$h_t = f_W(h_{t-1}, x_t)$$

function with parameter $W$ — Input vector at some time step

# Recurrent Neural Networks

RNNs are networks with loops, allowing information to persist.




Recurrent Neural Networks have loops.

In the above diagram, a chunk of neural network, ***A = $f_W$,*** looks at some input ***xt*** and outputs a value ***ht***. A loop allows information to be passed from one step of the network to the next.

$$h_t = f_W(h_{t-1}, x_t)$$

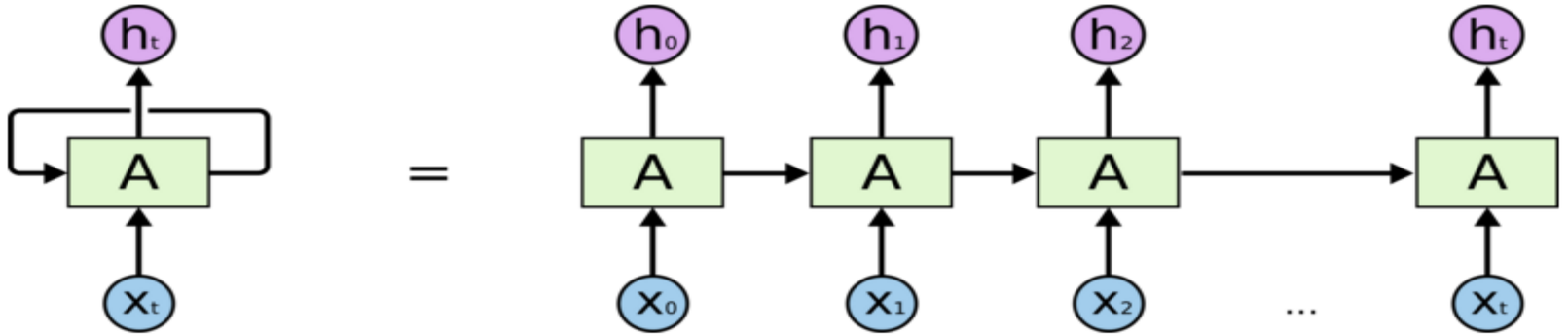
$$\downarrow$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

# Recurrent Neural Networks

RNN can be thought of as multiple copies of the same network, each passing a message to a successor.



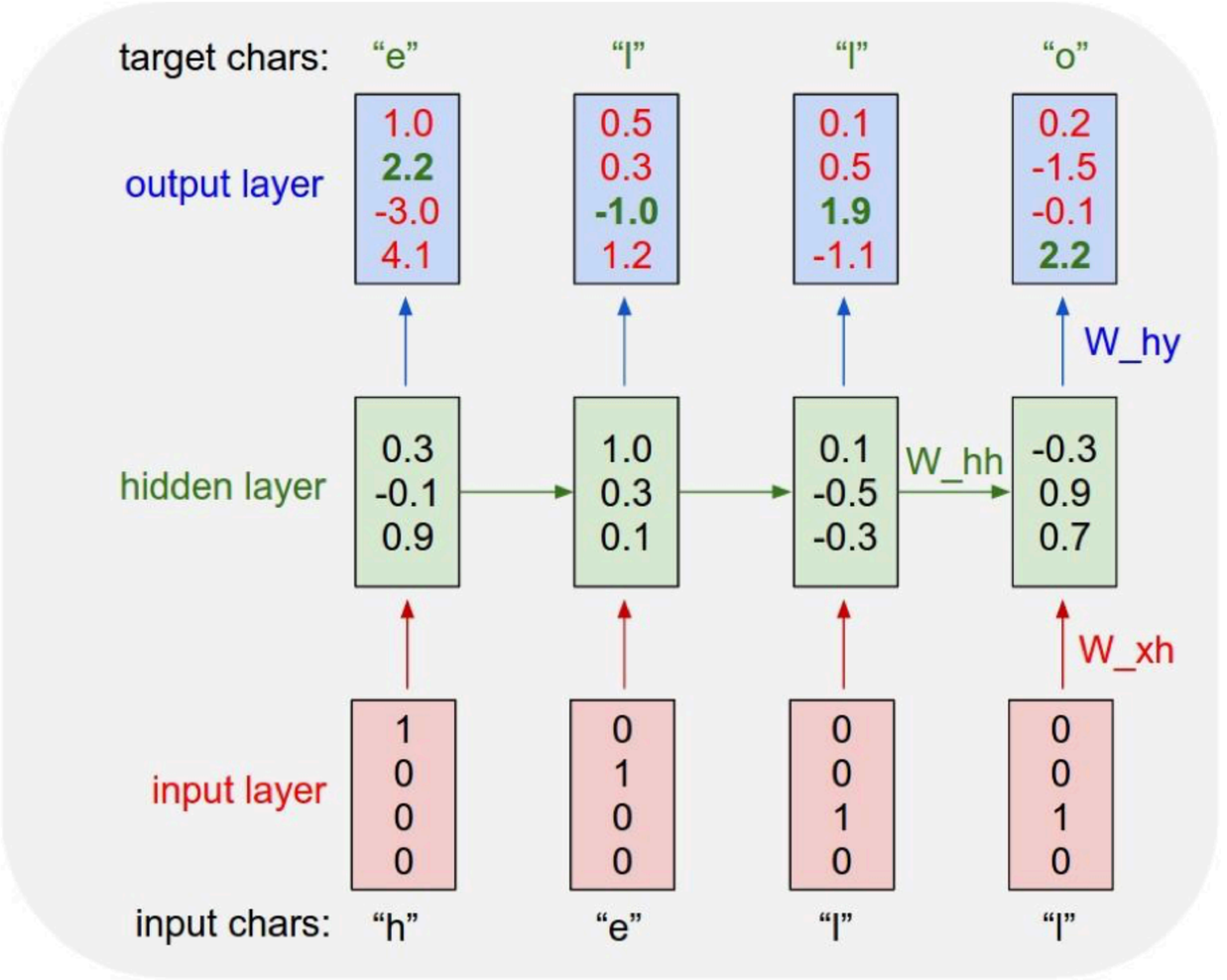

An unrolled recurrent neural network.

