Master of Data Science and Innovation

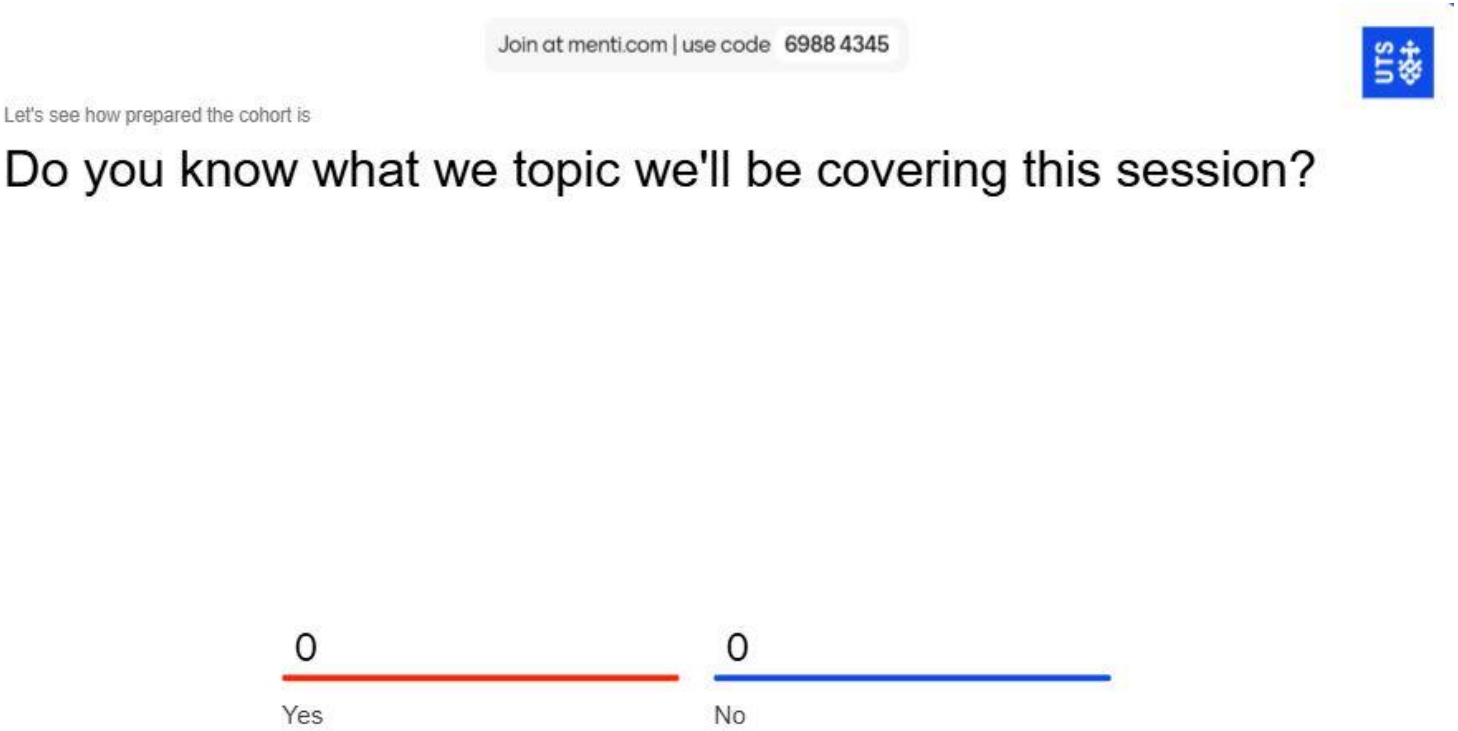
36118 Applied Natural Language Processing (ANLP)

Session 4

IMPORTANT !!!
While we wait for the session to start, please add your group details for AT2 on the Miro board;
Or add your details if you are still looking for group members

Dr. Antonette Shibani
Senior Lecturer, MDSI

Menti poll


Join at menti.com | use code 6988 4345
Let's see how prepared the cohort is
Do you know what we topic we'll be covering this session?
0
Yes
0
No

So far…

- Foundations of NLP (core concepts)
- NLP for text analysis, Python packages
- Text representation (Count, Tf-idf vectorizers)
- Unsupervised approaches (Clustering, Topic modelling)
- Supervised machine learning (Text classification)
- Sentiment analysis

Agenda for today's session

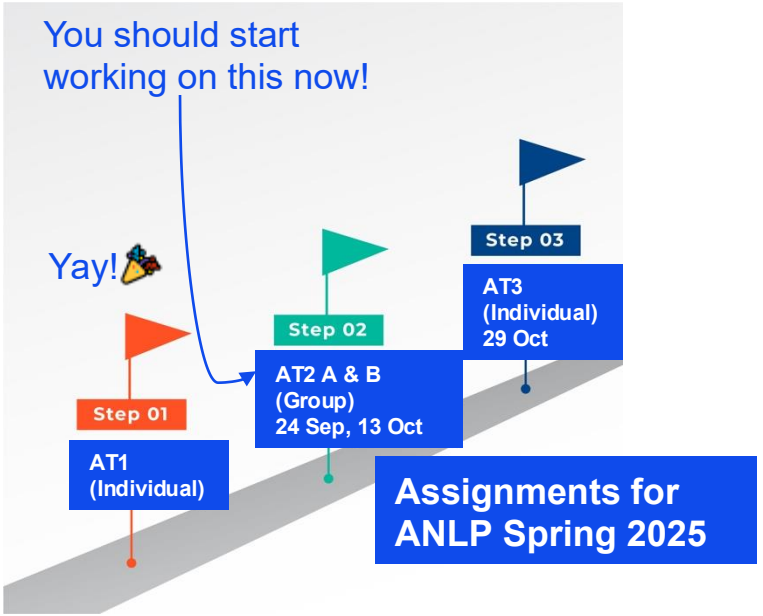
- Word embeddings and vector space representation
- Deep Learning Introduction
- CNNs
- RNN
- LSTM (Extended topic)

Update on Assignments

Well done on AT1 – You've crossed the first milestone!

Extension requests

-only in exceptional circumstances

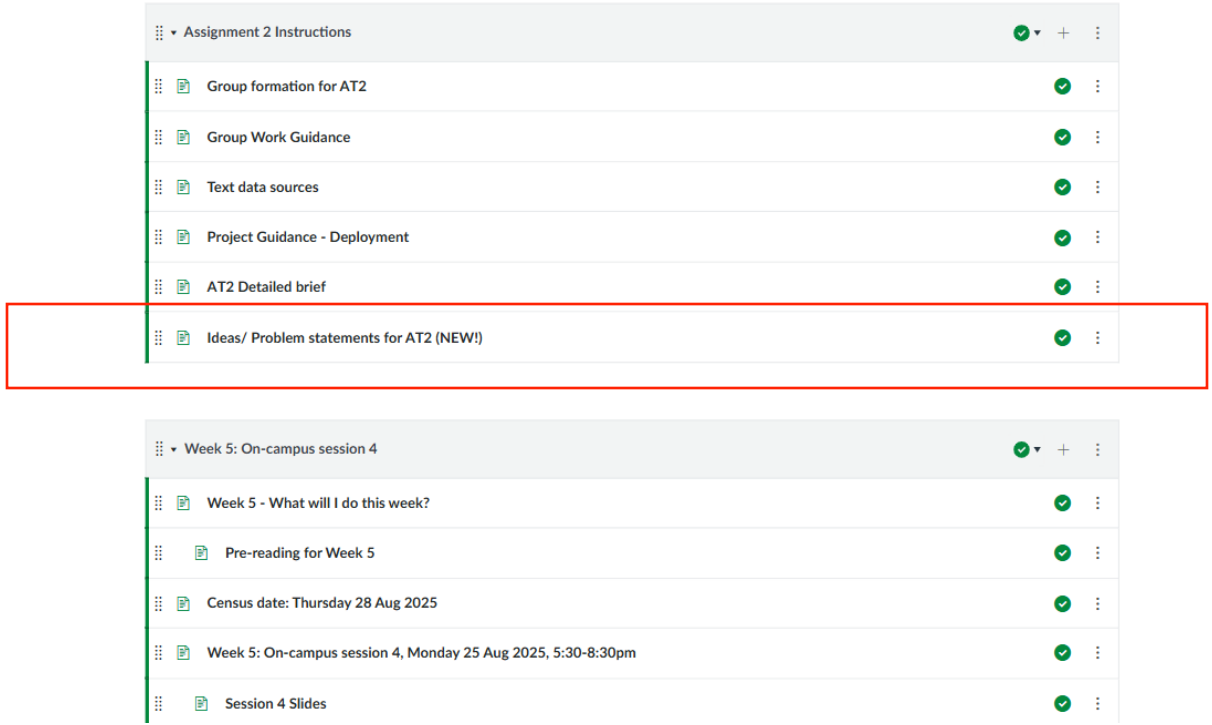


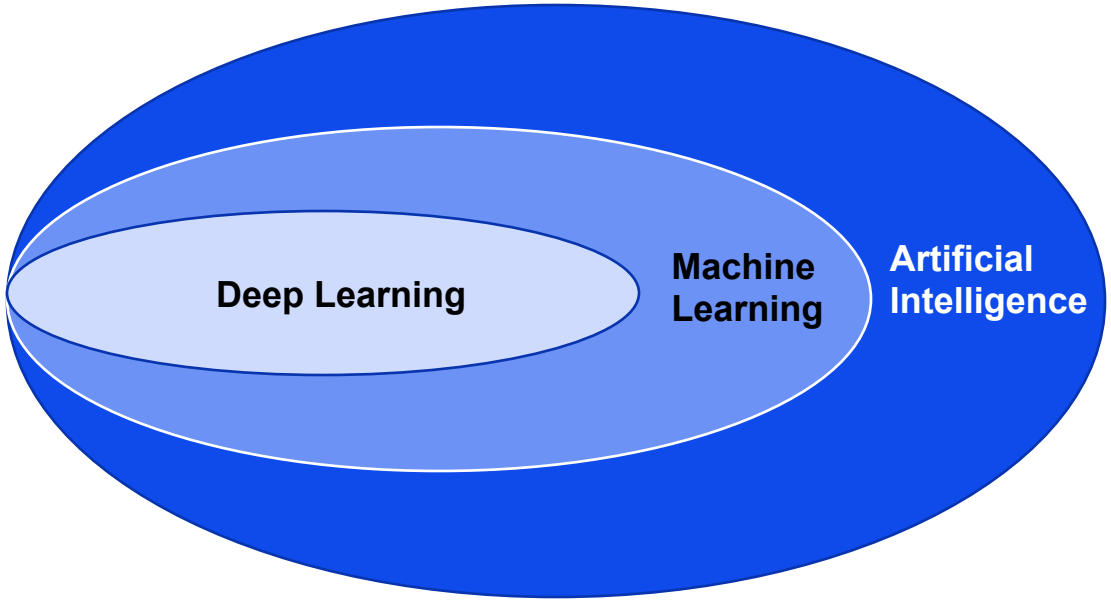
Credit points and estimated workload (university guidelines)

CP	Hours
2 (or 3)	50
6 (many UG subjects)	150
8 (many PG subjects)	200
12	300

Apart from the 27 hours spent in class, you should be doing a lot more self-directed work to achieve your learning objectives for the subject!

Check out this new page on Canvas for AT2 ideas for your team. Projects and teams should be finalised by the end of this week!




Deep Learning
Machine Learning
Artificial Intelligence

Applications

- Chat bots
- Conversational Assistants
- Question answering systems, Search
- Machine translation
- Sentiment analysis
- Text summarization
- Automated content generation
- ….

The vast majority of these are now powered by Deep Learning!

Vectorisation and Embeddings

Representing Words — Traditional NLP

- The methods we've seen so far for text representation deal with words as atomic entities – each word is encoded as a 1 or 0 in the DTM (or TF-IDF) matrix.
- We call this a "one-hot" representation. E.g.,

  shop is represented as [0 0 0 0 0 0 1 0 0 0]

  store is represented as [0 0 1 0 0 0 0 0 0 0]

Traditional representation

- In an **One-Hot Encoding**
  - Length of vector = size of vocabulary V (number of unique words)
  - Vector values: 1 if dimension reflects word, 0 otherwise

Note: Vector Space Model document vector = aggregation over word vectors (with some weighting, e.g., sum, tf-idf)

- Toy example: V = {dog, cat, lion, bear, cobra, cow, frog, ...}

	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	...	$w_{\|V\|}$
dog	**1**	0	0	0	0	0	0	0	0	...	0
cat	0	**1**	0	0	0	0	0	0	0	...	0
lion	0	0	**1**	0	0	0	0	0	0	...	0
bear	0	0	0	**1**	0	0	0	0	0	...	0
...	...	...	...	...	...	...	...	...	...	...	...

one-hot vector of *"cat"*

These create sparse, high-dimensional vectors!

Limitation of symbolic representations

*"I saw a **cat**."*

vs.

*"I saw a **kitty**."*

- ~4kg
- ~45cm long
- 4 legs
- long tail
- whiskers
- furry
- purrs
- eats mice
- common pet

cat = [0 0 0 0 … 0 0 0 0 1 0 0 0 0 0 0]

kitty = [0 0 1 0 0 0 0 0 0 0 0 … 0 0 0]

Orthogonal word vectors,
No semantic relationship

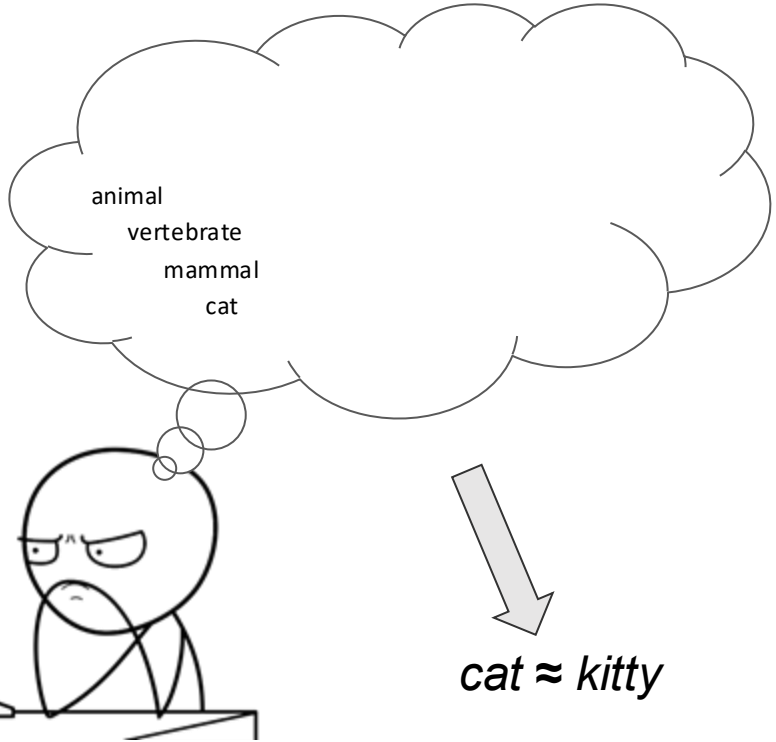
cat ≠ kitty

Limitation of symbolic representations

*"I saw a **cat**."*

vs.

*"I saw a **kitty**."*



cat = [0 0 0 0 … 0 0 0 0 1 0 0 0 0 0 0]

kitty = [0 0 1 0 0 0 0 0 0 0 0 … 0 0 0]

orthogonal
word vectors

cat ≠ *kitty*

cat ≈ *kitty*

Quiz
(How do we give the word a meaning?)

*You can only get **Stollen** around Christmas time.*

*It's expensive to buy **Stollen** in Australia.*

*We're planning to have **Stollen** during the afternoon break.*

*Some recipes for **Stollen** make it rather dry.*

What do you think "***Stollen***" is?

A a drink

B a food

C a drug

D a show

Quiz answer

B. a food

This is "***Stollen***"!

Source: Stollen (Wikipedia Commons)

Word meaning

Definition: **meaning** (Webster dictionary)

- the idea that is represented by a word, phrase, etc.
- the idea that a person wants to express by using words, signs, etc.

➔ We can define the meaning of a word by other words commonly surrounding it – encoding by a dense vector

there have never been any shark attacks at places like

juvenile white sharks used to area closer to the entrance

inhabited with far more great white sharks than previously thought

tagged great white sharks, some up to 3.2 metres long

only one shark being detected in the western section

…

shark = $\begin{pmatrix} 0.134 \\ -0.231 \\ 0.564 \\ 0.034 \\ -0.013 \\ 0.983 \\ \dots \end{pmatrix}$

Importance of Word Order

- Obvious for many tasks, e.g.:
  - Keyphrase extraction
  - Named entity recognition
  - Machine translation
  - Summarization
- Also important for basic tasks
  - Example: sentiment analysis

"I don't hate the movie, I like it."

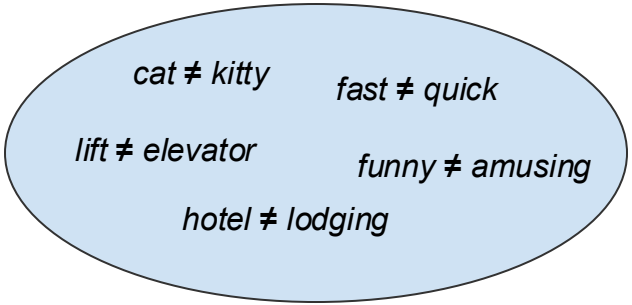
vs.

"I don't like the movie, I hate it."

Difference between sentences only regarding word order
(assuming only unigrams!)

Symbolic representations

- Problem: **No notion of similarity**
  - Words are just labels without meaning
  - Different words (syntax) ➜ orthogonal word vectors
    (even for words with the same/similar meaning)



- Goal: Similar words (meaning) ➜ similar word vectors
  - Word vectors no longer just labels but also encode "some" meaning
  - Improve basically all NLP and Text Mining tasks!

Move from:

Symbolic Representation of Words ----> Word Embeddings

Word Embeddings

- Word embeddings are vector representations of words in a continuous vector space

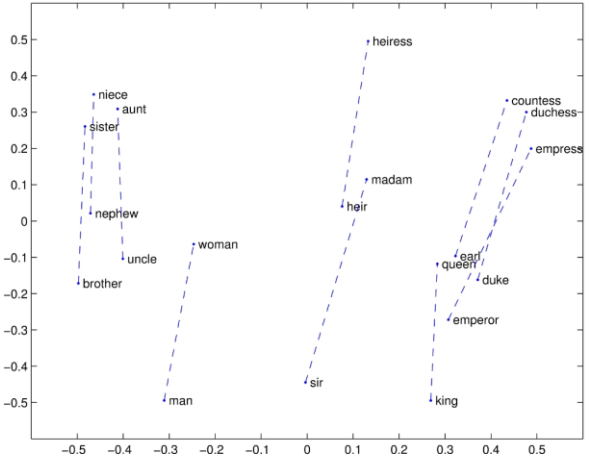
(A vector is bunch of numbers that represents the meaning of the word)

- They capture semantic and syntactic relationships between words mathematically
- They are based on the principle that we can gain knowledge of the **different contexts** in which a word is used by looking at its **neighbours**.

What does this mean?

Consider the words “ice” and “steam”. Different, yet related because they are both forms of water.

- So, we would expect words related to water (like “water” and “wet”) to appear equally in the context of “ice” and “steam”.
- In contrast, words like “cold” and “solid” would probably appear near “ice” but would not appear near “steam”.



Similar words occur together in the vector space

Source: Socher 2016, lecture 3, http://web.stanford.edu/class/cs224n/syllabus.html

These create dense, low-dimensional vectors!

What are the dimensions of the vector space?

The dimensions are abstract and derived from the data, so their exact meanings can vary depending on the training corpus and the embedding model used.

As a simple example, let's assume a cat is represented by the vector **[0.2, 0.8, -0.5]** in a 3-dimensional vector space:

0.2 could indicate it's an animal.

0.8 could indicate it's a domesticated animal.

-0.5 could indicate it's a small animal.

Main Approaches

1. **Word2Vec** (Mikolov et al., 2013)
2. GloVe (Global Vectors for Word Representation) (Pennington et al., 2014)
3. FastText (Bojanowski et al., 2017)

Static
(Each word has a fixed vector representation regardless of context)

E.g. Single vector for "bank" regardless of meaning

4. Elmo (Peters et al., 2018)
5. BERT (Devlin et al., 2019)

Contextual
(Different vector representations based on a word's surrounding context)

E.g. Different vectors for "I deposited money in the bank" and "I sat by the bank of the river."

Word2Vec

- word2vec is a group of models used for word embedding, Mikolov et al. 2013 (google inc.)
- Neural network models are trained on a large corpus of text (for example, all of Wikipedia) to reconstruct linguistic contexts of words.