# Recurrent Neural Networks

**Example: Character-level Language Model**

Vocabulary:
[h,e,l,o]

Example training sequence:
**“hello”**



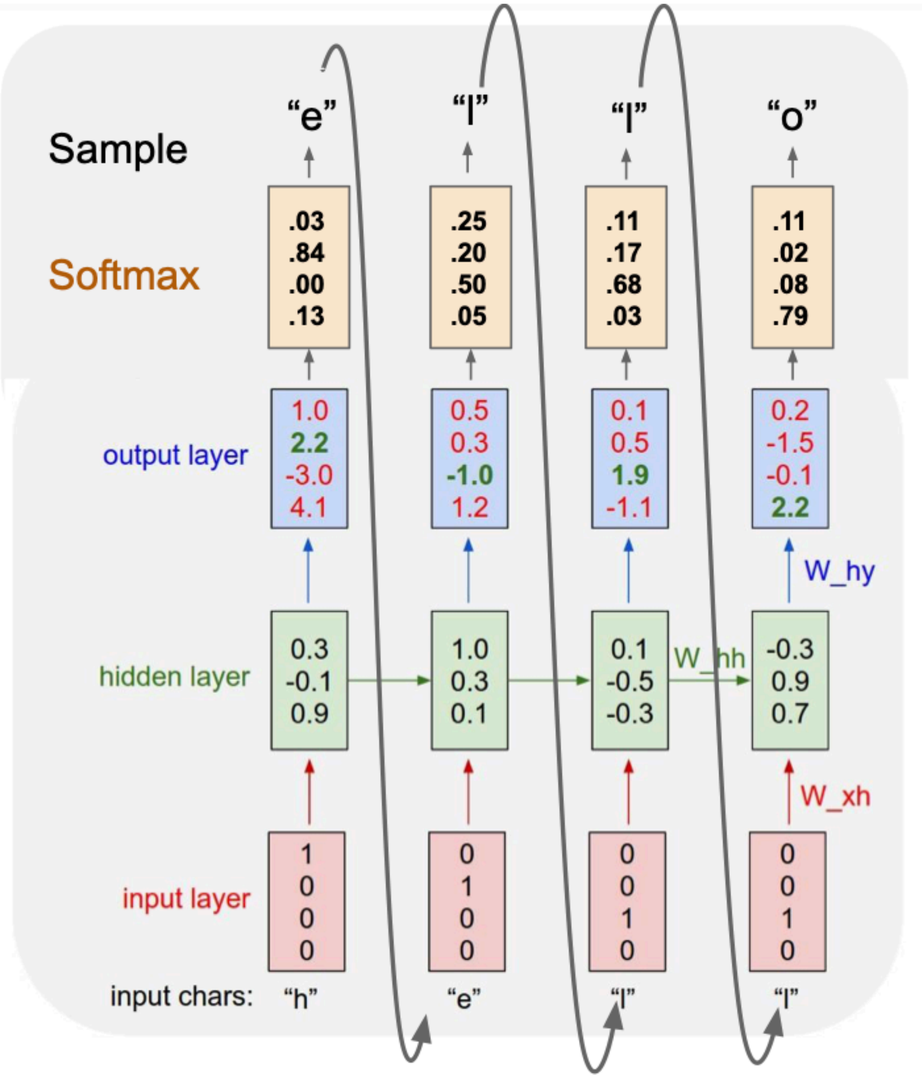

*Illustrations taken from Stanford University CS231

# Recurrent Neural Networks

**Example: Character-level Language Model Sampling**

Vocabulary:
[h,e,l,o]

At test-time sample characters one at a time, feed back to model




*Illustrations taken from Stanford University CS231

# Recurrent Neural Networks

Main advantages of recurrent structure:

Specialized for processing a sequence of values $x^{(1)}, \ldots, x^{(\tau)}$

Each value $x^{(i)}$ is processed with the **same network *A* that preserves past information**

Can scale to much **longer sequences** than would be practical for networks without a recurrent structure