Two algorithms:

1. Continuous bag-of-words (CBOW): predict the current word from a window of surrounding context words. Order does not matter
2. Skip-grams (SG): predict the surrounding window of context words from the current word

Word2Vec: CBOW & Skip-Gram

Source: Exploiting Similarities among Languages for Machine Translation

Continuous Bag of Words (CBOW)

Predict the current word from a window of surrounding context words. Order does not matter

Given context ➜ Predict target word

Skip-gram

Predict the surrounding window of context words from the current word

Target word ➜ Predict context

Example: "I **am** cold"

Context: ["I", "cold"] Target: "am"

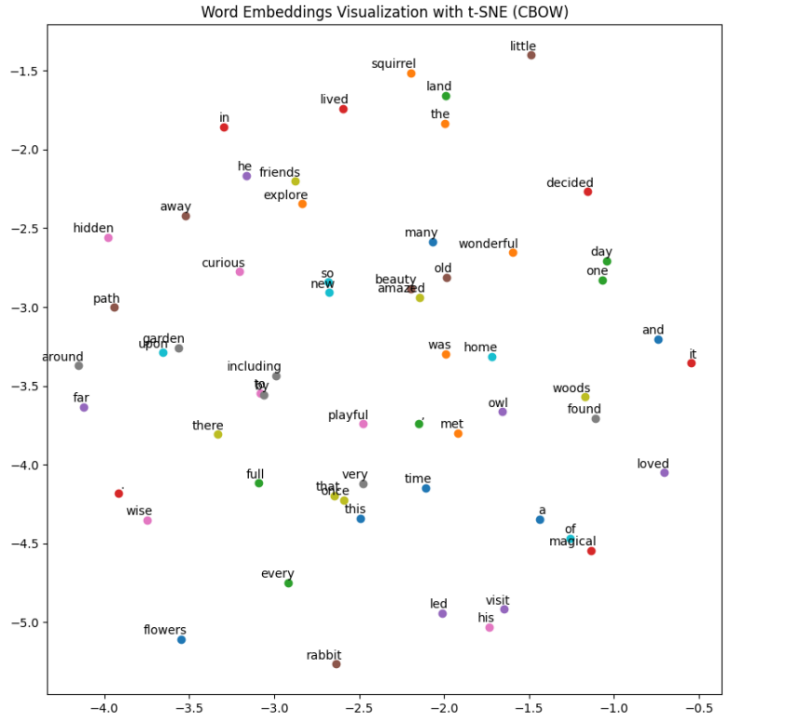
The model predicts "am" given "I" and "cold"

Target: "am" Context: ["I", "cold"]

The model predicts both "I" and "cold" given "am"

CBOW	Skip-gram
Generally faster and more efficient on large datasets	Performs better on smaller datasets
Works well when you want to capture the context of frequent words	More effective at capturing the context of rare words
If you need quicker training times, CBOW is a good choice	If you need higher accuracy and are willing to invest more training time, Skip-gram is preferable

Word2Vec Implementation




```
#function to process the text input and create a list of tokenized sentences
def createDataArray(text):
  # replaces newline with space to ensure the text is treated as a continuous string
  continuousText = text.replace("\n", " ")
  data = [] #initializes an empty list to store the tokenized sentences.
  # iterate through each sentence in the file
  for i in sent_tokenize(continuousText):
    temp = []

    # tokenize the sentence into words
    for j in word_tokenize(i):
        temp.append(j.lower())
    data.append(temp)
  return data

# Create a CBOW model
def createCBOWModel(data):
  cbow_model = gensim.models.Word2Vec(data, min_count=1,
                                      vector_size=100, window=5, sg=0)
  return cbow_model

# Create a Skip Gram model
def createSkipGramModel(data):
  skipgram_model = gensim.models.Word2Vec(data, min_count=1, vector_size=100,
                                      window=5, sg=1)
  return skipgram_model

#Parameters explained:
#min_count=1: Ignores all words with a total frequency lower than this
#vector_size=100: The dimensionality of the word vectors
#window=5: The maximum distance between the current and predicted word within a sentence
#sg=0: Specifies the training algorithm. 0 means CBOW, 1 means Skip-gram
```

Let's use a toy example to create a small word vector embedding

```
[ ] sample_text = """
    Once upon a time in a land far, far away, there lived a little rabbit.
    This rabbit was very curious and loved to explore the woods around his home.
    One day, he found a hidden path that led to a magical garden full of wonderful flowers.
    The rabbit was so amazed by the beauty of the garden that he decided to visit it every day.
    In the garden, he met many new friends, including a wise old owl and a playful squirrel.
    """
```

Testing our embedding

And test the model

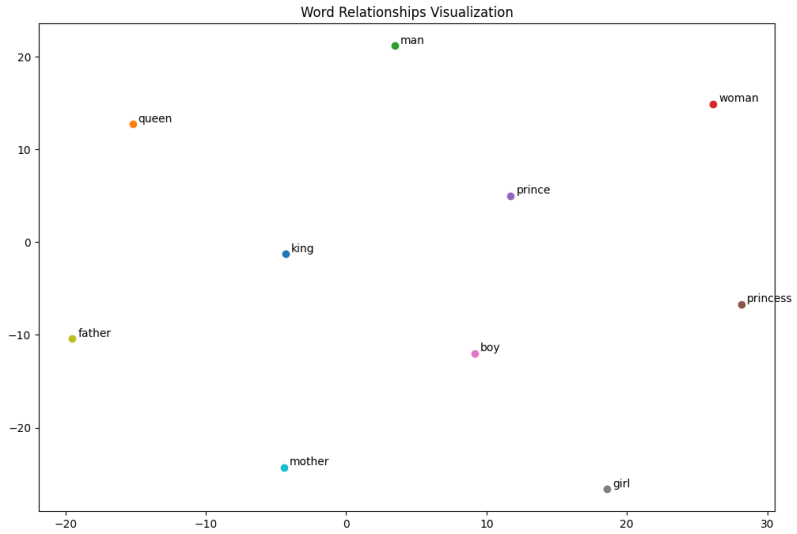
```
data = createDataArray(sample_text)
cbow_model = createCBOWModel(data)
skipgram_model = createSkipGramModel(data)

# Test the CBOW model
print("CBOW Model - Words similar to 'rabbit':")
similar_words_cbow = cbow_model.wv.most_similar("rabbit", topn=5)
for word, similarity in similar_words_cbow:
    print(f"{word}: {similarity:.4f}")

# Test the Skip-gram model
print("\nSkip-gram Model - Words similar to 'rabbit':")
similar_words_skipgram = skipgram_model.wv.most_similar("rabbit", topn=5)
for word, similarity in similar_words_skipgram:
    print(f"{word}: {similarity:.4f}")
```

```
CBOW Model - Words similar to 'rabbit':
was: 0.1666
this: 0.1624
he: 0.1389
.: 0.1313
once: 0.1283

Skip-gram Model - Words similar to 'rabbit':
was: 0.1659
this: 0.1624
he: 0.1388
.: 0.1321
once: 0.1276
```

```
#Get the vector for the word 'garden' from both models
garden_vector_cbow = cbow_model.wv['garden']
garden_vector_skipgram = skipgram_model.wv['garden']

print(f"\nCBOW Vector for 'garden': {garden_vector_cbow[:10]}...")  # Display the first 10 dimensions for brevity
print(f"Skip-gram Vector for 'garden': {garden_vector_skipgram[:10]}...")
```

```
CBOW Vector for 'garden': [ 0.00814698 -0.00442942  0.00898294  0.00828784 -0.00437581  0.00023826
  0.00431408 -0.00384858 -0.00560351 -0.00656039]...
Skip-gram Vector for 'garden': [ 8.0698775e-03 -4.3716696e-03  8.9669591e-03  8.3990060e-03
 -4.2797145e-03  6.6322013e-05  4.5168474e-03 -3.6192976e-03
 -5.7294723e-03 -6.6772620e-03]...
```

We also try a pre-trained embedding




```python
# First, let's import the necessary libraries
import gensim.downloader as api
from gensim.models import KeyedVectors
import numpy as np

# Download a pre-trained word2vec model
# This might take a few minutes depending on your internet connection
model = api.load('word2vec-google-news-300')

print("Model loaded successfully!")
```

Perform some similarity and analogy tasks using the model

```python
# Find similar words
similar_words = model.most_similar('computer', topn=5)
print("Words most similar to 'computer':")
for word, score in similar_words:
    print(f"{word}: {score:.4f}")

# Find the odd word out
odd_word = model.doesnt_match(['breakfast', 'cereal', 'dinner', 'lunch'])
print(f"\nOdd word out: {odd_word}")

# Solve analogies
result = model.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
print("\nSolving the analogy: man is to king as woman is to...")

# Accessing the first (and only) result
first_result = result[0]

# Accessing the word (first element of the tuple)
result_word = first_result[0]

# Accessing the score (second element of the tuple)
result_score = first_result[1]

# Printing the results
print("Word:", result_word)
print("Score: {:.4f}".format(result_score))
```

```
Words most similar to 'computer':
computers: 0.7979
laptop: 0.6640
laptop_computer: 0.6549
Computer: 0.6473
com_puter: 0.6082


Odd word out: cereal

Solving the analogy: man is to king as woman is to...
Word: queen
Score: 0.7118
```

Word2Vec — Practical Considerations

- Embeddings dependent on the application / dataset

Dataset: Wikipedia

```
1 word2vec_wikipedia.wv.most_similar("house")
```

```
[('mansion', 0.7079392075538635),
 ('cottage', 0.6541333198547363),
 ('farmhouse', 0.6259987950325012),
 ('barn', 0.5747625827789307),
 ('bungalow', 0.5724436044692993),
 ('townhouse', 0.567018449306488),
 ('houses', 0.5506472587585449),
 ('parsonage', 0.5426527857780457),
 ('tavern', 0.5370140671730042),
 ('summerhouse', 0.5307810306549072)]
```

Dataset: Google News

```
1 word2vec_googlenews.most_similar("house")
```

```
[('houses', 0.7072390913963318),
 ('bungalow', 0.6878559589385986),
 ('apartment', 0.6628996729850769),
 ('bedroom', 0.6496936678886414),
 ('townhouse', 0.6384080052375793),
 ('residence', 0.6198420524597168),
 ('mansion', 0.6058192253112793),
 ('farmhouse', 0.5857570171356201),
 ('duplex', 0.5757936239242554),
 ('appartment', 0.5690325498580933)]
```

Word2Vec — Limitations

- Unable to represent phrases
  - *"New York"*, *"snow cat"*, *"ice cream"*, *"land mine"*, *"hot dog"*, *"disc drive"*, etc.

- Unable to handle polysemy and part of speech
  - Polysemy: multiple meanings for the same word
  - Part of speech: the same word used as noun, verb, or adjective

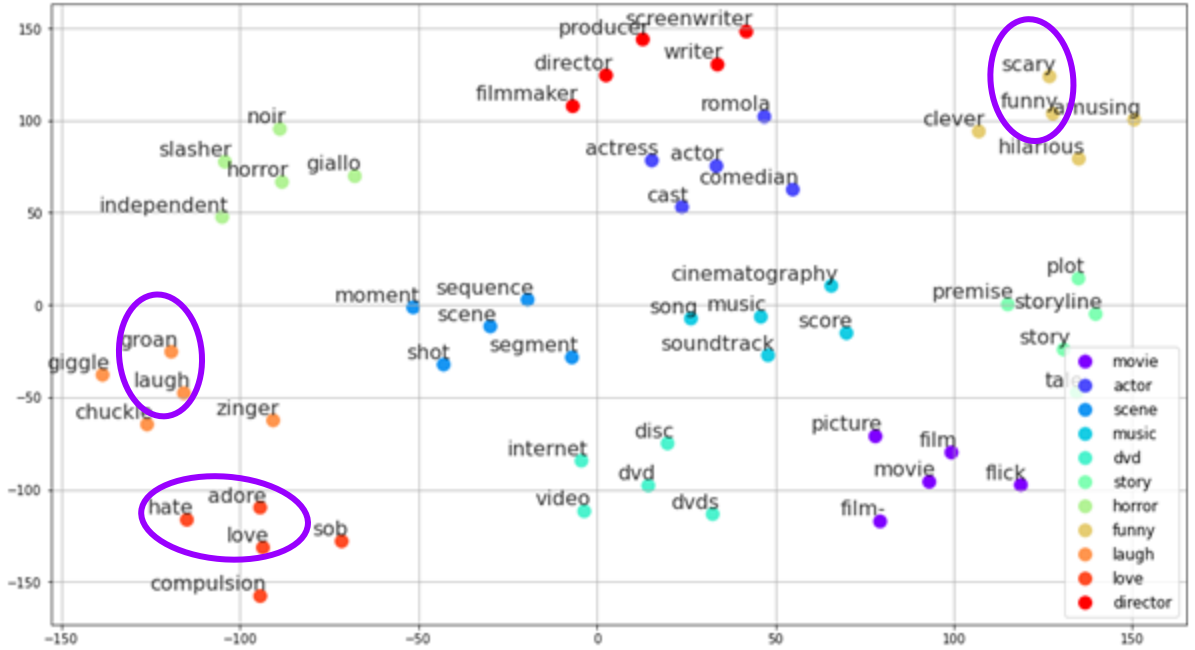
```
1 word2vec_wikipedia.wv.most_similar("light", topn=10)
```

```
[('lights', 0.5668156743049622),
 ('illumination', 0.5530915260314941),
 ('glow', 0.5415263175964355),
 ('sunlight', 0.5396571159362793),
 ('lamp', 0.5024341344833374),
 ('flame', 0.48772770166397095),
 ('lamps', 0.47849947214126587),
 ('dark', 0.4764614701271057),
 ('luminous', 0.4740492105484009),
 ('lighting', 0.47177615761756897)]
```

Word2Vec — Limitations

Why?

- Distributional representation does not capture all semantics
  - Common case: words with opposite polarity (sentiment)



Word2Vec — Limitations

- Embeddings can propagate human biases in the models
- For instance, machine learning tools were associating '**woman**' with words like '*kitchen*' and '*art*,' while associating '**man**' with '*science*' and '*technology*'
- If a supervised training model uses biased embeddings, it will also produce biased results!

How to apply vector embeddings?

Embeddings can be used as text representation (input) for pretty much any NLP problem!

Approach 1: Use a pre-trained embedding

- Commonly used approach - Often provides good performance out-of-the-box
- Leverages knowledge from large, diverse datasets
- Useful when you have limited training data
- Saves time and computational resources

Approach 2: Create your own embeddings

- Tailored to your specific domain or task
- Can capture unique vocabulary or language patterns
- More control over the embedding process
- Potentially better performance for specialized tasks

Approach 3: Hybrid Approach: Fine-tuning Pre-trained Embeddings

Word2Vec in Practice (directly taken from https://code.google.com/archive/p/word2vec/)

- Architecture:
  - Skip-gram: slower, better for infrequent words
  - CBOW (fast)
- Training:
  - Hierarchical softmax: better for infrequent words
  - Negative sampling: better for frequent words, better with low dimensional vectors
- Sub-sampling of frequent words
  - can improve both accuracy and speed for large data sets (useful values are in range 1e-3 to 1e-5)
- Dimensionality of the word vectors
  - usually more is better, but not always
- Context (window) size
  - For skip-gram usually around 10, for CBOW around 5

Join at menti.com | use code 6988 4345

Undoing Quiz - What do word embeddings represent?

0	0	0	0
Word embeddings are sparse vectors where each word is represented by a unique binary vector	Word embeddings are dense vectors that capture semantic relationships between words based on their context in large text corpora	Word embeddings are vectors where each dimension represents the frequency of a word in a document	Word embeddings are vectors where each dimension is a random number assigned to a word

Join at menti.com | use code 6988 4345

Feedback on embeddings

A. This is a common misconception. Sparse vectors with unique binary representations are characteristic of one-hot encoding, not word embeddings.

B. Correct! Word embeddings are dense vectors that encode semantic relationships by analyzing the context in which words appear in large text datasets. This allows similar words to have similar vector representations.

C. This is a misconception. Frequency counts, or term frequency, simply count how often a word appears in a document and do not capture semantic relationships.

D. This is incorrect. While initial vectors might be random in some training processes, word embeddings are trained on large text corpora to capture meaningful semantic relationships between words based on their context.

Take away messages - Embeddings

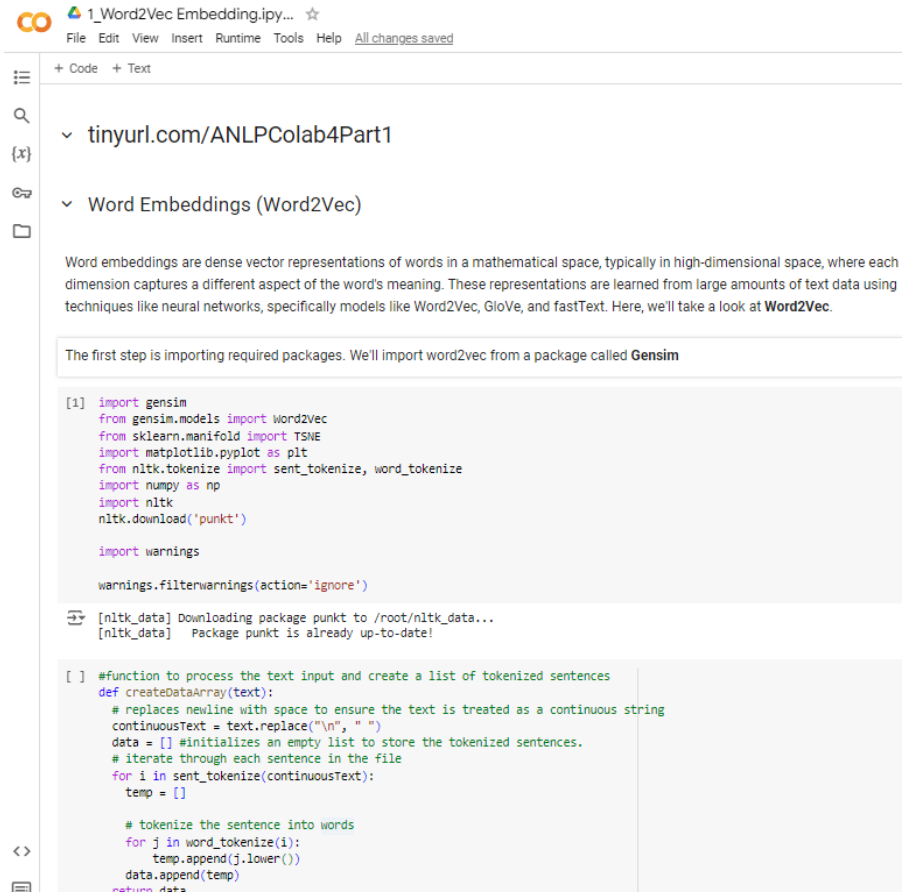
- Embeddings capture semantic and syntactic information about words or phrases in a dense vector format, making them suitable for many different types of NLP tasks
- Both static (e.g., Word2Vec, GloVe) and contextual embeddings (e.g., ELMo, BERT) can be used depending on the specific requirements of the task
- We can usually start with pre-trained embeddings (Check out this repository of word embeddings), then fine-tune with domain-specific data if needed

Tutorial

For today's tutorial, it is best to download the complete folder as we have several input files for the notebook later:

But you can access the current notebook using this link:

tinyurl.com/ANLPColab4Part1



Neural Networks and Deep Learning

Traditional ML → Deep Learning

Traditional Machine Learning (ML) methods have been reasonably effective for NLP tasks – We've seen a few examples:

Text Classification, Named Entity Recognition, Sentiment Analysis, Part-of-Speech Tagging, Topic Modeling, etc.

Deep Learning in NLP excels with scale…

Language Modeling E.g. Generating human-like text, completing sentences

Machine Translation E.g. Google Translate's neural machine translation

Sentiment Analysis E.g. Analyzing the sentiment of social media posts

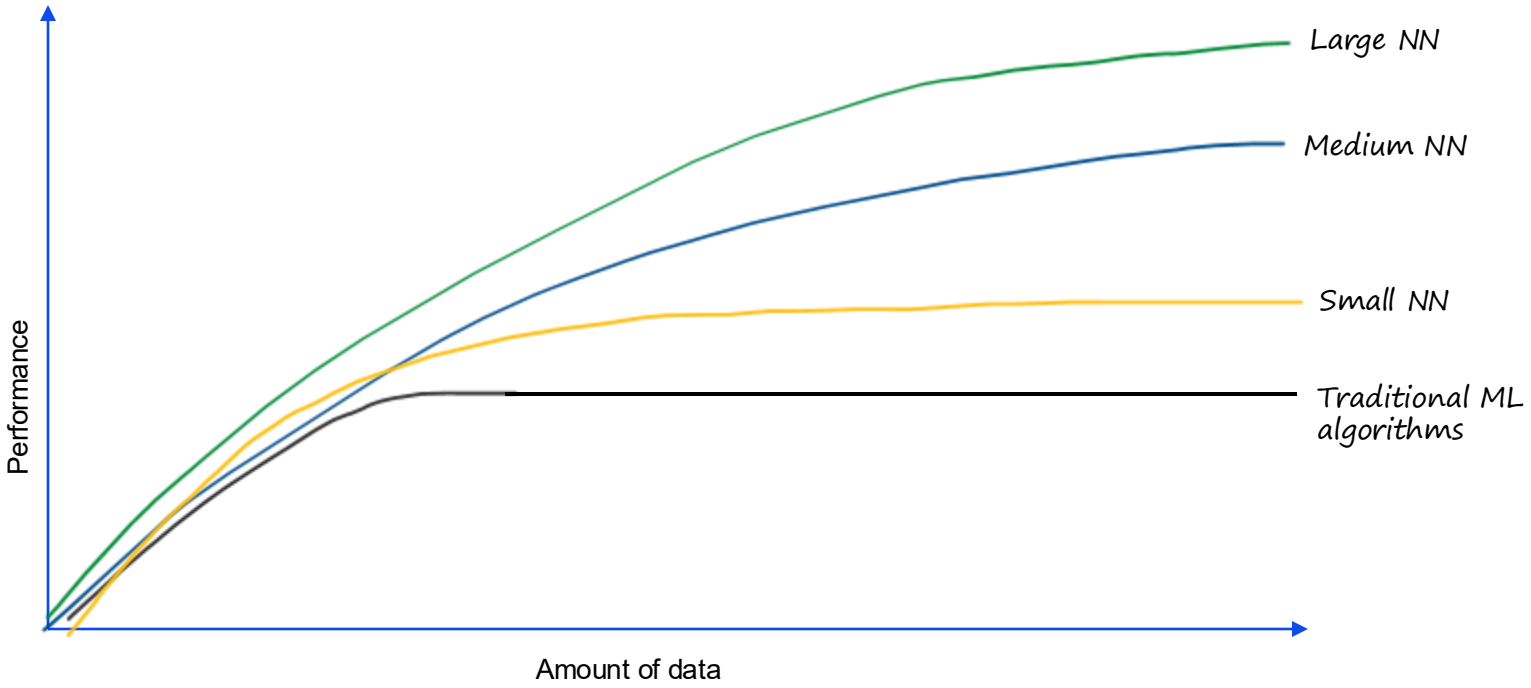
Named Entity Recognition E.g. Identifying and classifying entities in complex, domain-specific texts like medical records or legal documents

DL Applications (Contd..)

- Text Summarization E.g. Generating abstractive summaries of long articles
- Question Answering E.g. Open-domain question answering systems
- Speech Recognition E.g. Converting spoken language to text with high accuracy
- Dialogue Systems E.g. Chatbots in more natural, context-aware conversations
- Text-to-Speech E.g. Generating natural-sounding speech from text input
- Cross-lingual Transfer Learning E.g. Performing NLP tasks in low-resource languages by leveraging knowledge from high-resource languages

Most modern NLP (AI) applications are powered by Deep Learning!

Rise of Deep Learning



Scale drives deep learning progress

Source credit: Andrew Ng, Coursera DL course

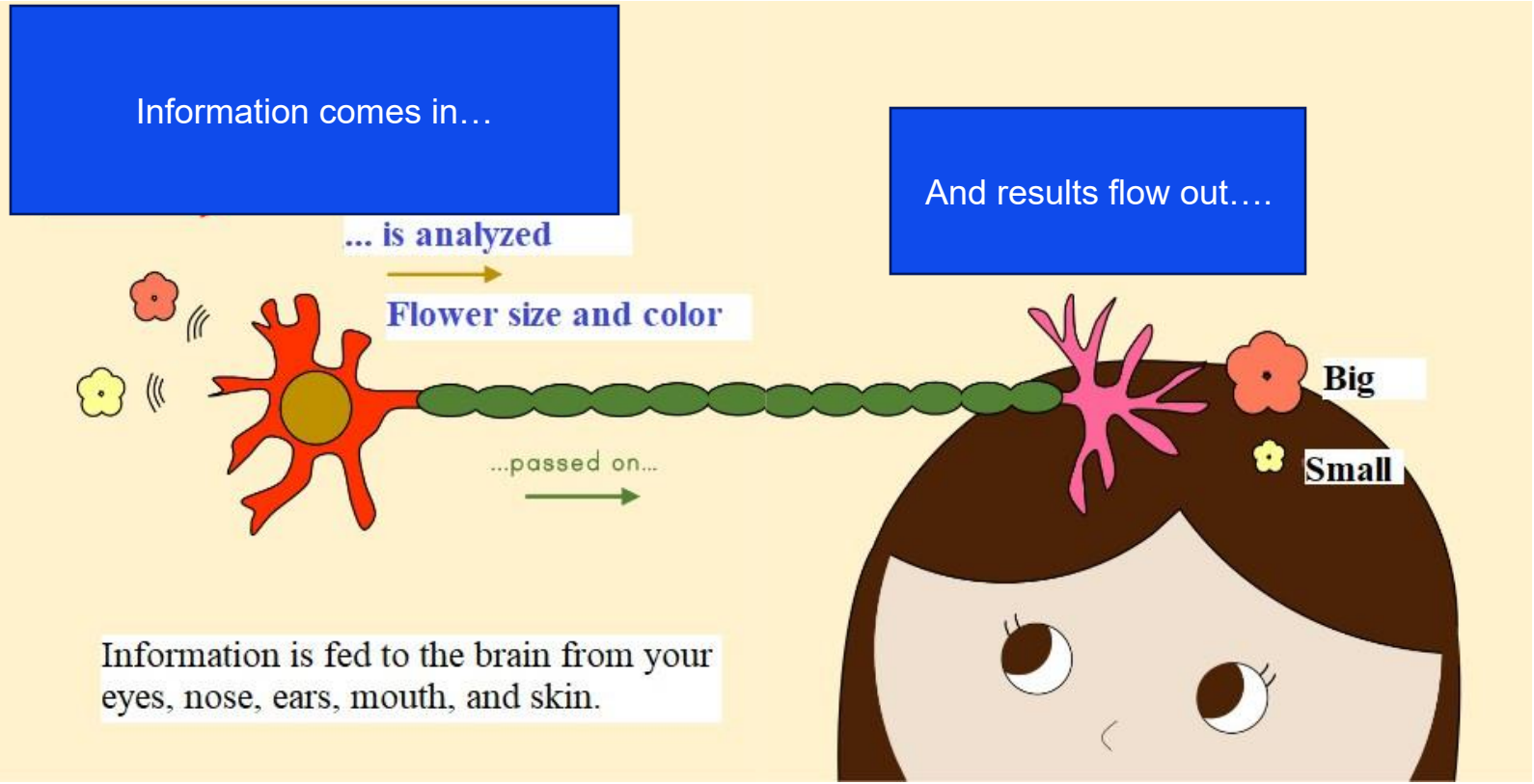
Innovations in

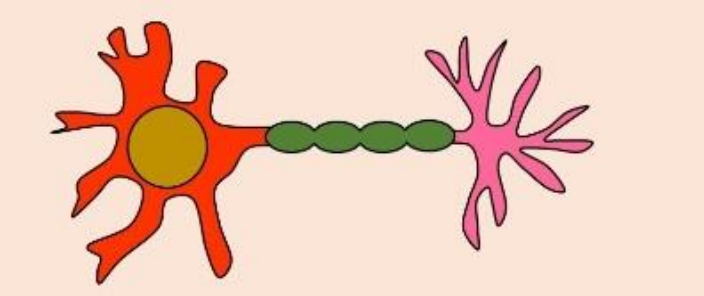
- Data
- Computation
- Algorithm

-> We are now able to run iterative experiments for better performance (E.g. switching from ReLU to Sigmoid)

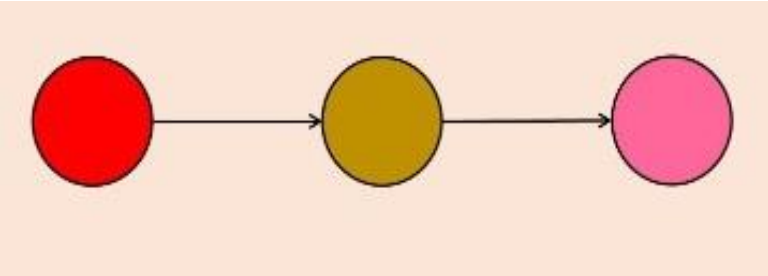

What is this?
How do you know this is a sunflower?
(with a bee)

Our brain makes these connections - Neurons help us understand our world!





Neurons in the human brain have dendrites, a cell body, an axon, and axon terminals



Neurons in a machine have nodes: input, output, and hidden layers

Algorithms we'll explore today

Perceptron: A simple neural network, which is an important foundational concept for understanding more complex neural networks

Multi-Layer Perceptrons (MLPs): Extends the perceptron by adding hidden layers and non-linear activation functions, enabling the network to learn complex patterns

Convolutional Neural Networks (CNNs): Specialized for processing grid-like data, such as images, by using convolutional layers to capture spatial hierarchies

Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTM) Networks: Designed for sequence data, such as time series or natural language, by maintaining memory of previous inputs

An introduction to foundational concepts, DL is a whole field!

Bayesian Inference with Tears

a tutorial workbook for natural language researchers

Kevin Knight

September 2009

1. Introduction

When I first saw this in a natural language paper, it certainly brought tears to my eyes:

$$P(k|\mathbf{x}_{-i},\beta) = \int P(k|\theta)P(\theta|\mathbf{x}_{-i},\beta)\,d\theta$$

Not tears of joy.

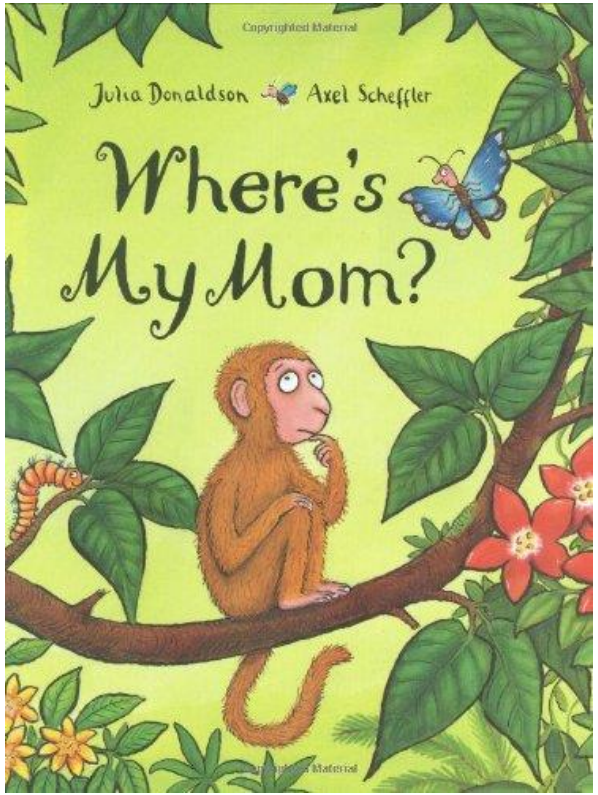
Later, I realized that I was no longer understanding many of the conference presentations I was attending. I was completely lost, just staring at some fly buzzing at the window. Afterwards, in the hallways, colleagues would jump up and down, exclaiming how simple it all was. "Put a Gaussian prior on the hyper-parameter!" they'd say. "Integrate over the possible values!" Or my all-time favorite: "Sample with *burn-in*!"

Here's some Kleenex. Let's go.

https://sites.socsci.uci.edu/~lpearl/courses/readings/Knight2009_BayesWithTears.pdf

Perceptron
(the simplest artificial NN)

Let’s read a book




"I've lost my mum!"

"Hush, little monkey, don't you cry.
I'll help you find her," said Butterfly.
"Let's have a think. How big is she?"

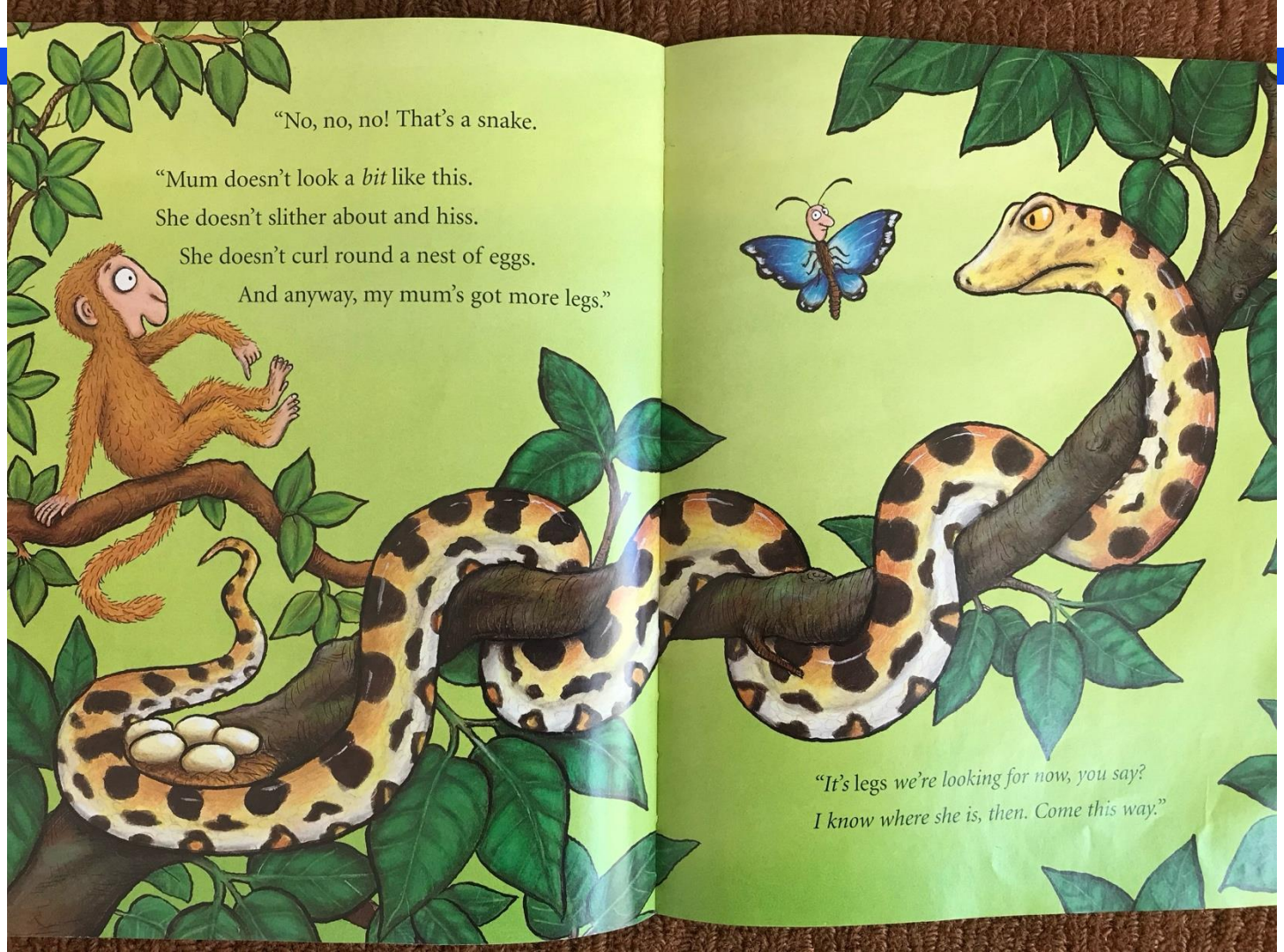
"She's big!" said the monkey. "Bigger than me."

"Bigger than you? Then I've seen your mum.
Come, little monkey, come, come, come."

"No, no, no! That's an elephant.

"My mum isn't a great grey hunk.
She hasn't got tusks or a curly trunk.
She doesn't have great thick baggy knees.
And anyway, *her* tail coils round trees."

"She coils round trees? Then she's very near.
Quick, little monkey! She's over here."



"No, no, no! That's a snake.

"Mum doesn't look a *bit* like this.
She doesn't slither about and hiss.
She doesn't curl round a nest of eggs.
And anyway, my mum's got more legs."

"It's legs *we're looking for now, you say?*
I know where she is, then. Come this way."


“Mum!”

What have we learned from this iasook?

Objects are described by their properties or *features*

- E.g.: isBig, hasTail, hasColor, numberOfLimbs…

Features have *values*

- Boolean: true/false
- Discrete: brown, white, etc.
- Numerical: 4 for numberOfLimbs

What have we learned from this book?

Objects are assigned a discrete *label*, e.g., isMyMom, isNotMyMom

A learning algorithm (the butterfly in the story) or *classifier* will learn how to assign labels to new objects

- Hint: features are important if they lead to the correct decision; less important otherwise

Learning algorithms produce incorrect classifications when not exposed to sufficient data. This situation is called *overfitting*.

Let's formalize what we know so far

Feature matrix **X**
One example per row
One feature per column

Label vector **y**

isBig	hasTail	hasTrunk	hasColor Brown	numberOf Limbs
1	1	1	0	4
0	1	0	0	0
0	1	0	1	4

Label
isNotMyMom
isNotMyMom
isMyMom

Is this a **supervised** or **unsupervised** approach?

Supervised classification!

Use case: text classification

Text classification = learning algorithms that assign labels to text

IMDB movie reviews dataset

Predicting the sentiment of a given movie review as positive or negative

Filename	Score	Binary Label	Review Text
`train/pos/24_8.txt`	8/10	Positive	*Although this was obviously a low-budget production, the performances and the songs in this movie are worth seeing. One of Walken's few musical roles to date. (he is a marvelous dancer and singer and he demonstrates his acrobatic skills as well - watch for the cartwheel!) Also starring Jason Connery. A great children's story and very likable characters.*
`train/neg/141_3.txt`	3/10	Negative	*This stalk and slash turkey manages to bring nothing new to an increasingly stale genre. A masked killer stalks young, pert girls and slaughters them in a variety of gruesome ways, none of which are particularly inventive. It's not scary, it's not clever, and it's not funny. So what was the point of it?*

More formally…

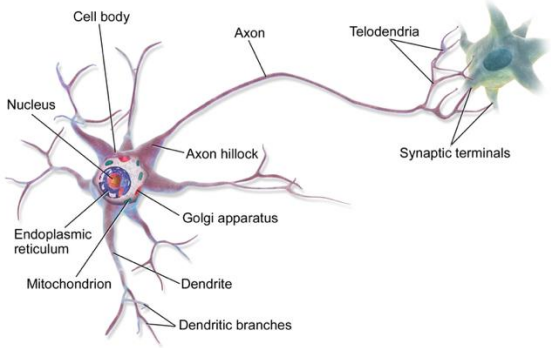
Feature matrix **X**
Individual feature: how many times a given word appears in a review

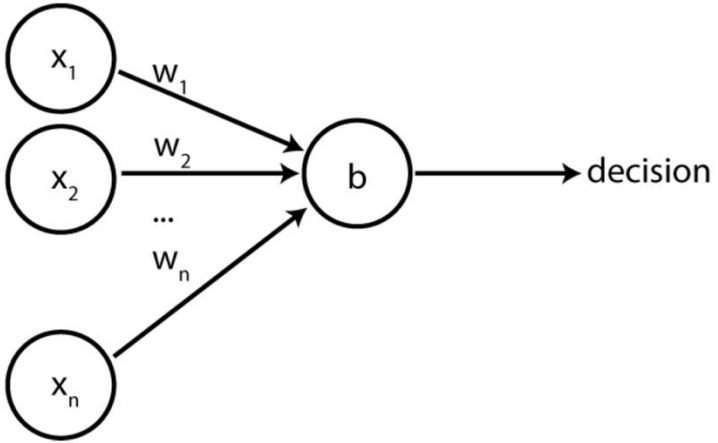
Label vector **y**

#	*good*	*excellent*	*bad*	*horrible*	*boring*	Label
#1	1	1	1	0	0	Positive
#2	0	0	1	1	0	Negative
#3	0	0	1	0	1	Negative

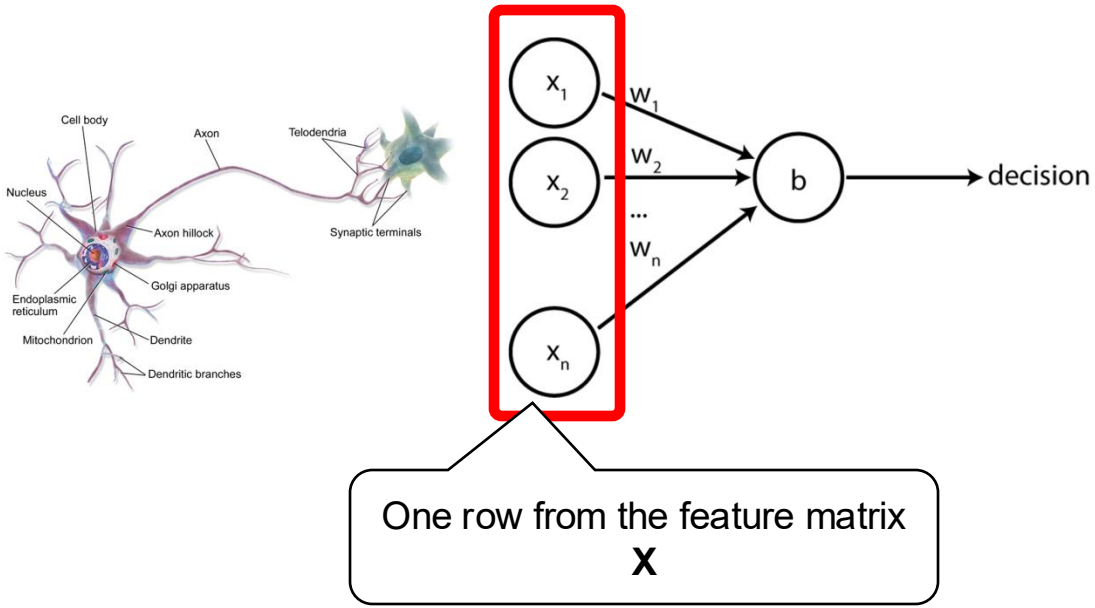
Quiz: how many columns does **X** have?

The perceptron

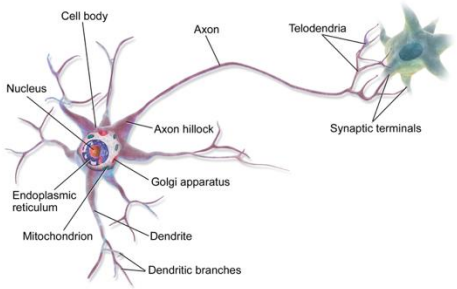


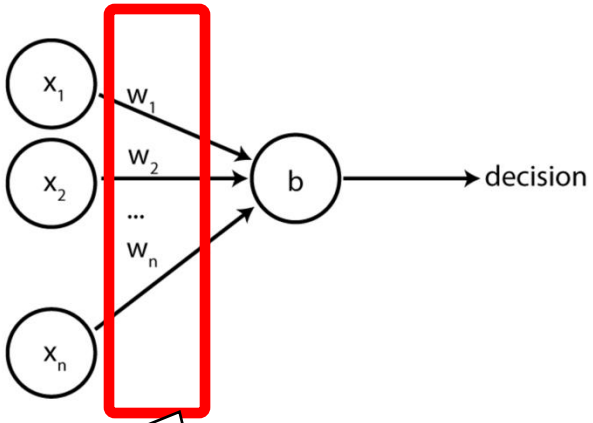


The perceptron



The perceptron



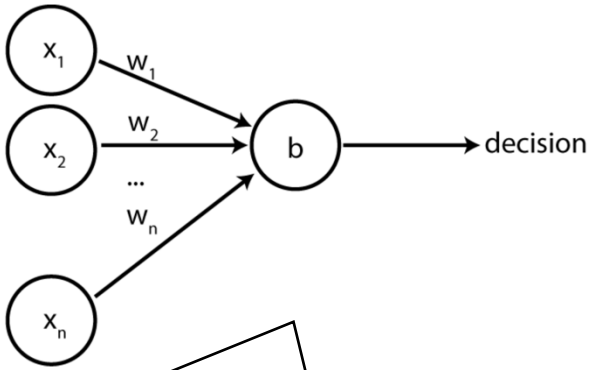


Weights that indicate how important each feature is

A simple intuition comparing weights with Linear Regression

Linear Regression
We explicitly define a linear function to model the relationship between input features and the target variable
E.g. House Price

Perceptron
Learns a decision boundary through an iterative process



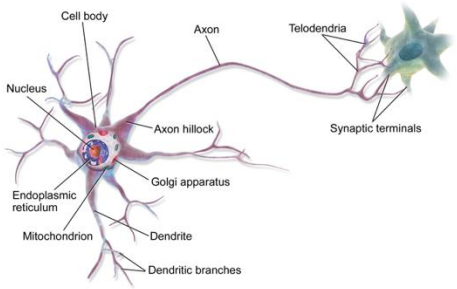
$$y = w_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n$$

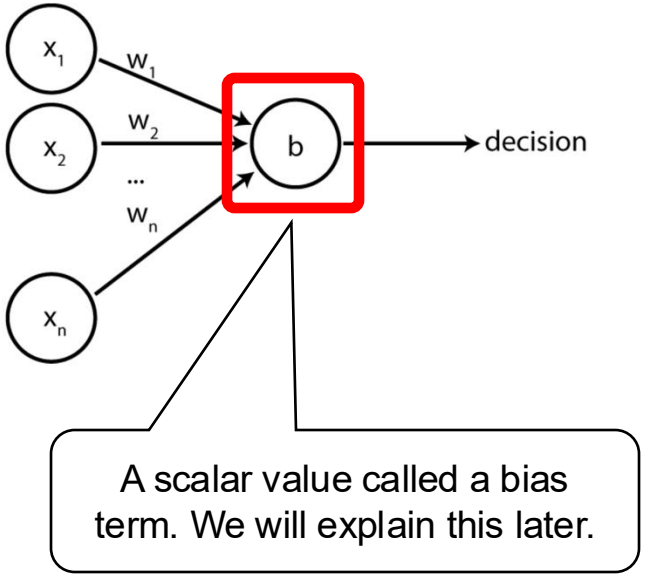
Where:

- y is the predicted house price
- w_0 is the bias term (intercept)
- w_1, w_2, ..., w_n are the weights for each feature
- x_1, x_2, ..., x_n are the input features (e.g., square footage, number of bedrooms, etc.)