Reusing network ***A*** reduces the required amount of parameters in the network

Can process **variable-length sequences**

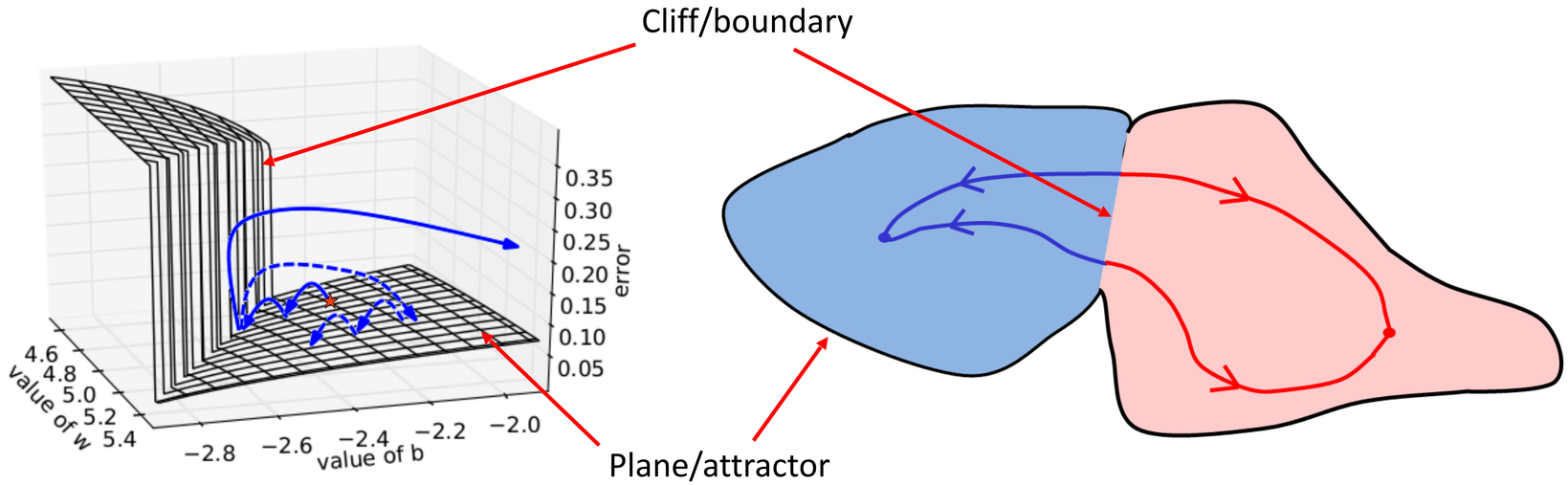
The network complexity does not vary when the input length change

# Recurrent Neural Networks

Main disadvantages of recurrent structure:

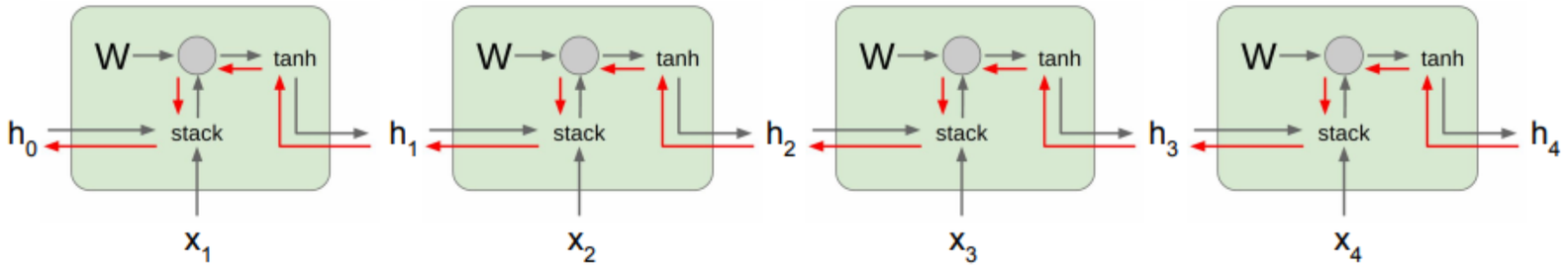
Vanilla RNNs suffer from the training difficulty due to **exploding and vanishing gradients**.

# Recurrent Neural Networks




- **Exploding:** If we start almost exactly on the boundary (cliff), tiny changes can make a huge difference.
- **Vanishing:** If we start a trajectory within an attractor (plane, flat surface), small changes in where we start make no difference to where we end up.
- **Both cases** hinder the learning process.

# Recurrent Neural Networks




In vanilla RNNs, computing this gradient involves many factors of $W_{hh}$
If we decompose the singular values of the gradient multiplication matrix:

- Largest singular value > 1 **Exploding gradients**

  Slight error in the late time steps causes drastic updates in the early time steps (Unstable learning)

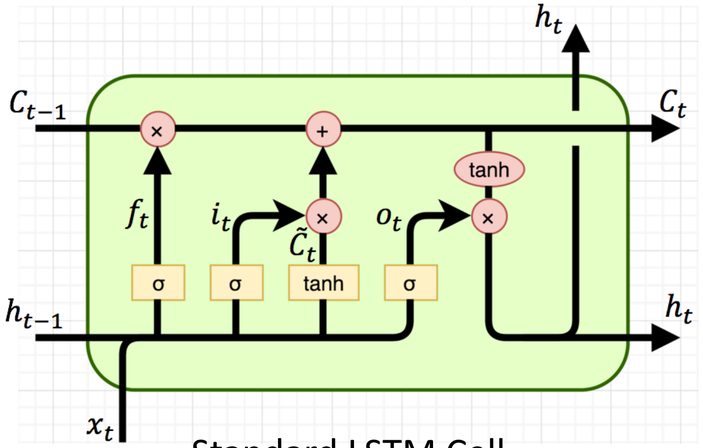
- Largest singular value < 1 **Vanishing gradients**

  Gradients passed to the early time steps is close to 0 (Uninformed correction)

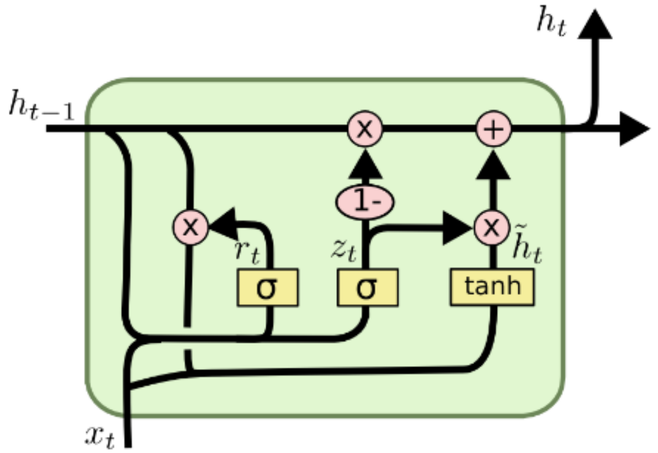
# Recurrent Neural Networks (with Memory)