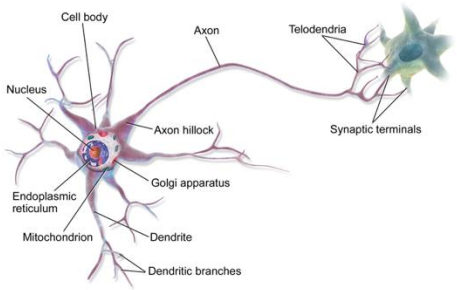
Weights are **learned when we train the NN!**

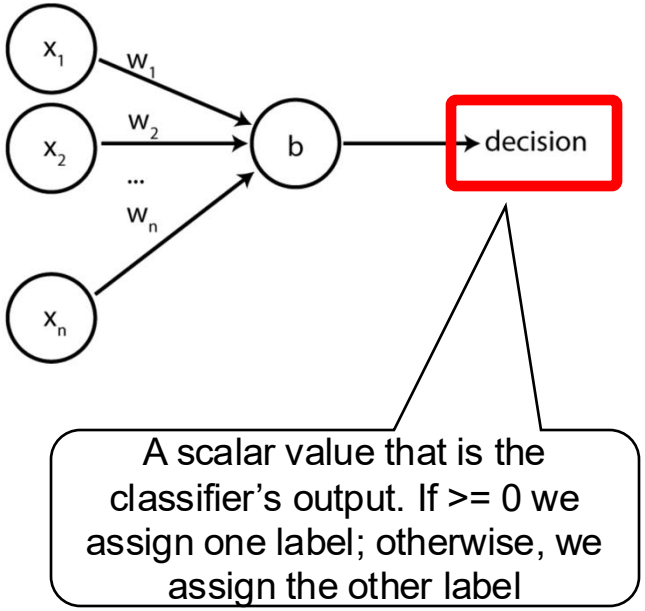
The perceptron



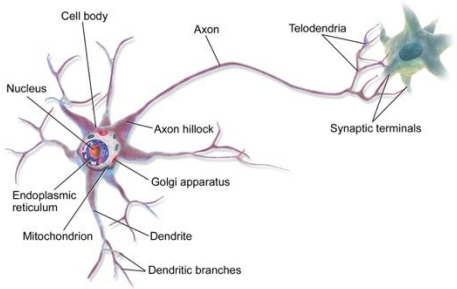


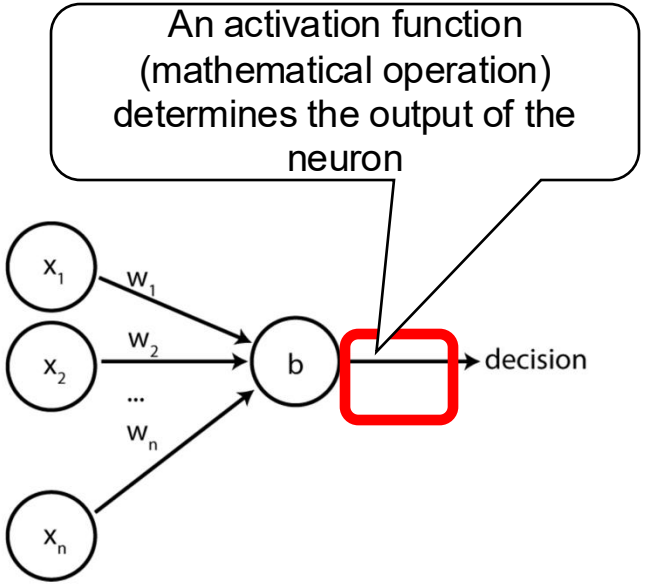
The perceptron





The perceptron





Perceptron decision function

Algorithm 1: The decision function of the perceptron.

```
if w · x + b > 0 then
    return Yes
else
    return No
end
```

Dot product of two vectors

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^{n} x_i y_i$$

Intuition

If the Yes class is isMyMom, then

- We want the weight associated with hasColorBrown to be positive, and
- The weight for hasTrunk to be negative

Similarly, for review classification (Yes == Positive)

- we want positive words to have positive weights, and negative words to have negative weights

How do we learn the perceptron weights to achieve this?

Training the perceptron

Initialize weights and bias with 0s

Algorithm 2: Perceptron learning algorithm.

```
1  w = 0
2  b = 0
3  while not converged do
4      for each training example x_i in X do
5          d = decision(x_i, w, b)
6          if d == y_i then
7              continue
8          else if y_i == Yes and d == No then
9              b = b + 1
10             w = w + x_i
11         else if y_i == No and d == Yes then
12             b = b − 1
13             w = w − x_i
14     end
15 end
```

Training the perceptron

Cycle through the training data until the weights no longer change

Algorithm 2: Perceptron learning algorithm.

```
1  w = 0
2  b = 0
3  while not converged do
4      for each training example x_i in X do
5          d = decision(x_i, w, b)
6          if d == y_i then
7              continue
8          else if y_i == Yes and d == No then
9              b = b + 1
10             w = w + x_i
11         else if y_i == No and d == Yes then
12             b = b − 1
13             w = w − x_i
14     end
15 end
```

Training the perceptron

Predict label using the current weights

Algorithm 2: Perceptron learning algorithm.

```
1  w = 0
2  b = 0
3  while not converged do
4      for each training example x_i in X do
5          d = decision(x_i, w, b)
6          if d == y_i then
7              continue
8          else if y_i == Yes and d == No then
9              b = b + 1
10             w = w + x_i
11         else if y_i == No and d == Yes then
12             b = b - 1
13             w = w - x_i
14     end
15 end
```

Training the perceptron

Algorithm 2: Perceptron learning algorithm.

```
1  w = 0
2  b = 0
3  while not converged do
4      for each training example x_i in X do
5          d = decision(x_i, w, b)
6          if d == y_i then
7              continue
8          else if y_i == Yes and d == No then
9              b = b + 1
10             w = w + x_i
11         else if y_i == No and d == Yes then
12             b = b - 1
13             w = w - x_i
14     end
15 end
```

The label is correct. Move on

Training the perceptron

We made a mistake on the positive (Yes) label

Add x_i to the weights

Algorithm 2: Perceptron learning algorithm.

```
1  w = 0
2  b = 0
3  while not converged do
4      for each training example x_i in X do
5          d = decision(x_i, w, b)
6          if d == y_i then
7              continue
8          else if y_i == Yes and d == No then
9              b = b + 1
10             w = w + x_i
11         else if y_i == No and d == Yes then
12             b = b - 1
13             w = w - x_i
14     end
15 end
```

Training the perceptron

Algorithm 2: Perceptron learning algorithm.

```
1  w = 0
2  b = 0
3  while not converged do
4      for each training example x_i in X do
5          d = decision(x_i, w, b)
6          if d == y_i then
7              continue
8          else if y_i == Yes and d == No then
9              b = b + 1
10             w = w + x_i
11         else if y_i == No and d == Yes then
12             b = b - 1
13             w = w - x_i
14     end
15 end
```

We made a mistake on the negative (No) label

Subtract x_i from the weights

Exercise

What are the perceptron's weights after one pass over this training data?

Initially:

- **w** = (0, 0, 0, 0, 0)
- b = 0

#	*good*	*excellent*	*bad*	*horrible*	*boring*	Label
#1	1	1	1	0	0	Positive
#2	0	0	1	1	0	Negative
#3	0	0	1	0	1	Negative

Exercise

#	*good*	*excellent*	*bad*	*horrible*	*boring*	Label
#1	1	1	1	0	0	Positive
#2	0	0	1	1	0	Negative
#3	0	0	1	0	1	Negative

Example seen: #1

$\mathbf{x} \cdot \mathbf{w} + b =$

Decision $=$

Exercise

#	*good*	*excellent*	*bad*	*horrible*	*boring*	Label
#1	1	1	1	0	0	Positive
#2	0	0	1	1	0	Negative
#3	0	0	1	0	1	Negative

Example seen: #2

$\mathbf{x} \cdot \mathbf{w} + b =$

Decision $=$

Exercise

#	*good*	*excellent*	*bad*	*horrible*	*boring*	Label
#1	1	1	1	0	0	Positive
#2	0	0	1	1	0	Negative
#3	0	0	1	0	1	Negative

Example seen: #3

$\mathbf{x} \cdot \mathbf{w} + b =$

Decision $=$

But why do we need that bias term?

In some cases, the perceptron cannot learn without a bias term.

Let’s see an example

But why do we need that bias term?

After 4 passes (epochs) over this training data, the learned parameters are:

- **w** = ⟨2⟩
- b = -4
- Decision: 2**x** – 4 > 0

That is, a review must have at least three positive words to receive a positive label

#	Number of positive words	Label
#1	1	Negative
#2	10	Positive
#3	2	Negative
#4	20	Positive

The bias term

- Allows the activation function to be shifted left or right. This means that even if all input features are zero, the neuron can still produce a non-zero output, which helps in better fitting the data.
- Helps the model better capture the underlying patterns in the data, leading to improved accuracy. It essentially acts as an additional parameter that the model can adjust during training.
- Helps in handling non-linear relationships between input features and the target variable. Without bias, the model might struggle to learn complex patterns!

Visualizing the perceptron learning

In general, this line is a hyper plane. Classifiers whose decision boundary is a hyper plane are **linear classifiers.**

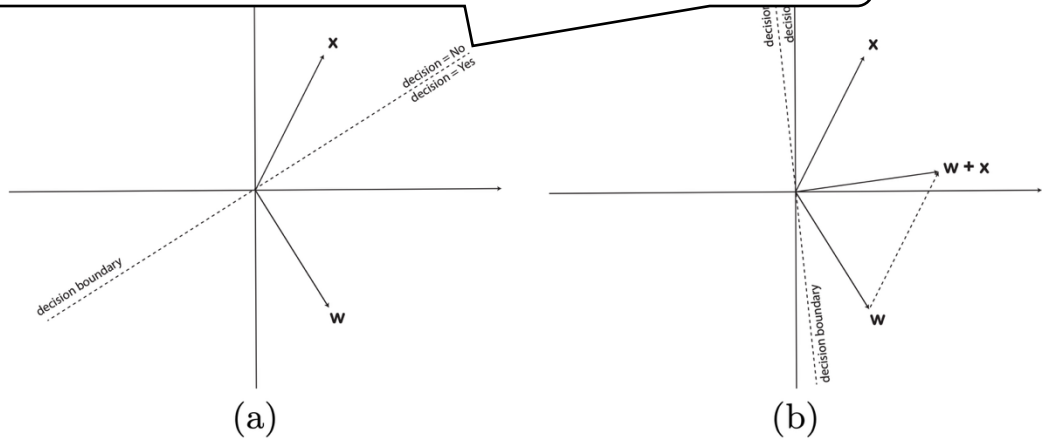


Figure 2.4 Visualization of the perceptron learning algorithm: (a) incorrect classification of the vector $\mathbf{x}$ with the label Yes, for a given weight vector $\mathbf{w}$; and (b) $\mathbf{x}$ lies on the correct side of the decision boundary after $\mathbf{x}$ is added to $\mathbf{w}$.

Drawbacks of the perceptron

It can only learn linear functions.

- We hardcoded: $\mathbf{w} \cdot \mathbf{x} + b$

It has no “smooth” updates during training.

- This leads to instability during training and poorer performance during evaluation.

Perceptron - Snapshot

Structure: Simplest form of artificial neural network • Single-layer neural network

Function:

- Performs binary classification
- Acts as a linear classifier

Components: • Input layer • Weighted sum • Activation function (typically step function)

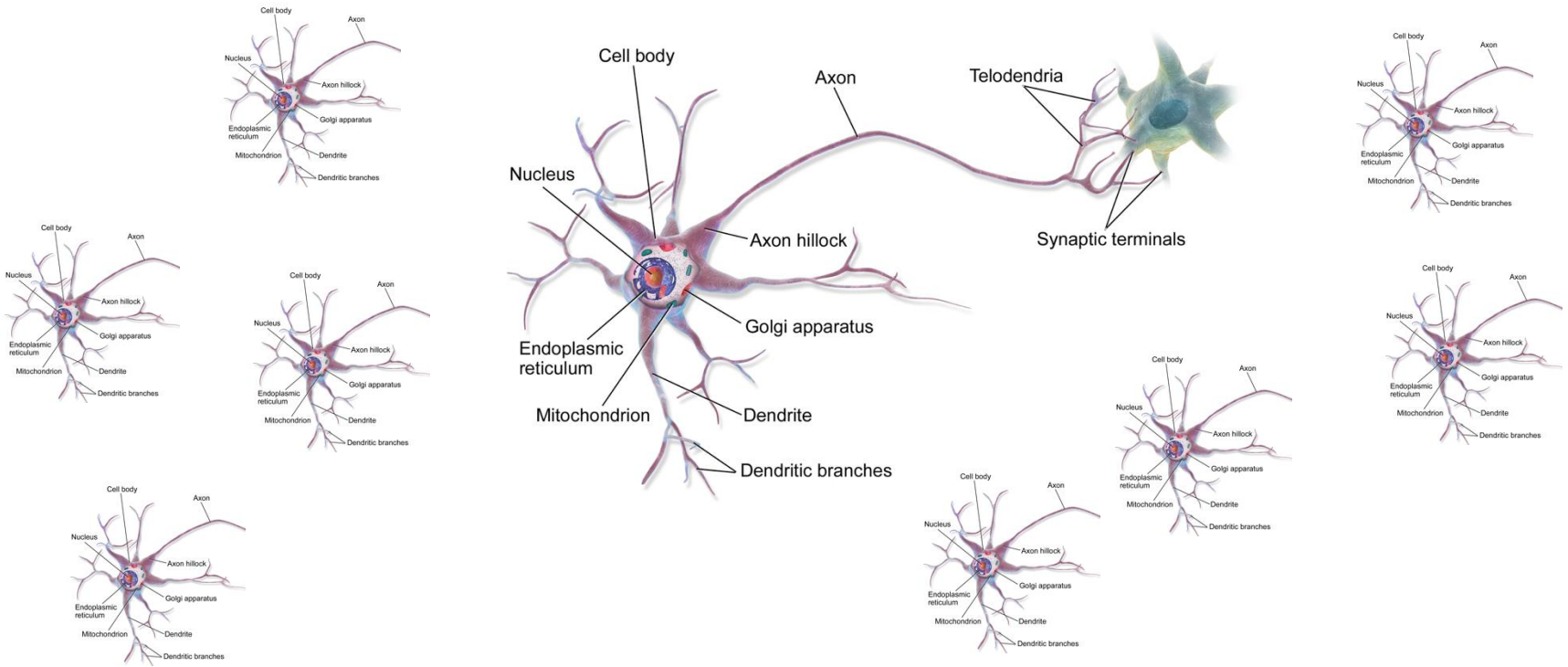
Operation: • Takes multiple inputs, applies weights, sums them up • Passes the sum through an activation function • Produces a binary output (0 or 1)

We generally don’t use the perceptron in practical applications, but it is fundamental to understanding more complex NNs!

Attendance check, Break

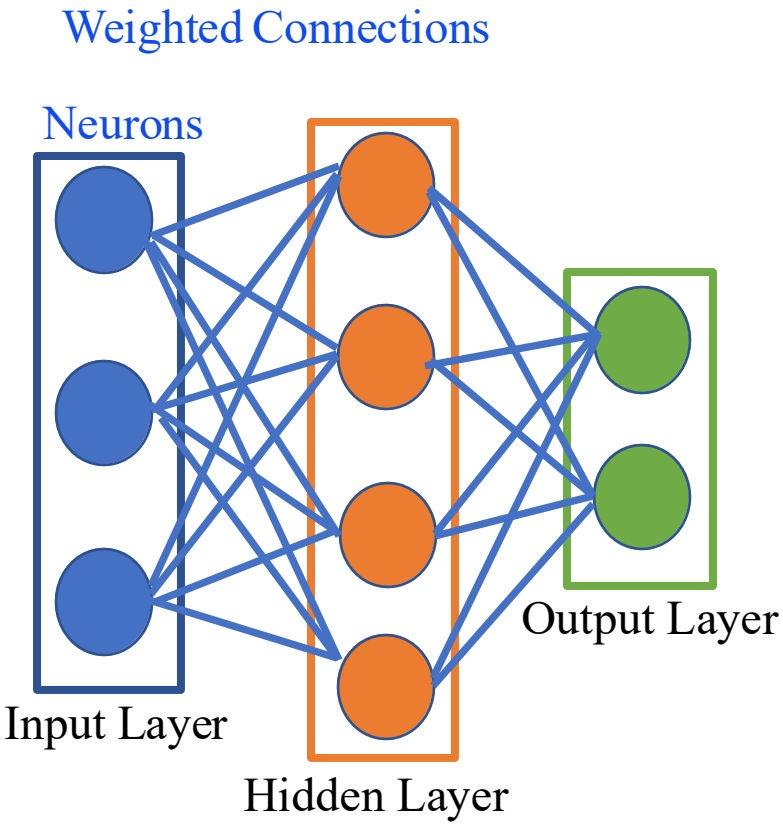
Multi-Layer Perceptrons (MLPs)/ Feed forward neural networks

This is in our brain



Neurons process information and pass messages to other neurons

This is in a machine



This is probably where the biological synergies end…

Neural Network Unit

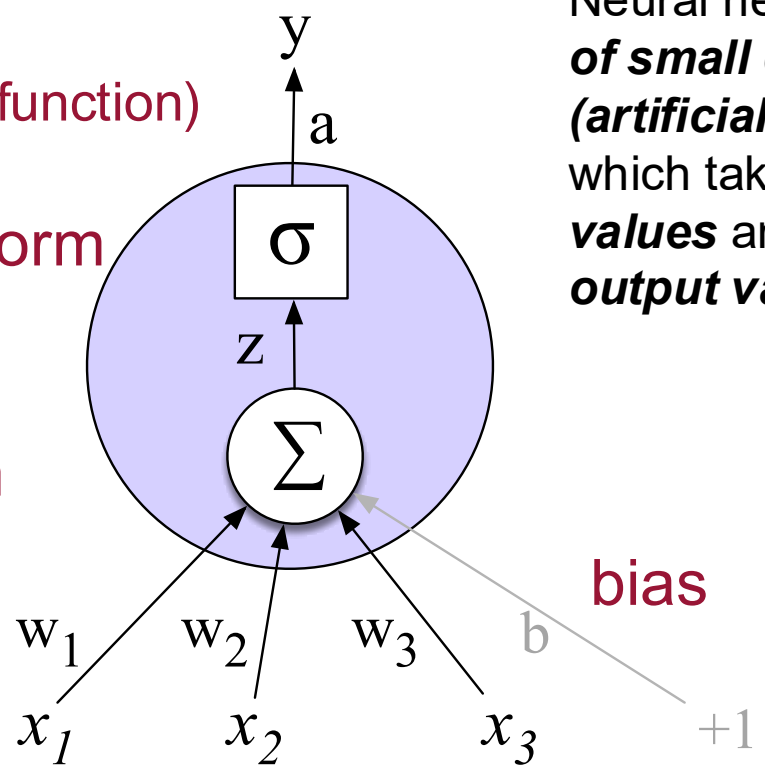
Output value
(using an activation function)

Non-linear transform

Weighted sum

Weights

Input layer



Neural network is a ***network of small computing units (artificial neurons)***, each of which takes a ***vector of input values*** and produces a ***single output value***

Why not linear functions?

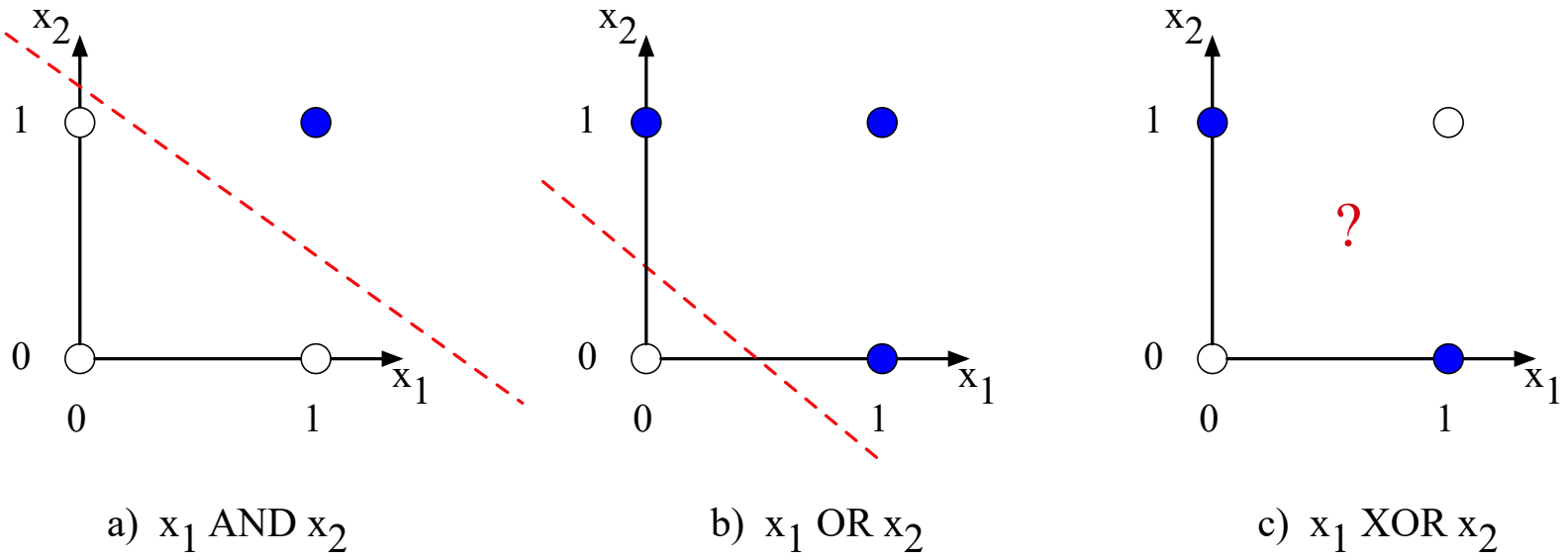
- With linear activations, each neuron can only represent linear boundaries or planes in the feature space
- If we only used linear activation functions, no matter how many layers we add to the network, the final output would still be a linear combination of the inputs

Let's consider the XOR problem, a classic example where a linear model fails:

Input 1	**Input 2**	**Output**
0	0	0
0	1	1
1	0	1
1	1	0

Where have you used a linear function?