Two recurrent cell designs were proposed and widely adopted:

**Long Short-Term Memory (LSTM)** (Hochreiter and Schmidhuber, 1997)
**Gated Recurrent Unit (GRU)** (Cho et al. 2014)




Standard LSTM Cell



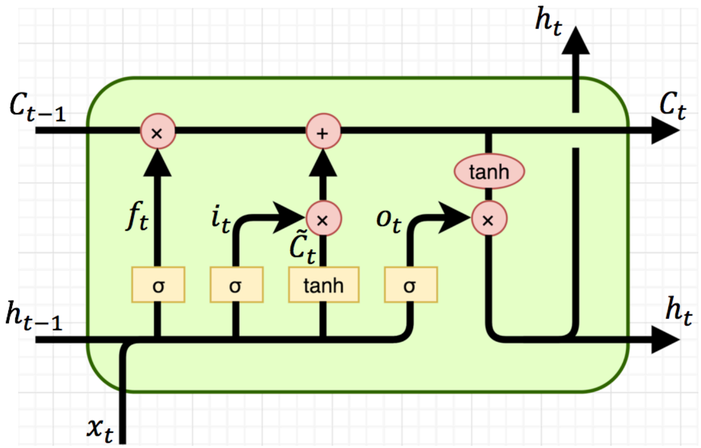

GRU Cell

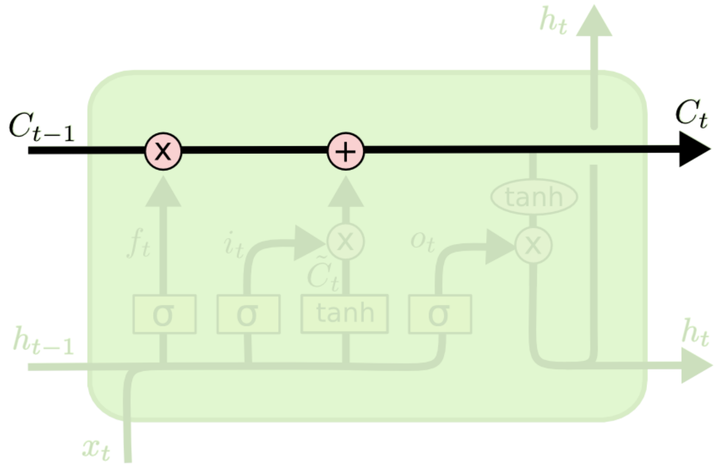
*Both designs process information in an additive way with gates to control information flow.*

# Recurrent Neural Networks (with Memory)

The key to LSTMs is the **cell state**

Stores information of the past: **long-term memory**



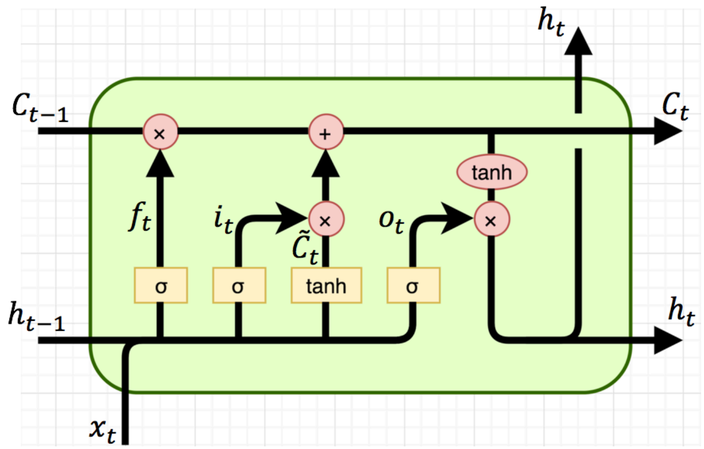

Standard LSTM Cell

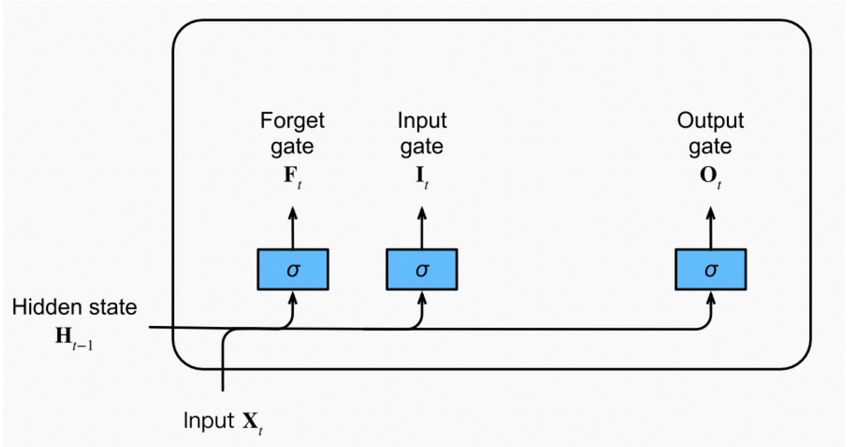
# Recurrent Neural Networks (with Memory)