Decision boundaries



a) x_1 AND x_2

b) x_1 OR x_2

c) x_1 XOR x_2

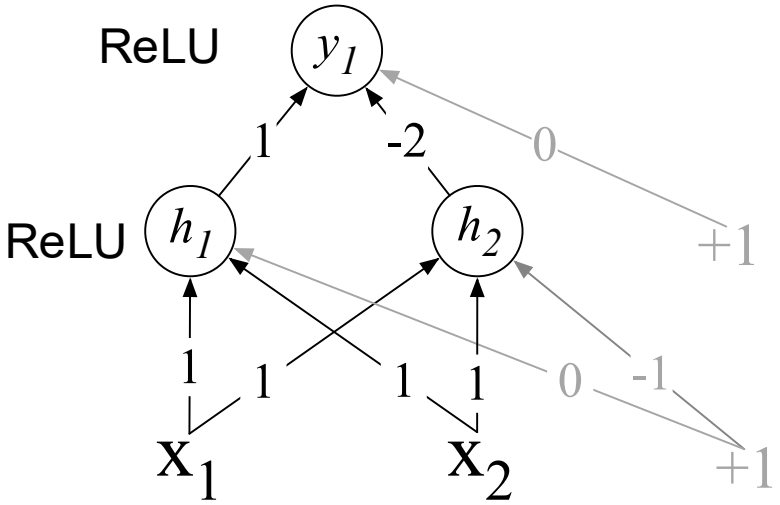
XOR is not a **linearly separable** function!

Solution to the XOR problem

XOR **can't** be calculated by a single perceptron

XOR **can** be calculated by a layered network of units

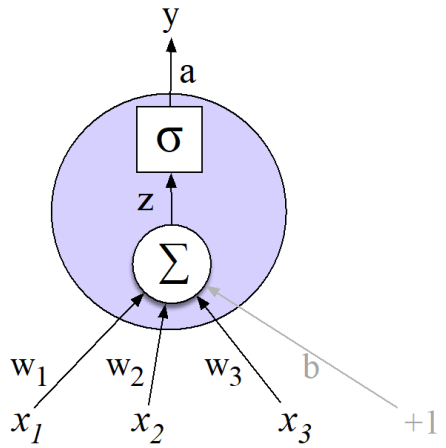
XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



Neural unit

Take weighted sum of inputs, plus a bias

$$z = b + \sum_i w_i x_i$$

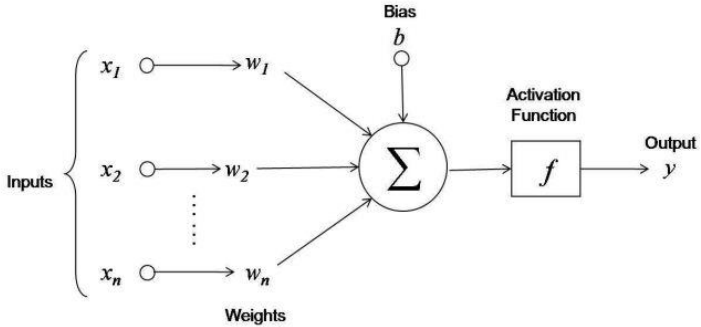


Expressing the weighted sum in vector notation (dot product)

$$z = w \cdot x + b$$

Instead of just using z, we'll apply a nonlinear activation function f:

$$y = a = f(z)$$



Since we are just modeling a single unit, the activation for the node is the final output, which we'll generally call y

Source: https://web.stanford.edu/~jurafsky/slp3/7.pdf

Activation function

One of the choices you get to make is what ***activation function*** to use in the hidden layers as well as at the output units of your neural network

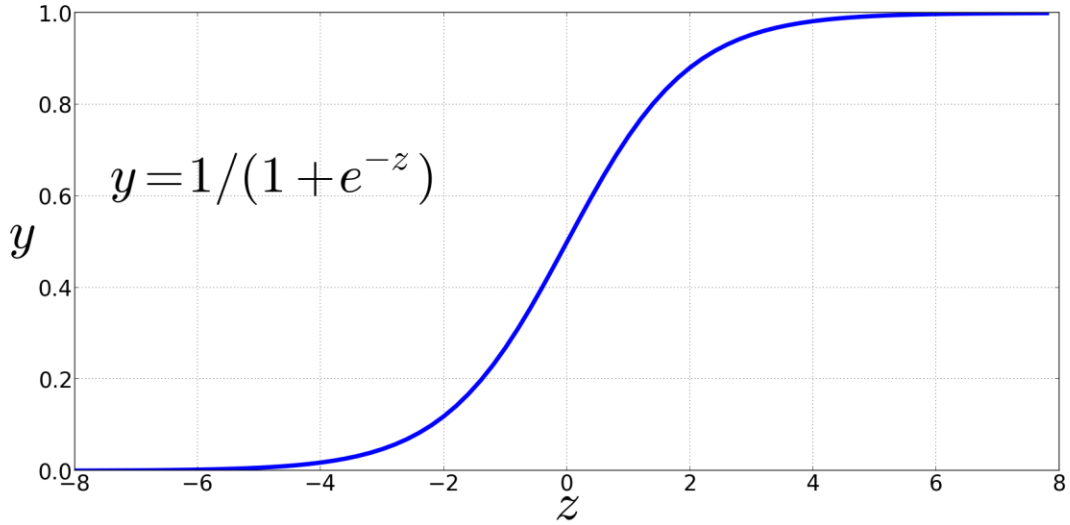
Non-Linear Activation Functions

Non-Linear Activation Functions

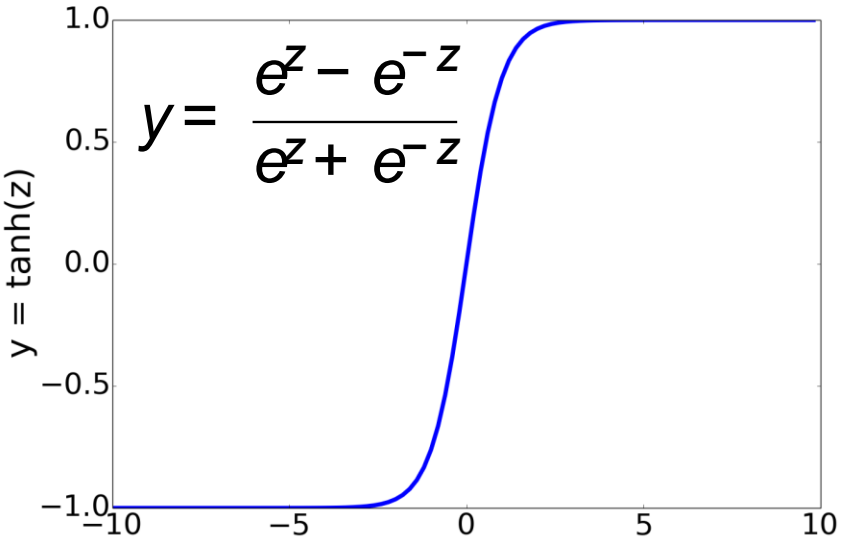
We're already seen the sigmoid for logistic regression:

Sigmoid

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

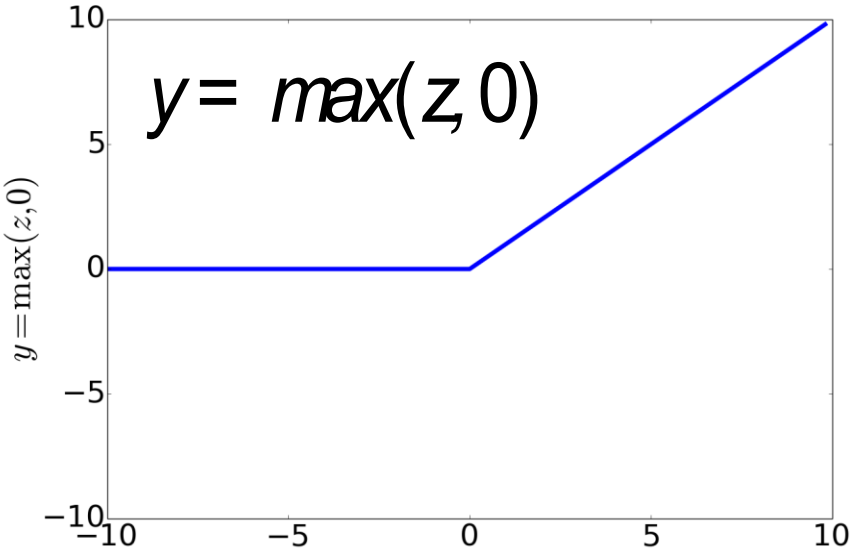


Non-Linear Activation Functions besides sigmoid



tanh

Most Common:



ReLU
Rectified Linear Unit

Final function the unit is computing

Applying the ***sigmoid activation function*** for computing y from an input x, a weight vector w, and a bias term b:

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

Final neural unit again

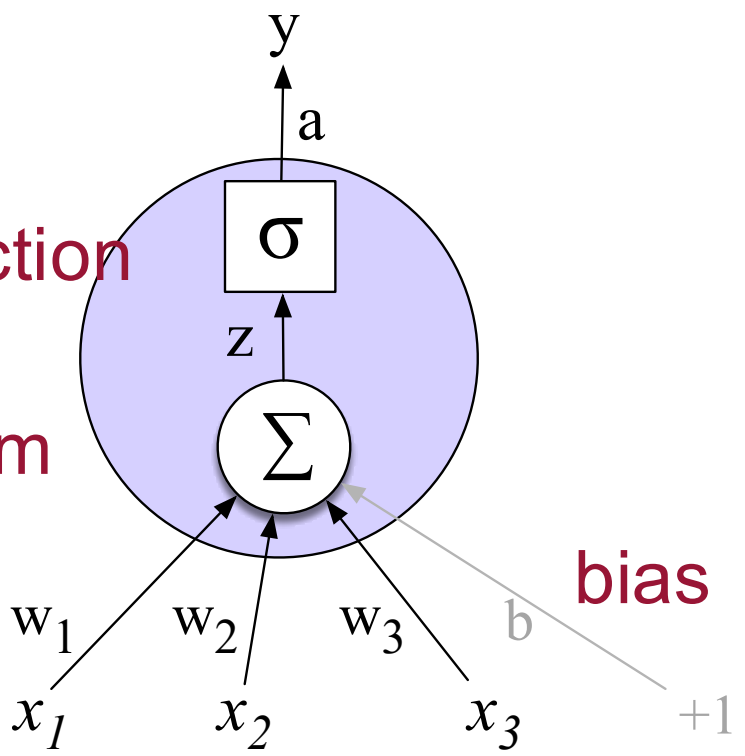
Output value

Non-linear activation function

Weighted sum

Weights

Input layer



An example

Suppose a unit has:

w = [0.2,0.3,0.9]

b = 0.5

What happens with the following input x?

x = [0.5,0.6,0.1]

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

An example

Suppose a unit has:

$w = [0.2, 0.3, 0.9]$

$b = 0.5$

What happens with input x:

$x = [0.5, 0.6, 0.1]$

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

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

An example

Suppose a unit has:

$w = [0.2, 0.3, 0.9]$

$b = 0.5$

What happens with input x:

$x = [0.5, 0.6, 0.1]$

What does **y = 0.70** mean for a Sentiment Analysis model?

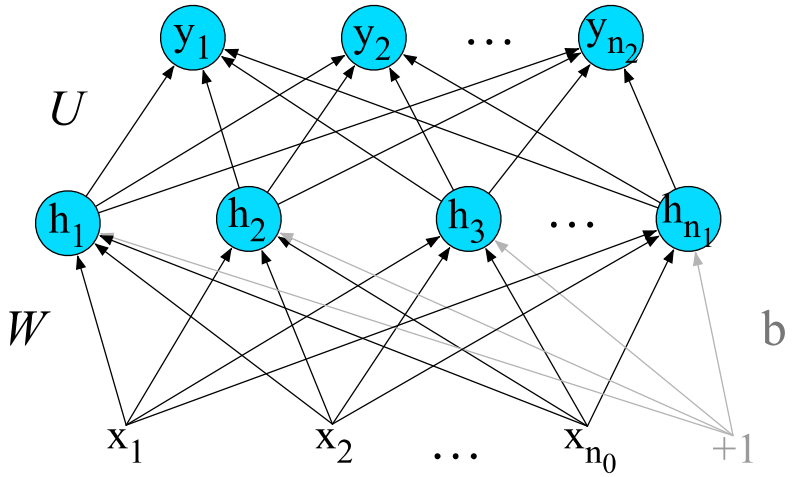
This is the output from a single neuron

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

$$\frac{1}{1 + e^{-(.5*.2 + .6*.3 + .1*.9 + .5)}} = \frac{1}{1 + e^{-0.87}} = .70$$

Feedforward neural network

- Feedforward neural network is a multilayer network in which the units are connected with no cycles. The outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers
- Simple feedforward networks have three kinds of nodes: input units, hidden units, and output units



Modern NNs are often deep (have many layers) → **Deep learning**

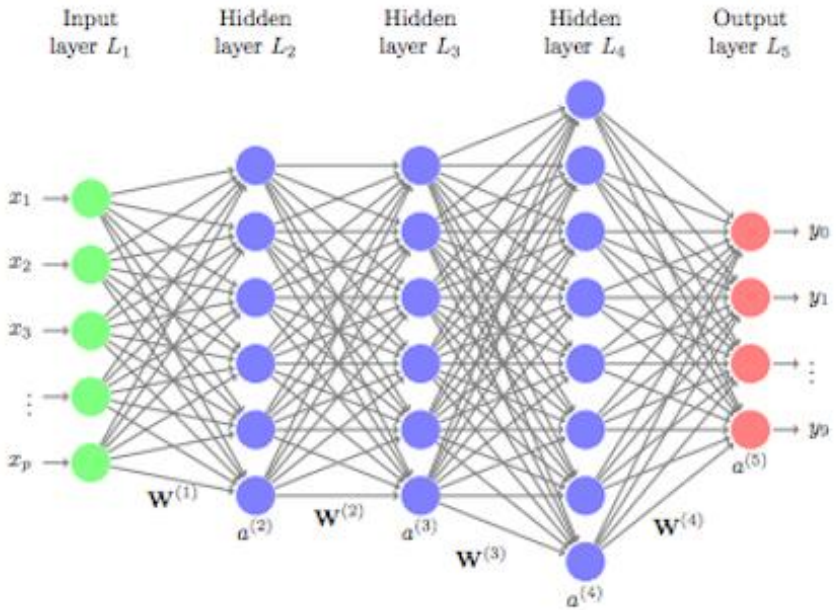
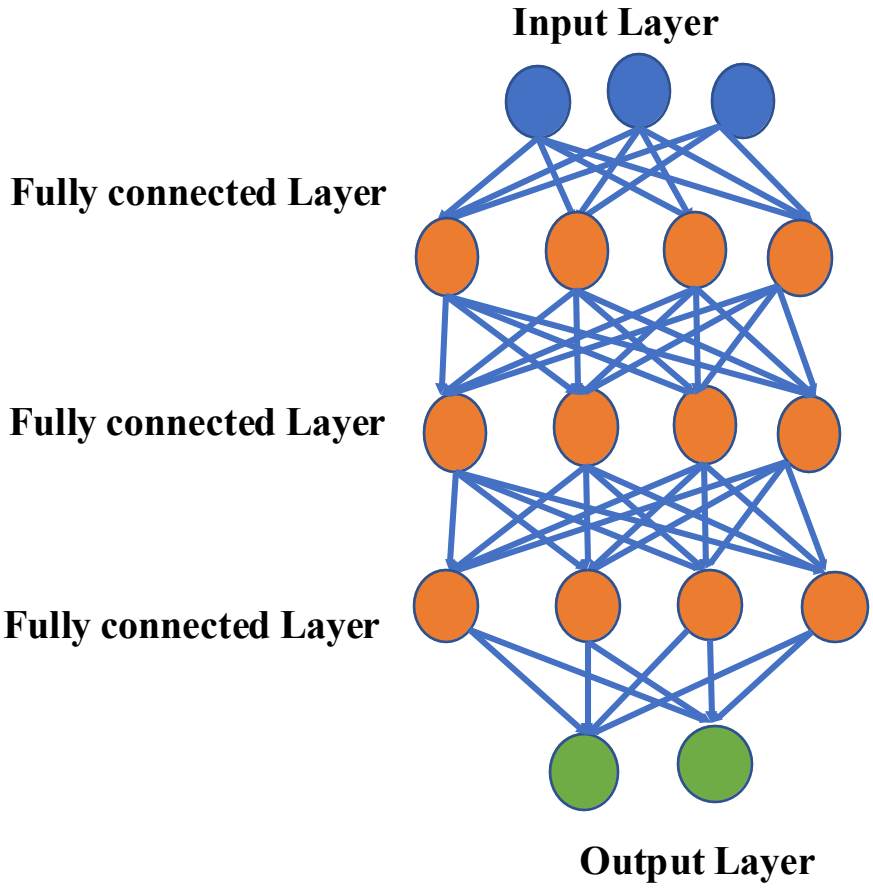
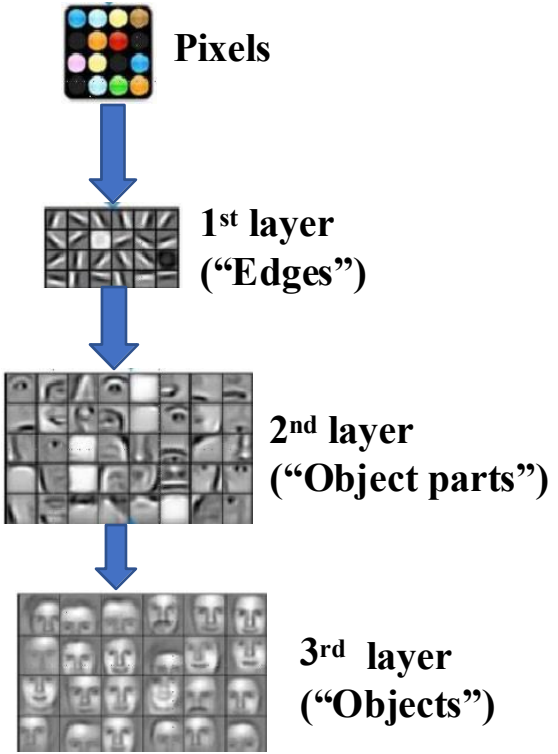


Image Source: https://uc-r.github.io/feedforward_DNN

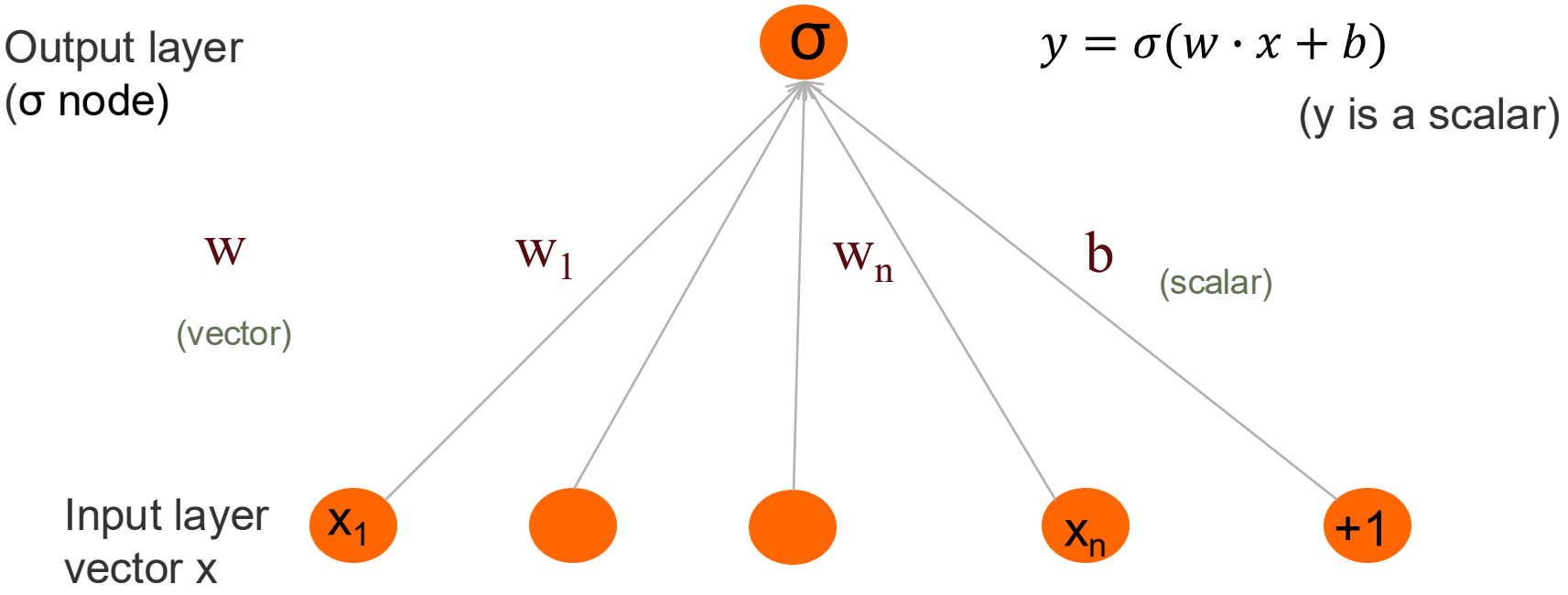
Deep neural network – Image Example





Binary Logistic Regression as a 1-layer Network

(we don't count the input layer in counting layers!)

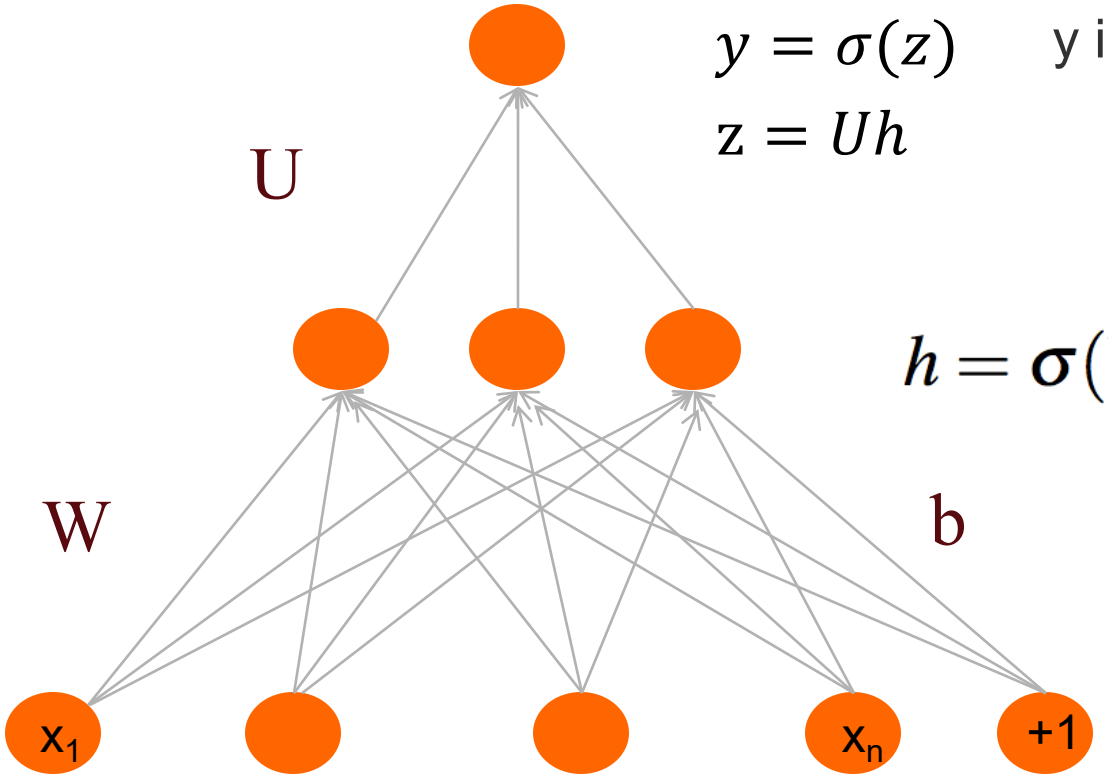


Two-Layer Network with scalar output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)



$y = \sigma(z)$

y is a scalar

$z = Uh$

$h = \sigma(Wx + b)$

Could be ReLU
Or tanh

Two-Layer Network with softmax output

Quiz: What application might need a multinomial classification?

Output layer
(σ node)

$$y = \text{softmax}(z)$$

$$z = Uh$$

y is a vector

U

hidden units
(σ node)

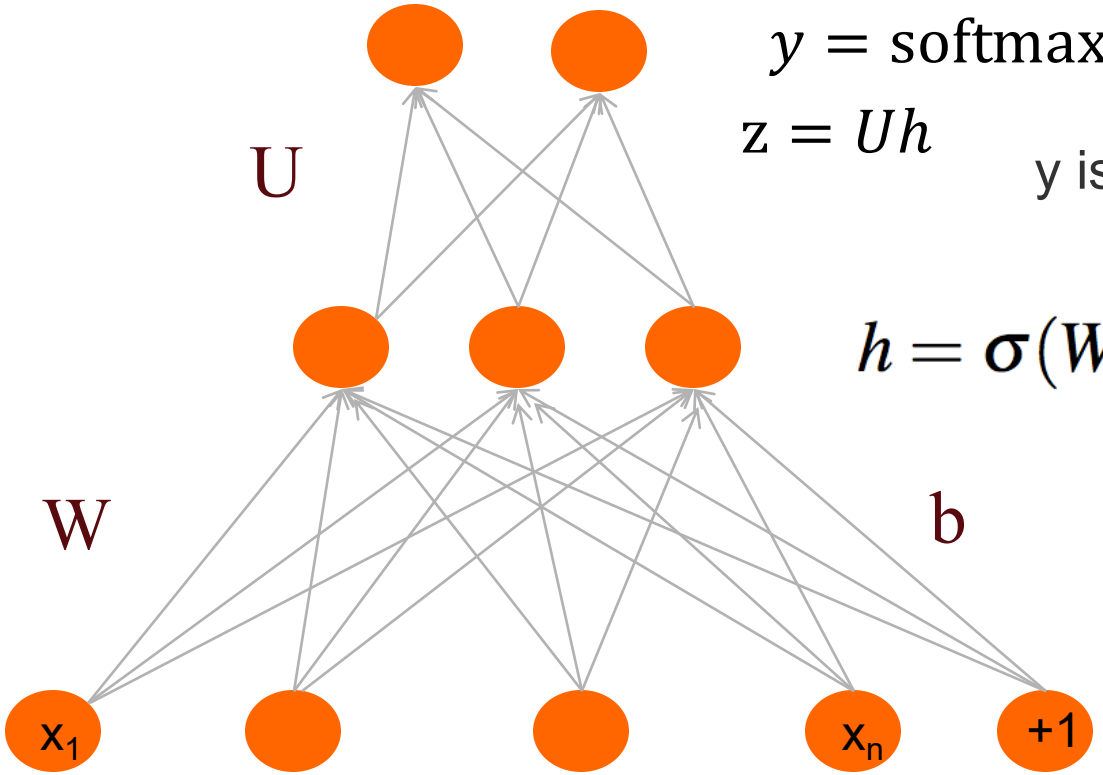
$$h = \sigma(Wx + b)$$

Could be
ReLU
Or tanh

W

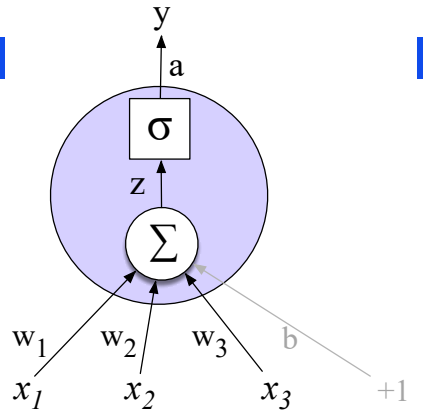
b

Input layer
(vector)



Multi Layer Notation

$$
\begin{aligned}
z^{[1]} &= W^{[1]} a^{[0]} + b^{[1]} \\
a^{[1]} &= g^{[1]}(z^{[1]}) \\
z^{[2]} &= W^{[2]} a^{[1]} + b^{[2]} \\
a^{[2]} &= g^{[2]}(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$



for i **in** 1..n

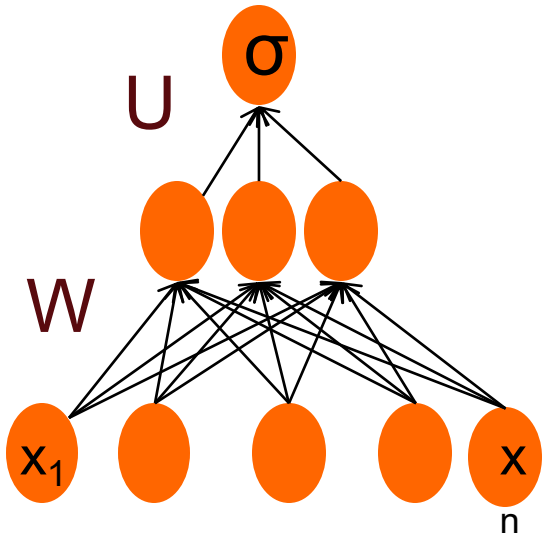
$$
\begin{aligned}
z^{[i]} &= W^{[i]} a^{[i-1]} + b^{[i]} \\
a^{[i]} &= g^{[i]}(z^{[i]}) \\
\hat{y} &= a^{[n]}
\end{aligned}
$$

Applying feedforward networks to NLP tasks

Use case - Text Classification: Sentiment Analysis

- We could do exactly what we did with logistic regression
- Input layer are binary features as before
- Output layer is 0 or 1

State of the art systems use more powerful neural architectures, but simple models are useful to consider!

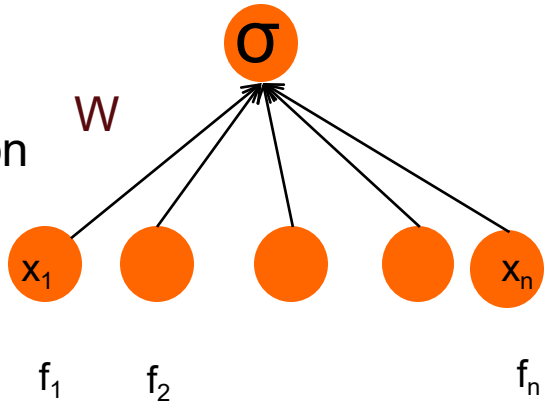


Sentiment Features

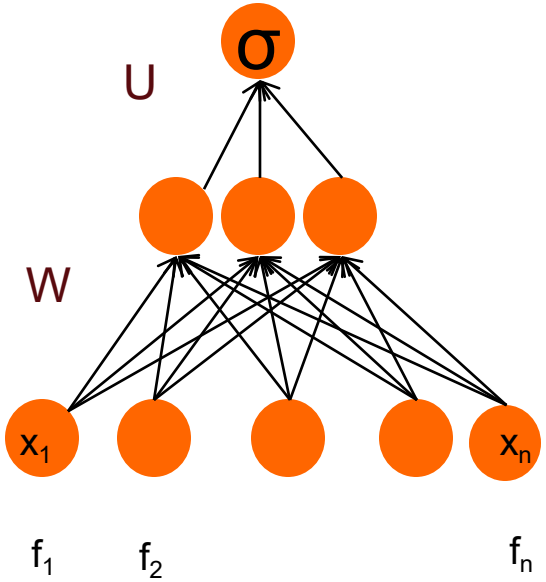
Var	Definition
x_1	count(positive lexicon) $\in$ doc)
x_2	count(negative lexicon) $\in$ doc)
x_3	$\begin{cases} 1 & \text{if } \text{“no”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$
x_4	count(1st and 2nd pronouns $\in$ doc)
x_5	$\begin{cases} 1 & \text{if } \text{“!”} \in \text{doc} \\ 0 & \text{otherwise}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

Feedforward nets for simple classification

Logistic Regression



2-layer feedforward network

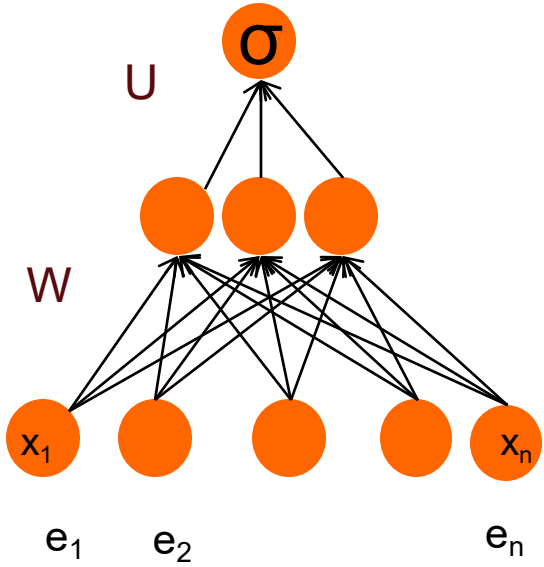


Just adding a hidden layer to logistic regression

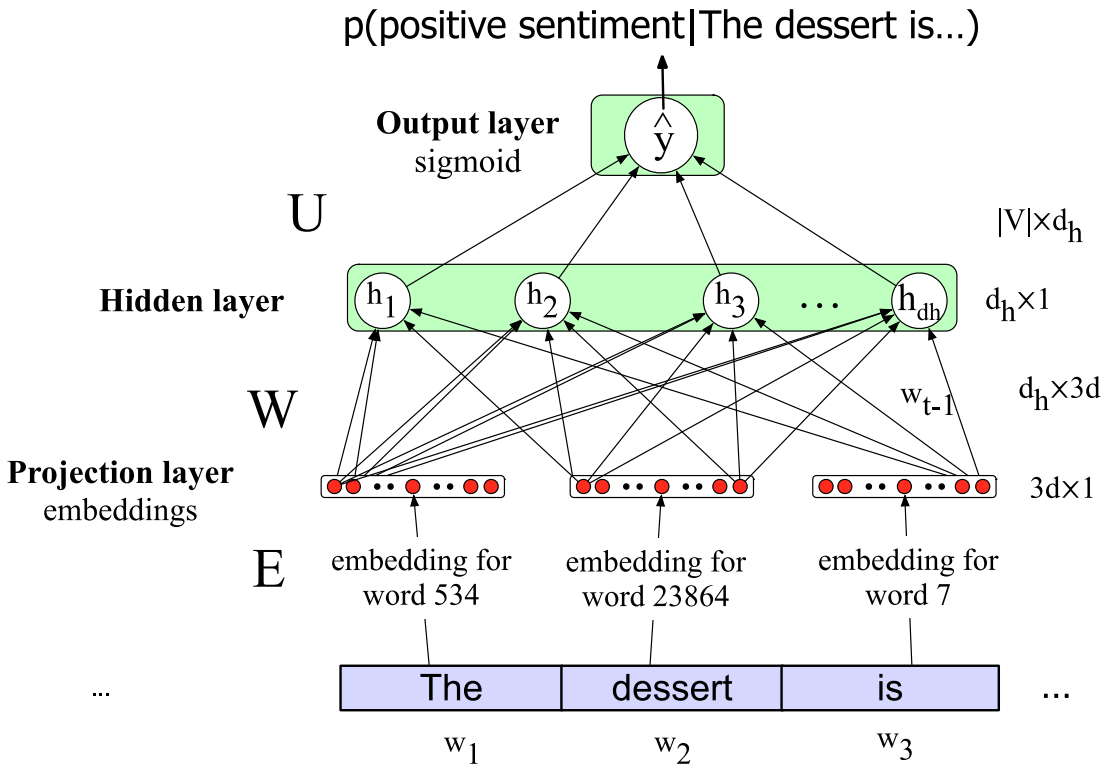
- allows the network to use non-linear interactions between features
- which may (or may not) improve performance

Even better: representation learning

- The real power of deep learning comes from the ability to **learn** features from the data
- Instead of using hand-built human-engineered features for classification
- Use learned representations like embeddings!



Neural Net Classification with embeddings as input features!



How does it look in code?

```
# Neural Network architecture

snn_model = Sequential() #initializes an empty sequential model for a feed forward NN
embedding_layer = Embedding(vocab_length, 100, weights=[embedding_matrix], trainable=False)

snn_model.add(embedding_layer) #Add the embedding layer to the Sequential model. The embedding layer converts word indices into dense vectors of fixed size (100 in t

snn_model.add(Flatten())
snn_model.add(Dense(1, activation='sigmoid'))
```

```
[ ] # Model compiling

snn_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
# Adam is an adaptive learning rate optimization algorithm used to train the model, there are many others (In class, we saw a stochastic gradient descent)
# The binary cross-entropy loss function is used, which is appropriate for binary classification tasks where the output is a probability between 0 and 1. It measures
#The model will track accuracy during training and evaluation

print(snn_model.summary())
```

```
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
embedding (Embedding)	?	9,239,400
flatten (Flatten)	?	0 (unbuilt)
dense (Dense)	?	0 (unbuilt)

```
Total params: 9,239,400 (35.25 MB)
Trainable params: 0 (0.00 B)
Non-trainable params: 9,239,400 (35.25 MB)
None
```

```
[ ] # Model training

snn_model_history = snn_model.fit(X_train, y_train, batch_size=128, epochs=6, verbose=1, validation_split=0.2)