**Input gate:** Controls the intake of new information

**Forget gate**: Determines what part of the cell state to be updated

**Output gate**: Determines what part of the cell state to output



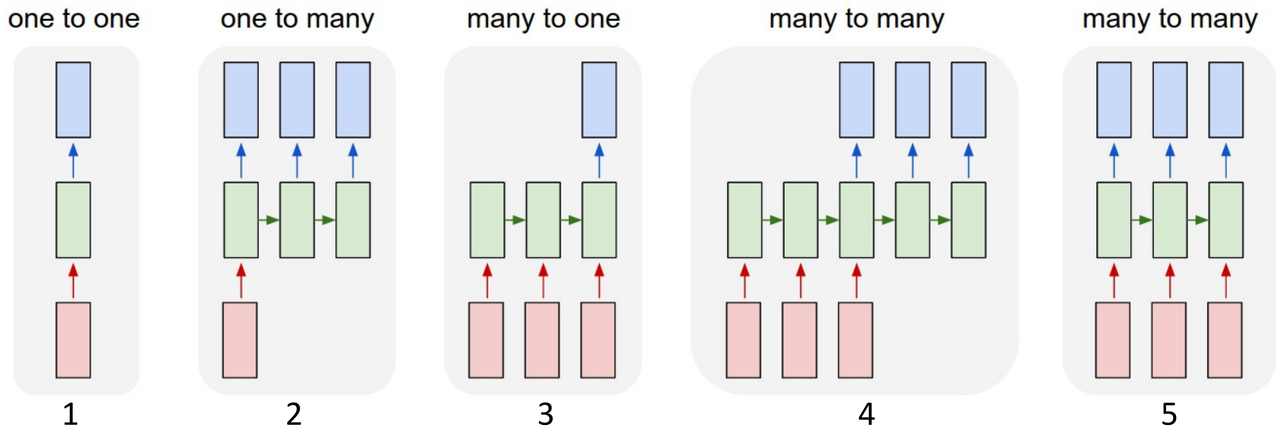




Standard LSTM Cell

# Recurrent Neural Networks (Multiple Layers)

- Learning on RNN is more robust when the vanishing/exploding gradient problem is resolved
- RNNs can now be applied to different Sequence Learning tasks
- RNN is flexible to operate over various sequences of vectors
- Sequence in the input, the output, or in the most general case both
- Architecture with one or more RNN layers

# Recurrent Neural Networks (Multiple Layers)




- Each rectangle is a vector
- Arrows represent functions (e.g. matrix multiply)
- Input in red, output in blue, RNN state in green

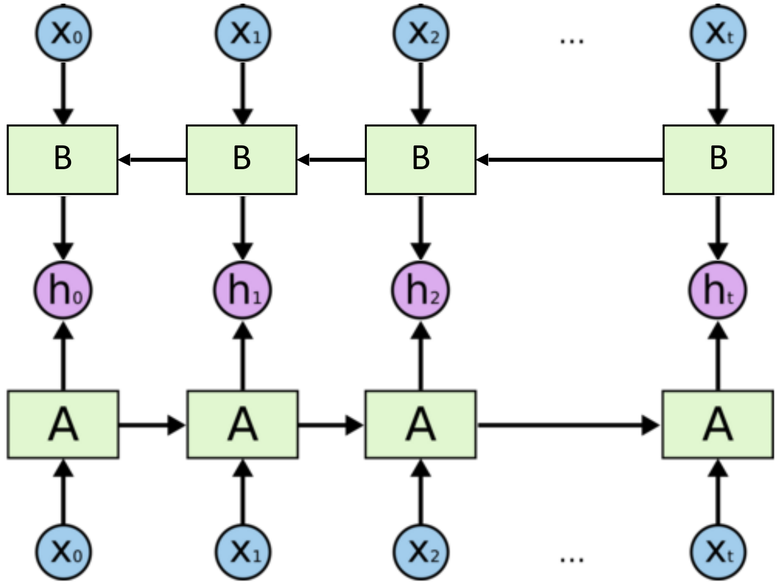
(1) Standard NN mode without recurrent structure (e.g. **image classification, one label for one image**)
(2) Sequence output (e.g. **image captioning, takes an image and outputs a sentence of words**)
(3) Sequence input (e.g. **sentiment analysis, a sentence is classified as expressing positive or negative sentiment**)
(4) Sequence input and sequence output (e.g. **machine translation, a sentence in English is translated into a sentence in French**)
(5) Synced sequence input and output (e.g. **video classification, label each frame of the video**)

# Recurrent Neural Networks (Multiple Layers)

**Bidirectional RNN**

- Connects two recurrent units of opposite directions to the same output
- Captures forward and backward information from the input sequence
- Apply to data whose current state (e.g., $h_0$) can be better determined when given future information (e.g., $x_1, x_2, \ldots, x_t$)

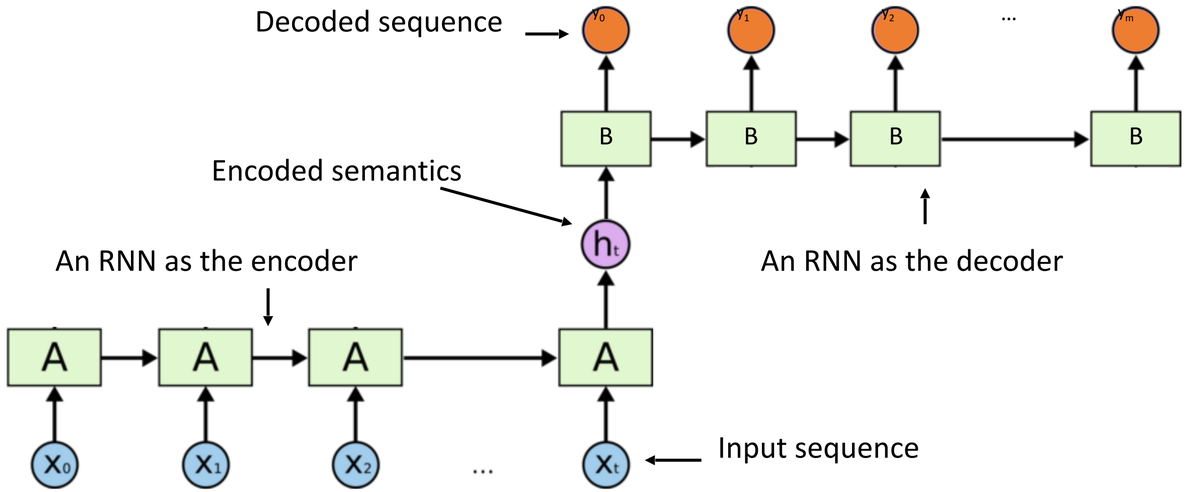
  E.g. “the bank is robbed,” the semantics of “bank” can be determined given the verb “robbed.”