#Breaking down the parameters:
#X_train: The training data (features).
#y_train: The training labels (targets).
#batch_size=128: The number of samples per gradient update. Using a batch size of 128 means the model will update its weights after processing 128 samples.
#epochs=6: The number of times the entire training dataset will pass through the model. In this case, the model will train for 6 epochs.
#verbose=1: Controls the verbosity of the output during training. verbose=1 means progress will be displayed with a progress bar.
#validation_split=0.2: Specifies the fraction of the training data to be used as validation data. Here, 20% of the training data will be used for validation.
```

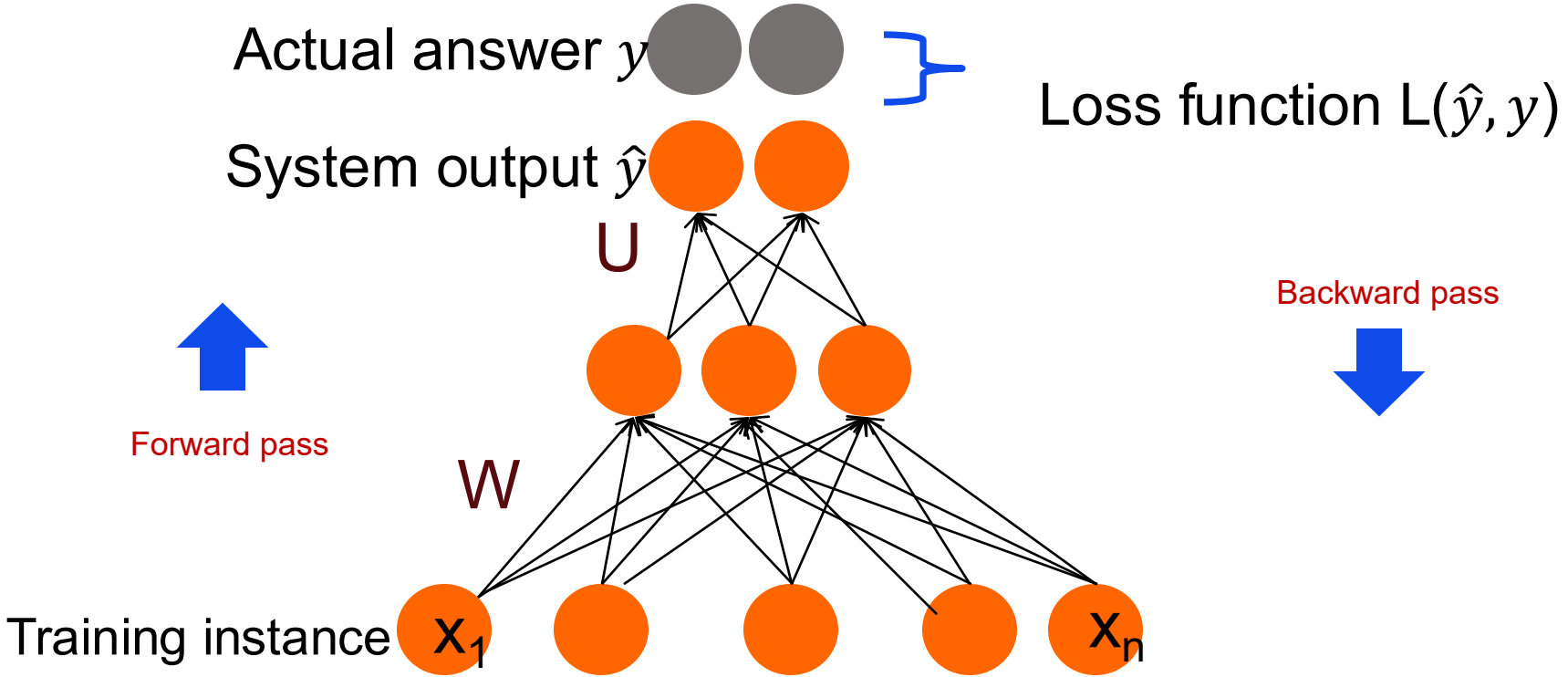
```
Epoch 1/6
250/250 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - acc: 0.6550 - loss: 0.6166 - val_acc: 0.7636 - val_loss: 0.4992
Epoch 2/6
250/250 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - acc: 0.7988 - loss: 0.4429 - val_acc: 0.7669 - val_loss: 0.4908
Epoch 3/6
250/250 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - acc: 0.8250 - loss: 0.4022 - val_acc: 0.7641 - val_loss: 0.5020
Epoch 4/6
```

Which framework to use in Python?

We use the simplest one, Keras (based on Tensorflow) for ease of application, but explore others…

https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article

Intuition: training a 2-layer Network



Intuition: Training a neural network

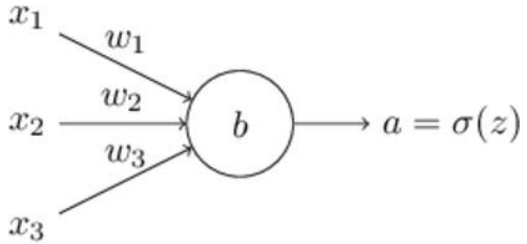
- **Gradient descent** is an optimization algorithm used to train neural networks. It works by calculating the gradient (or slope) of the loss function with respect to each weight in the network, then adjusting those weights to minimize the loss
- **Back propagation** is the process of learning - adjusting the weights and biases of the network's neurons to improve its ability to make accurate decisions

For every training tuple (x, y)

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

Learning with Backpropagation

Task: Update the **weights** and **biases** to decrease **loss function**



Loss (aka cost, objective) function:

$$C = \frac{(y-a)^2}{2}$$

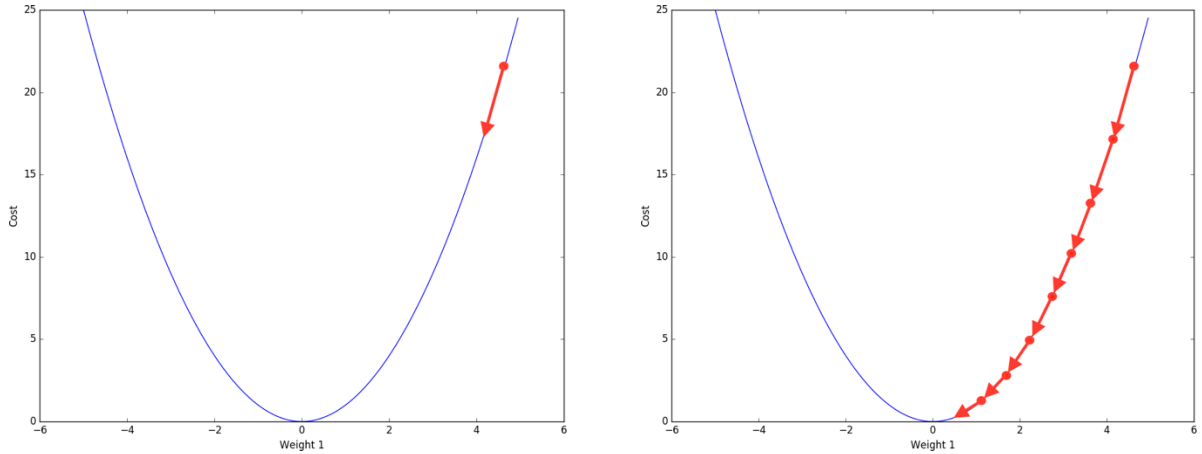
Subtasks:

1. Forward pass to compute network output and "error"
2. Backward pass to compute gradients
3. A fraction of the weight's gradient is subtracted from the weight.

Learning Rate

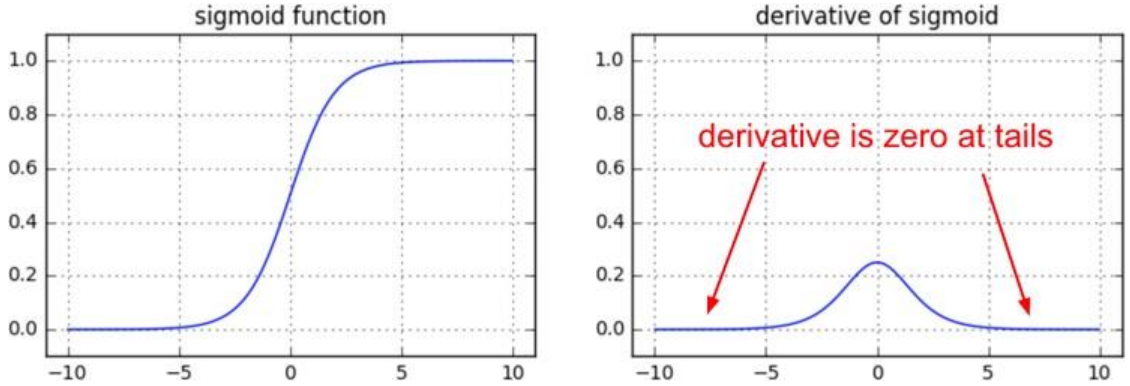
Learning is an Optimization Problem

Task: Update the **weights** and **biases** to decrease **loss function**



Use mini-batch or stochastic gradient descent

Optimization is Hard: Vanishing Gradients (also, exploding gradients)

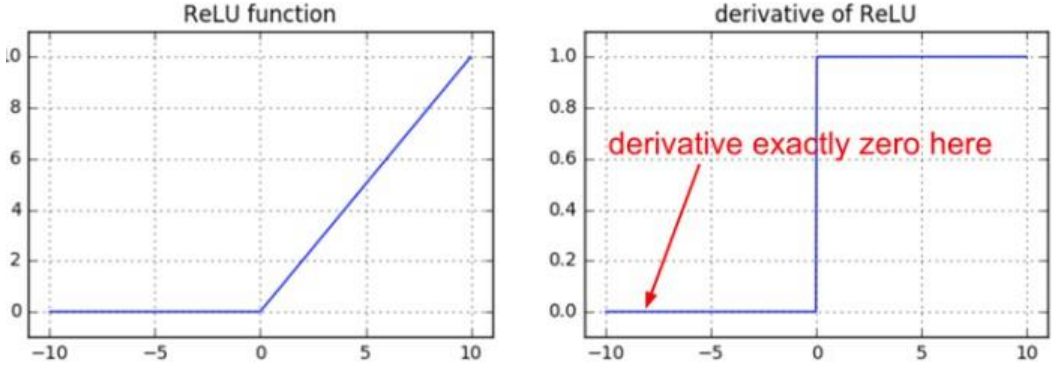


$$\frac{d\sigma(x)}{dx} = (1 - \sigma(x))\,\sigma(x)$$

Partial derivatives are small
= Learning is slow

- As the gradient is backpropagated through many layers, it's multiplied by the derivative of the activation function at each step.
- For sigmoid and tanh functions, these derivatives are always less than 1.
- Multiplying many small numbers results in an extremely small number.
- Consequently, the gradients become vanishingly small for the earlier layers of the network.

Optimization is Hard: Dying ReLUs



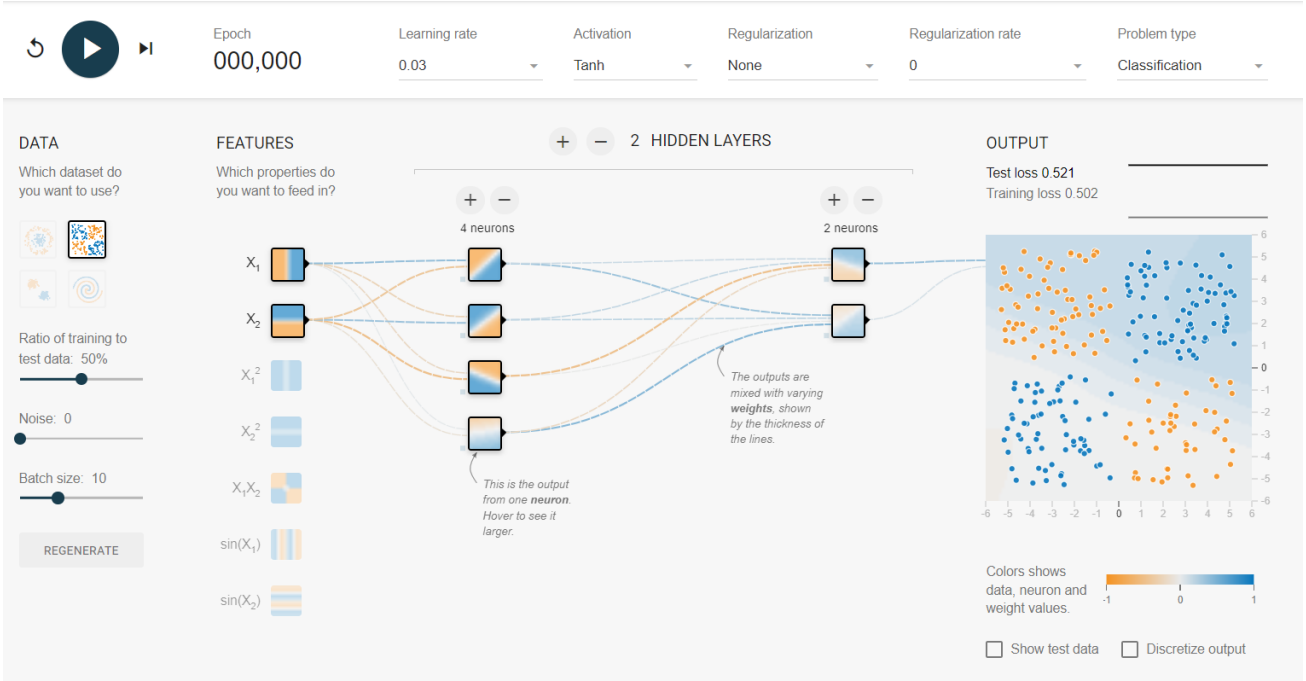
- If a neuron is initialized poorly, it might not fire for entire training dataset
- Large parts of your network could be dead ReLUs!

Recommended read: https://www.deeplearning.ai/ai-notes/initialization/index.html

Tinker with a neural network

Try it out!

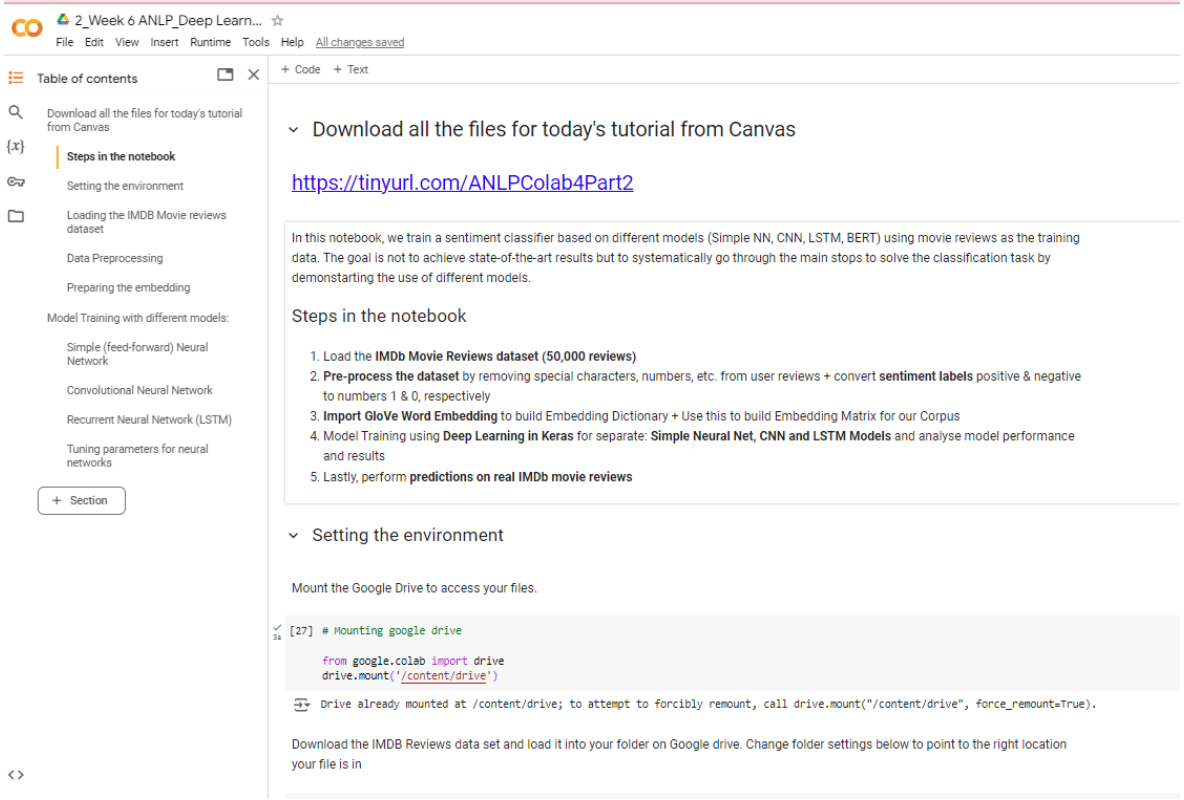
https://playground.tensorflow.org/



Tutorial

Download folder from Canvas

Upload all files to your Google drive to run

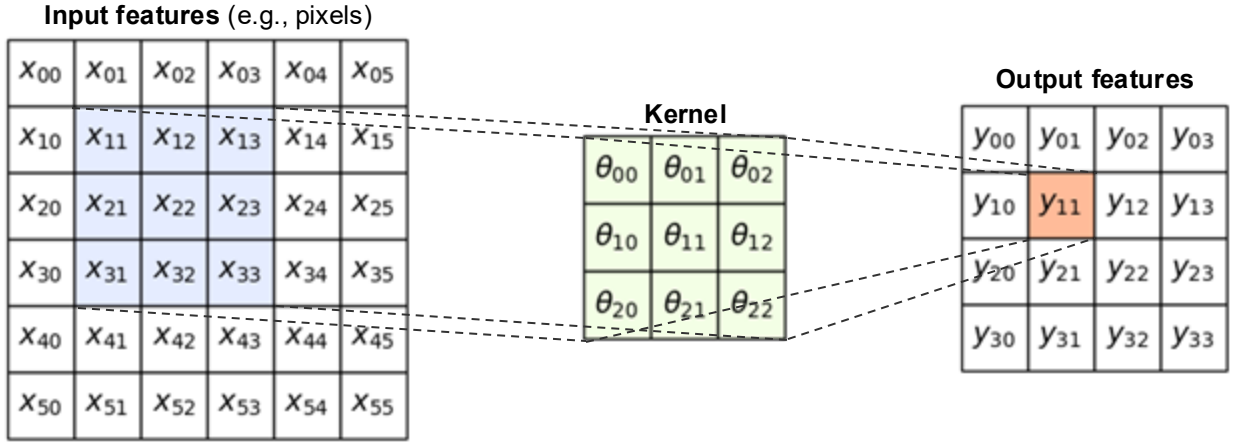


Convolutional Neural Networks (CNN)


or

Basic Concepts — Convolution & Kernel

- Convolution in Image Processing
  - Apply **kernel** (or filter) over each input pixel and its neighbors across the entire image
    (kernel = small weight matrix that gets moved across entire image to generate the output)
  - Purpose: images transformation (e.g., blurring, sharpening, edge detection) -> recognition



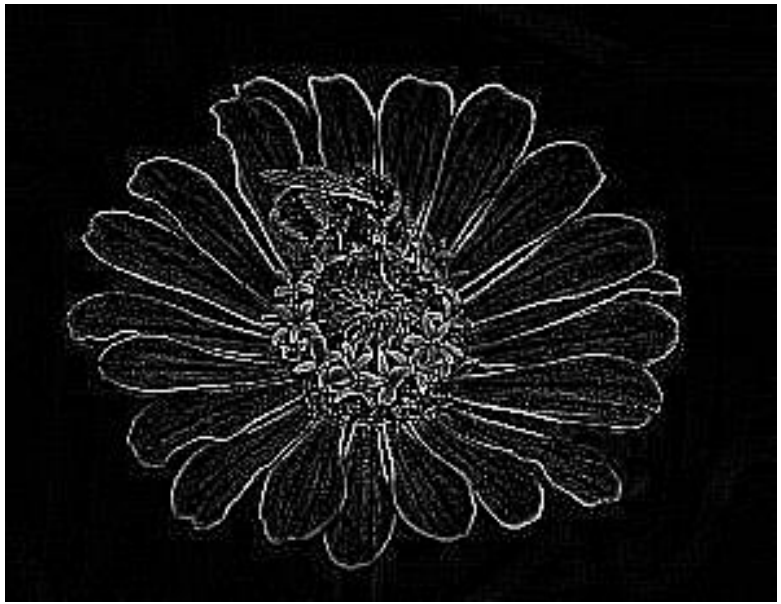
$$y_{11} = \theta_{00}x_{11} + \theta_{01}x_{12} + \theta_{02}x_{13} + \theta_{10}x_{21} + \theta_{11}x_{22} + \theta_{12}x_{23} + \theta_{20}x_{31} + \theta_{21}x_{32} + \theta_{22}x_{33}$$

Example: Blurring

$$\begin{bmatrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{bmatrix}$$

Example: Edge Detection

−1	−1	−1
−1	8	−1
−1	−1	−1



Convolutional Neural Networks (CNNs)

Kernel

θ_{00}	θ_{01}	θ_{02}
θ_{10}	θ_{11}	θ_{12}
θ_{20}	θ_{21}	θ_{22}

- Basic idea of CNNs
  - Do not specify the kernel, but **learn the kernel**!
  - Learn many kernels at the same time
  - Learn θ as usual by minimizing loss function

- Advantages of CNNs (compared to MLPs)