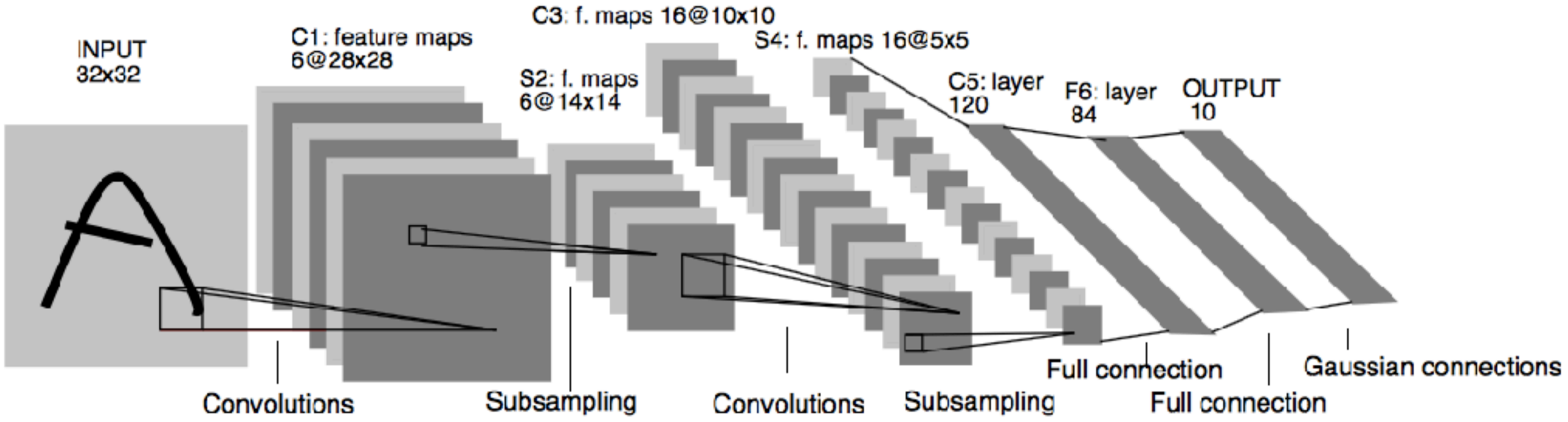
# Recurrent Neural Networks (Multiple Layers)

- **Sequence-to-Sequence (Seq2Seq) model**
  - Developed by Google in 2018 for use in machine translation.
  - Seq2seq turns one sequence into another sequence.

  - **Encoder RNN**: extract and compress the semantics from the input sequence
  - **Decoder RNN**: generate a sequence based on the input semantics
  - Apply to tasks such as machine translation
    - Similar underlying semantics
    - E.g., “I love you.” to “Je t’aime.”

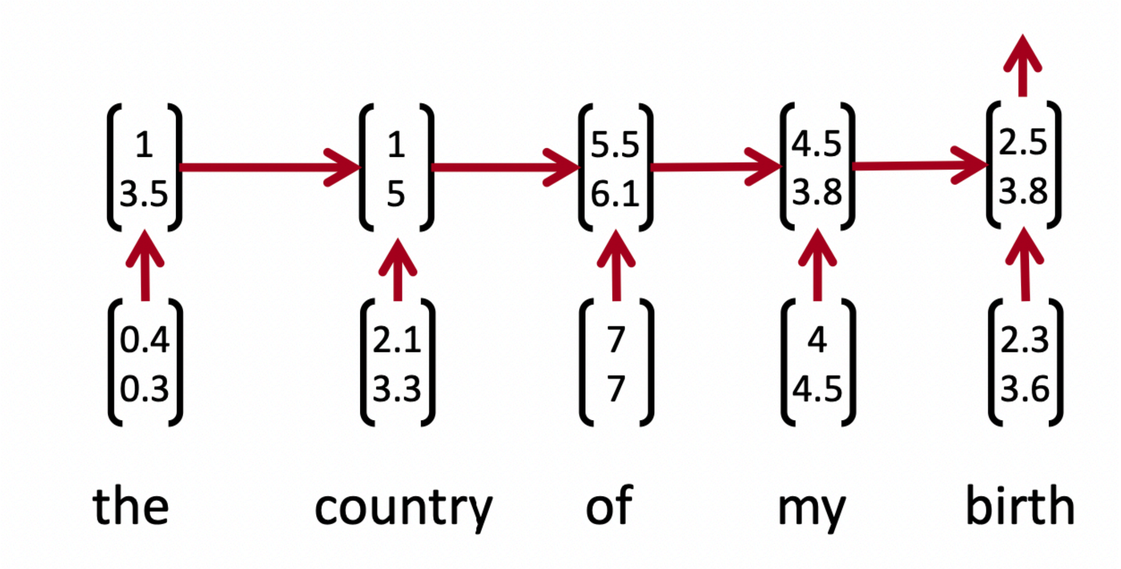


# Convolutional Neural Networks*




*LeCun et al. 1997

# Convolutional Neural Networks*

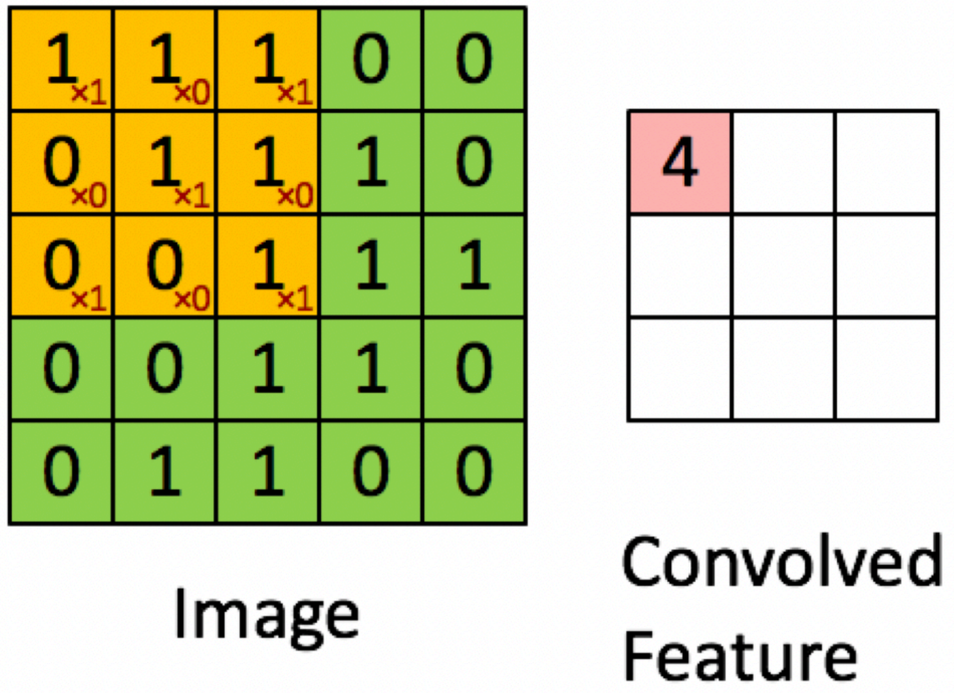



**Slides adapted from Stanford CS224N*

# Convolutional Neural Networks

A Convolutional Layer

$$(f * g)[n] = \sum_{m=-M}^{M} f[n-m]g[m].$$



# Convolutional Neural Networks for NLP

What if we compute vectors for every possible word subsequence of a certain length?

Example: “tentative deal reached to keep government open”

tentative deal reached
deal reached to
reached to keep
to keep government
keep government open

# Convolutional Neural Networks for NLP

| | | | | |
|---|---|---|---|---|
| tentative | 0.2 | 0.1 | −0.3 | 0.4 |
| deal | 0.5 | 0.2 | −0.3 | −0.1 |
| reached | −0.1 | −0.3 | −0.2 | 0.4 |
| to | 0.3 | −0.3 | 0.1 | 0.1 |
| keep | 0.2 | −0.3 | 0.4 | 0.2 |
| government | 0.1 | 0.2 | −0.1 | −0.1 |
| open | −0.4 | −0.4 | 0.2 | 0.3 |

| | |
|---|---|
| t,d,r | −1.0 |
| d,r,t | −0.5 |
| r,t,k | −3.6 |
| t,k,g | −0.2 |
| k,g,o | 0.3 |

Apply a **filter** (or **kernel**) of size 3

| | | | |
|---|---|---|---|
| 3 | 1 | 2 | −3 |
| −1 | 2 | 1 | −3 |
| 1 | 1 | −1 | 1 |

# Convolutional Neural Networks for NLP

| Ø | 0.0 | 0.0 | 0.0 | 0.0 |
|---|---|---|---|---|
| tentative | 0.2 | 0.1 | −0.3 | 0.4 |
| deal | 0.5 | 0.2 | −0.3 | −0.1 |
| reached | −0.1 | −0.3 | −0.2 | 0.4 |
| to | 0.3 | −0.3 | 0.1 | 0.1 |
| keep | 0.2 | −0.3 | 0.4 | 0.2 |
| government | 0.1 | 0.2 | −0.1 | −0.1 |
| open | −0.4 | −0.4 | 0.2 | 0.3 |
| Ø | 0.0 | 0.0 | 0.0 | 0.0 |

| Ø,t,d | −0.6 | 0.2 | 1.4 |
|---|---|---|---|
| t,d,r | −1.0 | 1.6 | −1.0 |
| d,r,t | −0.5 | −0.1 | 0.8 |
| r,t,k | −3.6 | 0.3 | 0.3 |
| t,k,g | −0.2 | 0.1 | 1.2 |
| k,g,o | 0.3 | 0.6 | 0.9 |
| g,o,Ø | −0.5 | −0.9 | 0.1 |

| max p | 0.3 | 1.6 | 1.4 |
|---|---|---|---|

Apply 3 **filters** of size 3

| 3 | 1 | 2 | −3 |
|---|---|---|---|
| −1 | 2 | 1 | −3 |
| 1 | 1 | −1 | 1 |

| 1 | 0 | 0 | 1 |
|---|---|---|---|
| 1 | 0 | −1 | −1 |
| 0 | 1 | 0 | 1 |

| 1 | −1 | 2 | −1 |
|---|---|---|---|
| 1 | 0 | −1 | 3 |
| 0 | 2 | 2 | 1 |

# Convolutional Neural Networks for NLP

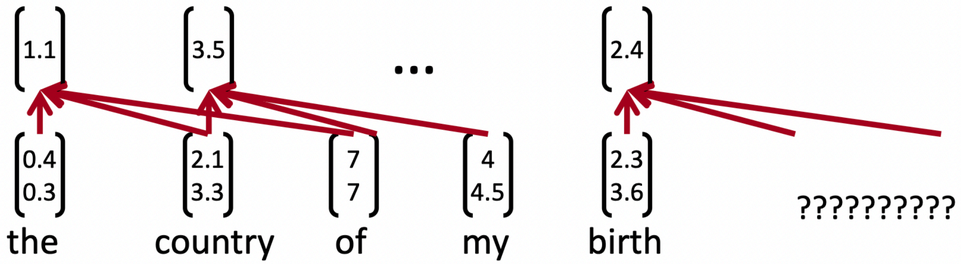
```
batch_size = 16
word_embed_size = 4
seq_len = 7
input = torch.randn(batch_size, word_embed_size, seq_len)
conv1 = Conv1d(in_channels=word_embed_size, out_channels=3,
               kernel_size=3) # can add: padding=1
hidden1 = conv1(input)
hidden2 = torch.max(hidden1, dim=2) # max pool
```

# Convolutional Neural Networks for NLP (TextCNN)

- Filter **w** is applied to all possible windows (concatenated vectors)
- To compute feature (one *channel*) for CNN layer:

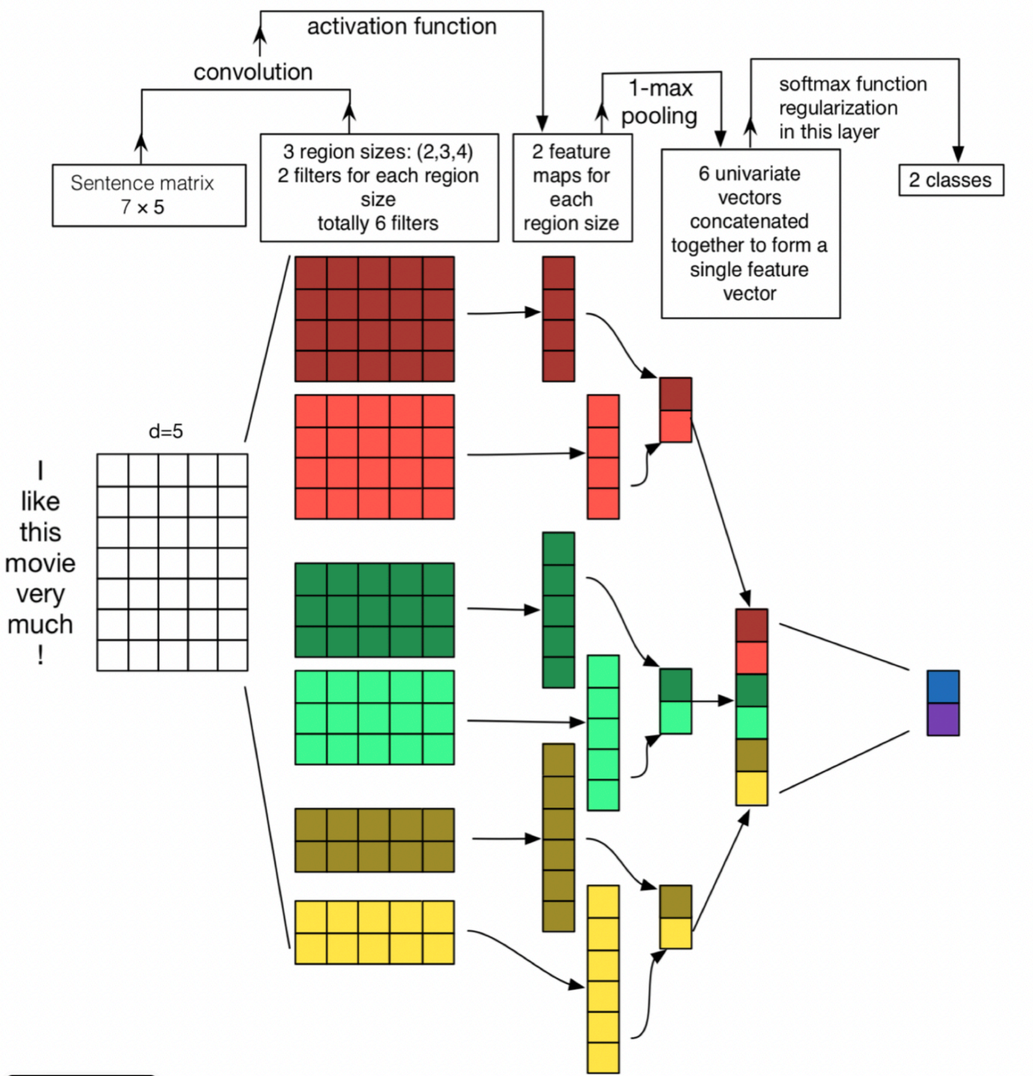
$$c_i = f\left(\mathbf{w}^T \mathbf{x}_{i:i+h-1} + b\right)$$

- Sentence: $\mathbf{x}_{1:n} = \mathbf{x}_1 \oplus \mathbf{x}_2 \oplus \ldots \oplus \mathbf{x}_n$
- All possible windows of length *h*: $\{\mathbf{x}_{1:h}, \mathbf{x}_{2:h+1}, \ldots, \mathbf{x}_{n-h+1:n}\}$
- Result is a feature map: $\mathbf{c} = [c_1, c_2, \ldots, c_{n-h+1}] \in \mathbb{R}^{n-h+1}$




*Yoon Kim (2014): Convolutional Neural Networks for Sentence Classification. EMNLP 2014. https://arxiv.org/pdf/1408.5882.pdf*

# Convolutional Neural Networks for NLP (TextCNN)



ORTA DOĞU TEKNİK ÜNİVERSİTESİ
MIDDLE EAST TECHNICAL UNIVERSITY

# Thanks for your participation!

**Çağrı Toraman**
**11.11.2025**