  - Lower number of weights to train (particularly for inputs with large number of features)
  - Consideration spatial relationships ➜ order matters! (at least within a given image region)
  - Spatial/shift invariance (e.g., it does not matter where the bee is located in the image)
  - Feature hierarchy through subsequent CNN layers (low-level ➜ mid-level ➜ high-level)

➜ So where do text documents come in?

CNNs for Text Data

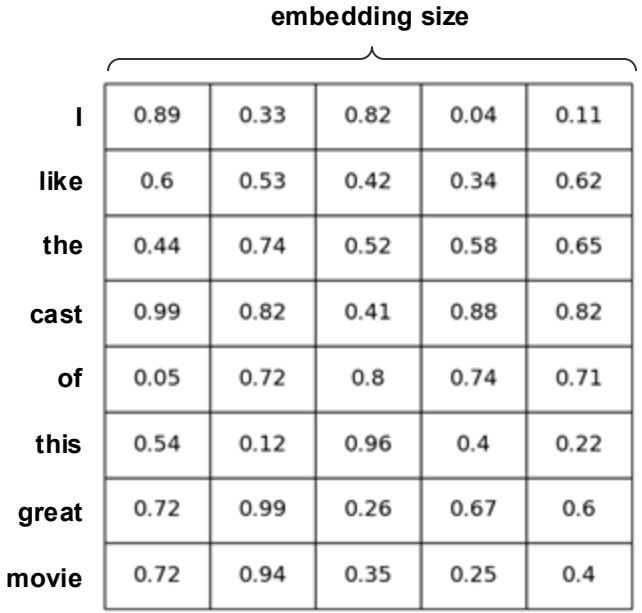
- Text as 2d matrix (like an image)
  - Number of words **x** size of word embeddings

	embedding size				
I	0.89	0.33	0.82	0.04	0.11
like	0.6	0.53	0.42	0.34	0.62
the	0.44	0.74	0.52	0.58	0.65
cast	0.99	0.82	0.41	0.88	0.82
of	0.05	0.72	0.8	0.74	0.71
this	0.54	0.12	0.96	0.4	0.22
great	0.72	0.99	0.26	0.67	0.6
movie	0.72	0.94	0.35	0.25	0.4

Question: What makes the **text matrix** different from an actual image with pixels?

CNNs for Text Data

- Text as 2d matrix (like an image)
  - Number of words **x** size of word embeddings



Important: text matrix (vs. image matrix)

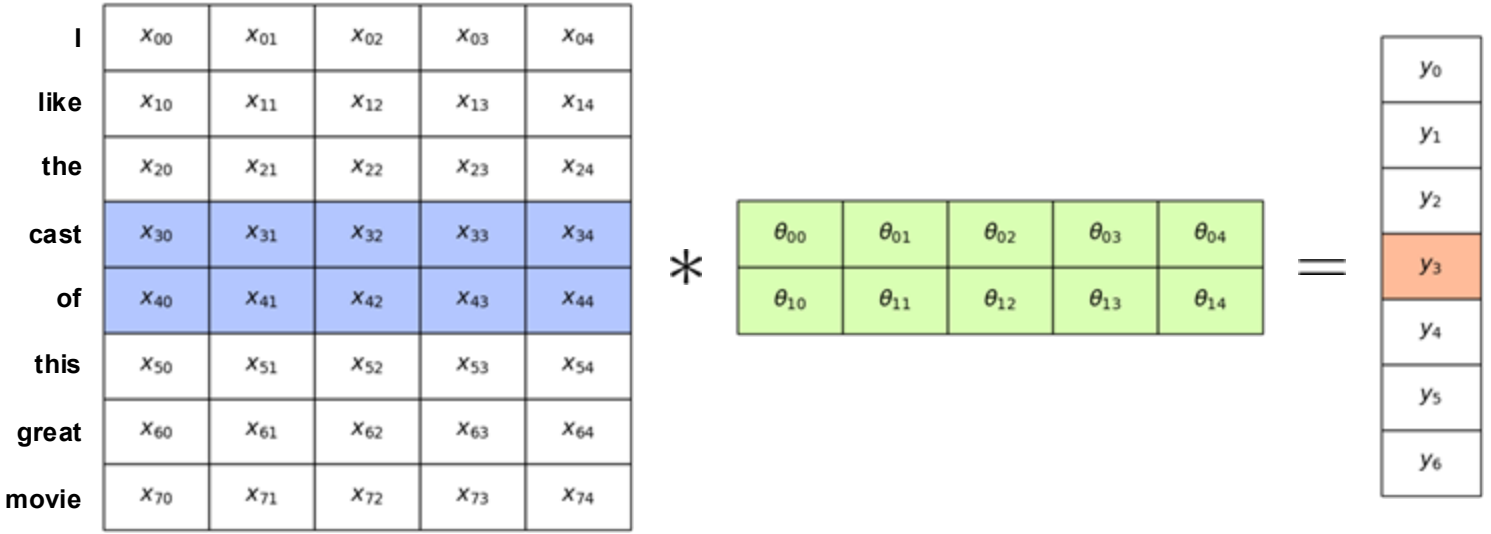
- Spatial relationship only along word dimension (Sequence of the word matters but there's no inherent 2D spatial structure in texts. Also, the embedding dimension is fixed and does not have a spatial relationship like pixels in an image)

- Convolving over embedding dimension meaningless (it would mix the different aspects of word meanings, which does not make sense because each dimension of the embedding has a specific semantic role). The correct approach is to apply convolutions along the sequence of words to capture local dependencies and patterns (e.g., n-grams) – see how in later slides!

➜ **Embeddings size (partially) determines kernel size**

When applying CNNs to text, the kernel size typically spans the entire embedding dimension. This means that if your embedding size is 300, the kernel size will be (filter height, 300)

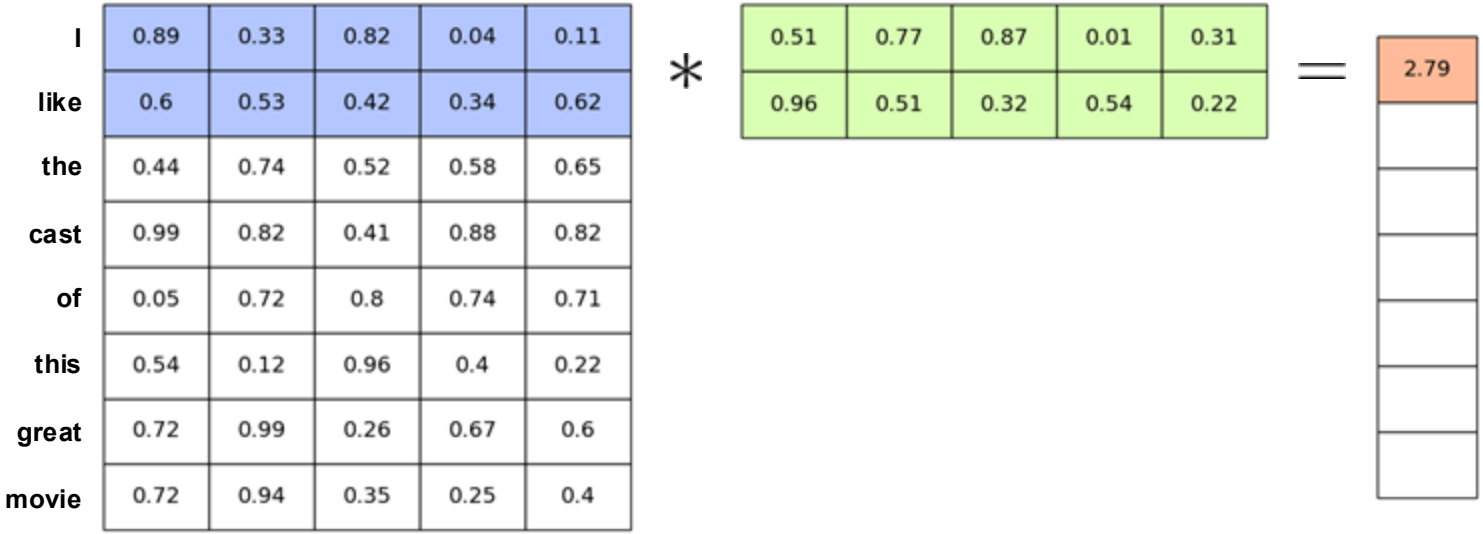
Basic Convolution



$$y_3 = (\theta_{00} \cdot x_{30}) + (\theta_{01} \cdot x_{31}) + (\theta_{02} \cdot x_{32}) + (\theta_{03} \cdot x_{33}) + (\theta_{04} \cdot x_{34}) + (\theta_{10} \cdot x_{40}) + (\theta_{11} \cdot x_{41}) + (\theta_{12} \cdot x_{42}) + (\theta_{13} \cdot x_{43}) + (\theta_{14} \cdot x_{44})$$

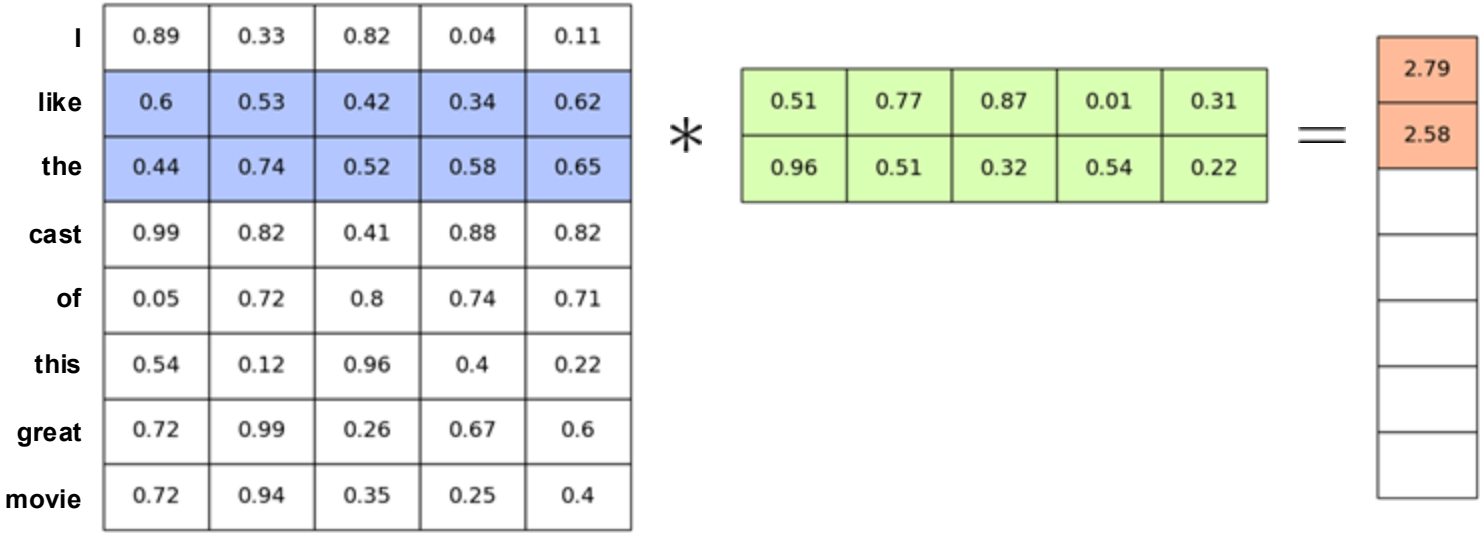
Matrix Multiplication between Image and Kernel is known as ***Convolution Operation***

Basic Convolution — Example



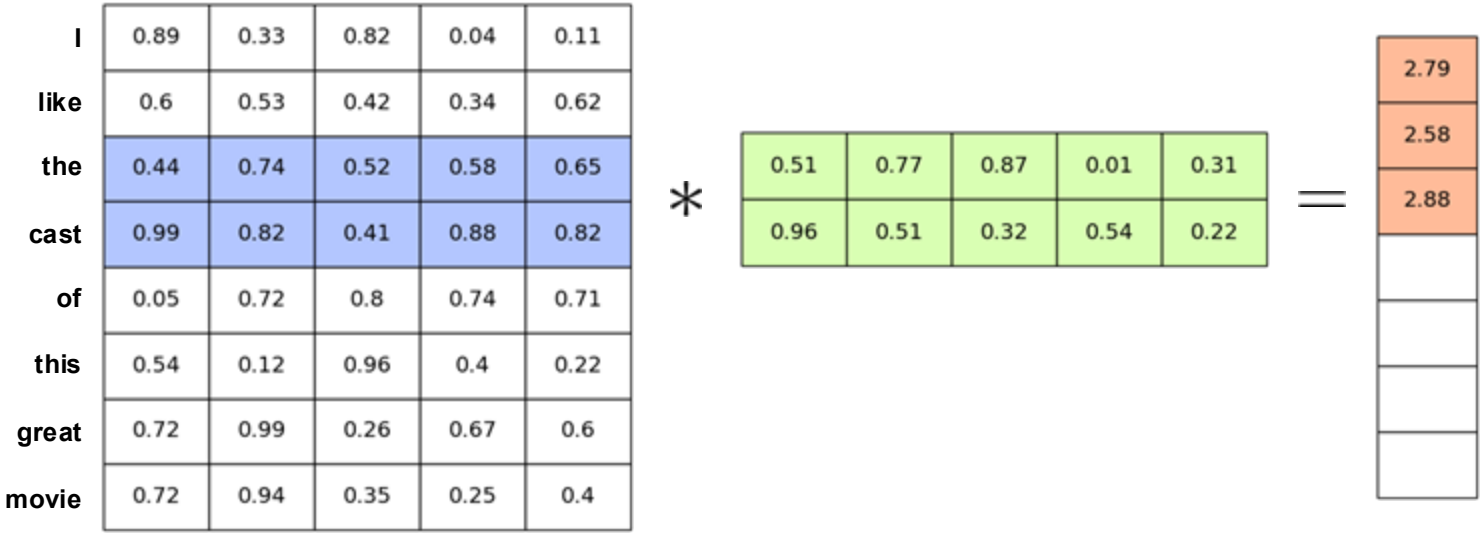
$$
\begin{aligned}
y_0 &= (0.89 \cdot 0.51) + (0.33 \cdot 0.77) + (0.82 \cdot 0.87) + (0.04 \cdot 0.01) + (0.11 \cdot 0.31) \\
&= (0.6 \cdot 0.96) + (0.53 \cdot 0.51) + (0.42 \cdot 0.32) + (0.34 \cdot 0.54) + (0.62 \cdot 0.22) \\
&= \mathbf{2.79}
\end{aligned}
$$

Basic Convolution — Example



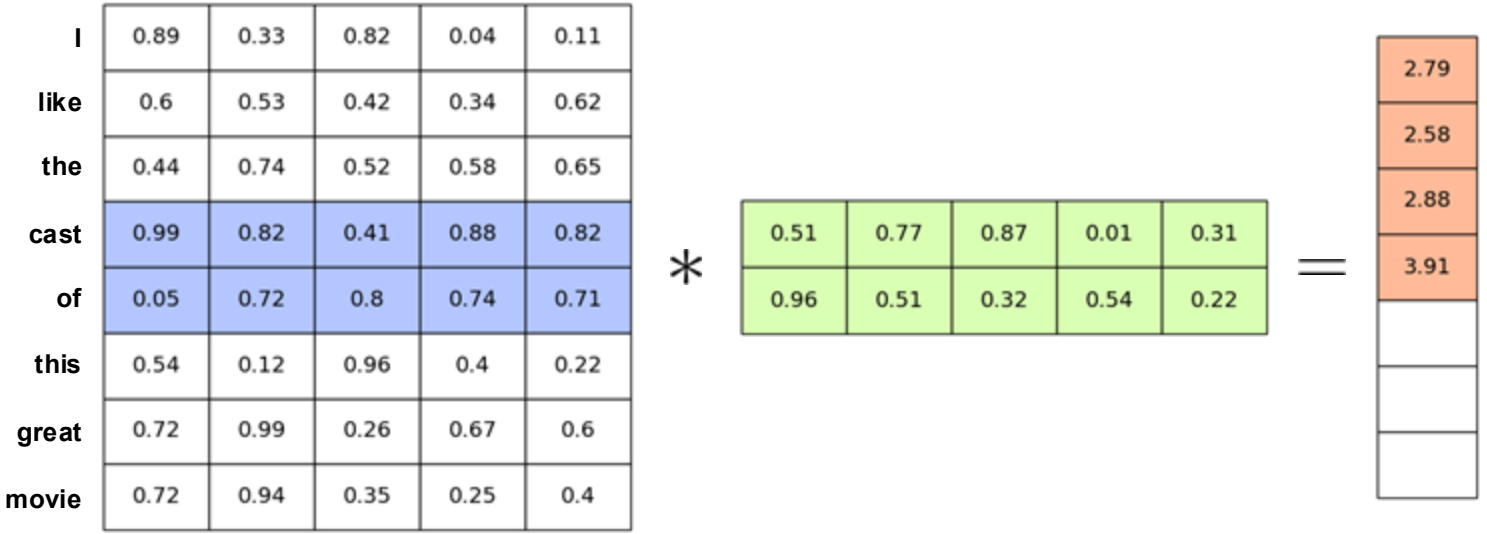
$$
\begin{aligned}
y_1 &= (0.6 \cdot 0.51) + (0.53 \cdot 0.77) + (0.42 \cdot 0.87) + (0.34 \cdot 0.01) + (0.62 \cdot 0.31) \\
&= (0.44 \cdot 0.96) + (0.74 \cdot 0.51) + (0.52 \cdot 0.32) + (0.58 \cdot 0.54) + (0.65 \cdot 0.22) \\
&= \mathbf{2.58}
\end{aligned}
$$

Basic Convolution — Example



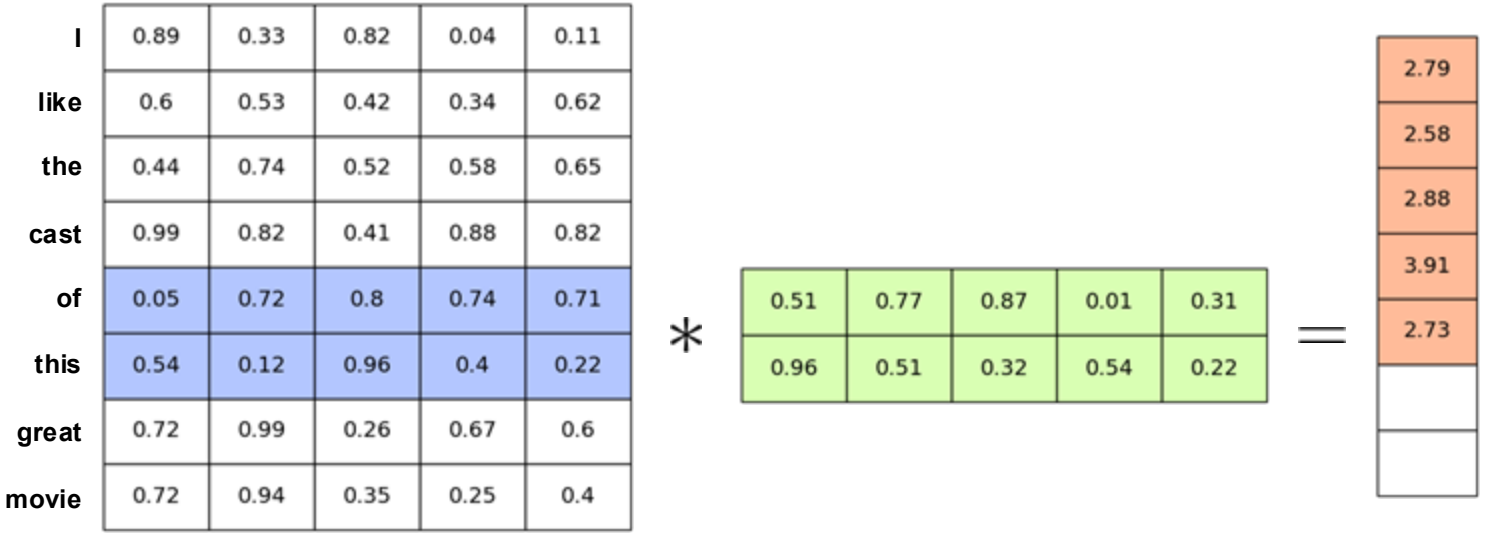
$$
\begin{aligned}
y_2 &= (0.44 \cdot 0.51) + (0.74 \cdot 0.77) + (0.52 \cdot 0.87) + (0.58 \cdot 0.01) + (0.65 \cdot 0.31) \\
&= (0.99 \cdot 0.96) + (0.82 \cdot 0.51) + (0.41 \cdot 0.32) + (0.88 \cdot 0.54) + (0.82 \cdot 0.22) \\
&= \mathbf{2.88}
\end{aligned}
$$

Basic Convolution — Example



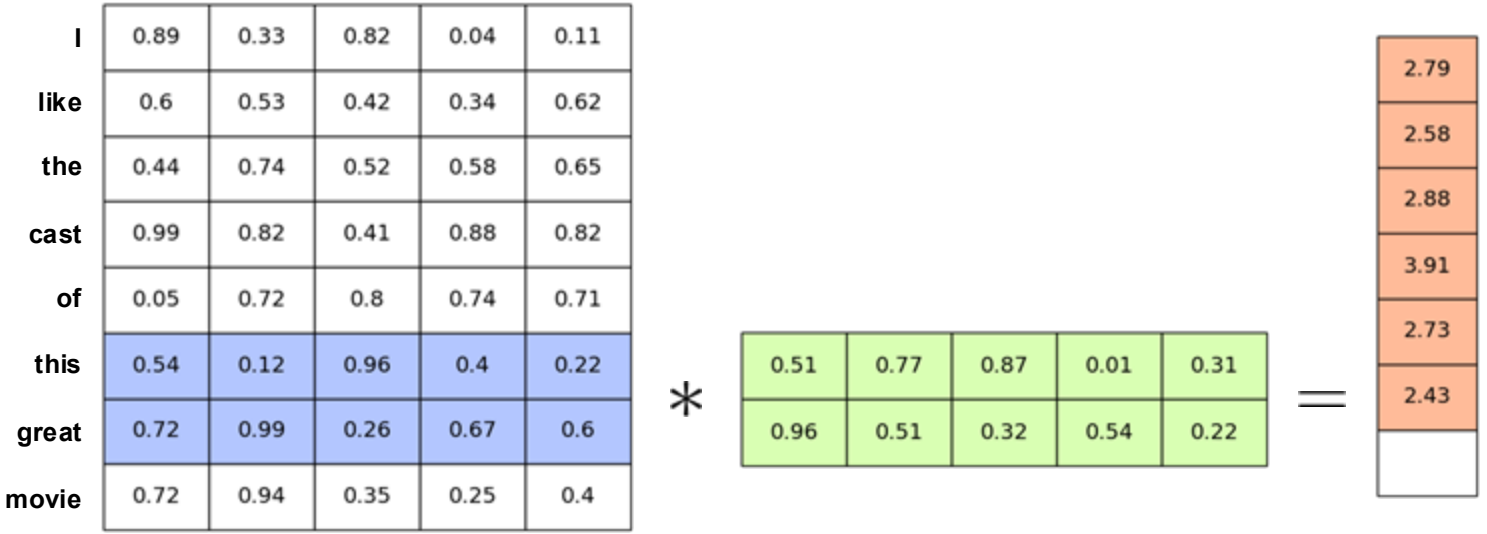
$$
\begin{aligned}
y_3 &= (0.99 \cdot 0.51) + (0.82 \cdot 0.77) + (0.41 \cdot 0.87) + (0.88 \cdot 0.01) + (0.82 \cdot 0.31) \\
&= (0.05 \cdot 0.96) + (0.72 \cdot 0.51) + (0.8 \cdot 0.32) + (0.74 \cdot 0.54) + (0.71 \cdot 0.22) \\
&= \mathbf{3.91}
\end{aligned}
$$

Basic Convolution — Example



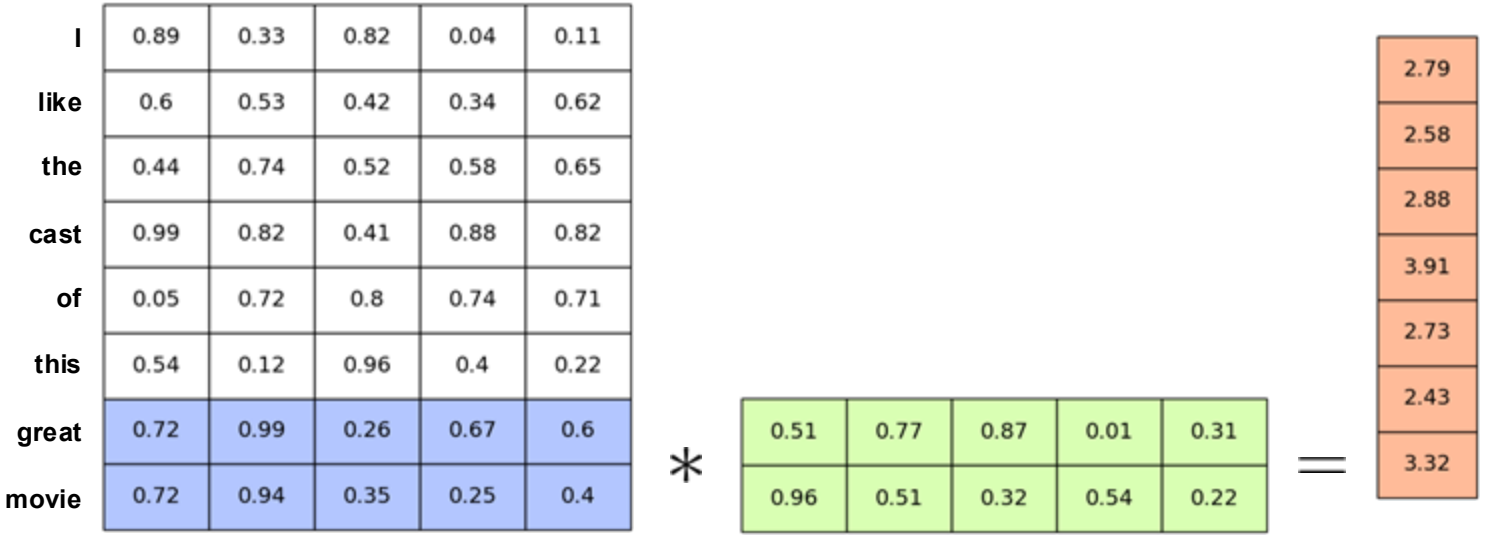
$$
\begin{aligned}
y_4 &= (0.05 \cdot 0.51) + (0.72 \cdot 0.77) + (0.8 \cdot 0.87) + (0.74 \cdot 0.01) + (0.71 \cdot 0.31) \\
&= (0.54 \cdot 0.96) + (0.12 \cdot 0.51) + (0.96 \cdot 0.32) + (0.4 \cdot 0.54) + (0.22 \cdot 0.22) \\
&= \mathbf{2.73}
\end{aligned}
$$

Basic Convolution — Example



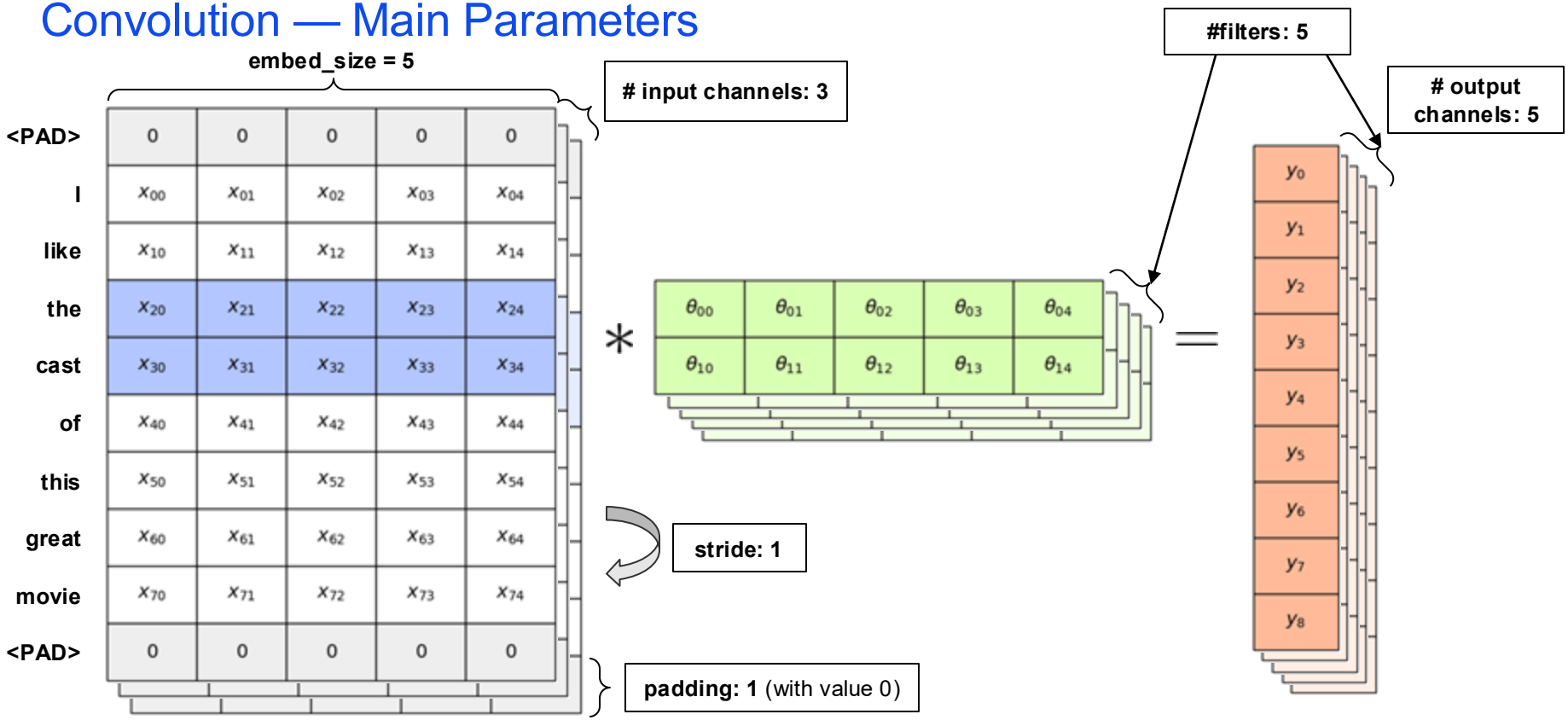
$$
\begin{aligned}
y_5 &= (0.54 \cdot 0.51) + (0.12 \cdot 0.77) + (0.96 \cdot 0.87) + (0.4 \cdot 0.01) + (0.22 \cdot 0.31) \\
&= (0.72 \cdot 0.96) + (0.99 \cdot 0.51) + (0.26 \cdot 0.32) + (0.67 \cdot 0.54) + (0.6 \cdot 0.22) \\
&= \mathbf{2.43}
\end{aligned}
$$

Basic Convolution — Example



$$
\begin{aligned}
y_6 &= (0.72 \cdot 0.51) + (0.99 \cdot 0.77) + (0.26 \cdot 0.87) + (0.67 \cdot 0.01) + (0.6 \cdot 0.31) \\
&= (0.72 \cdot 0.96) + (0.94 \cdot 0.51) + (0.35 \cdot 0.32) + (0.25 \cdot 0.54) + (0.4 \cdot 0.22) \\
&= \mathbf{3.32}
\end{aligned}
$$

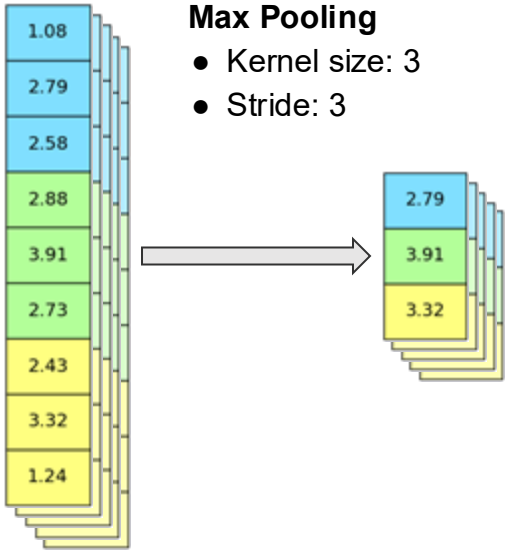
Convolution — Main Parameters

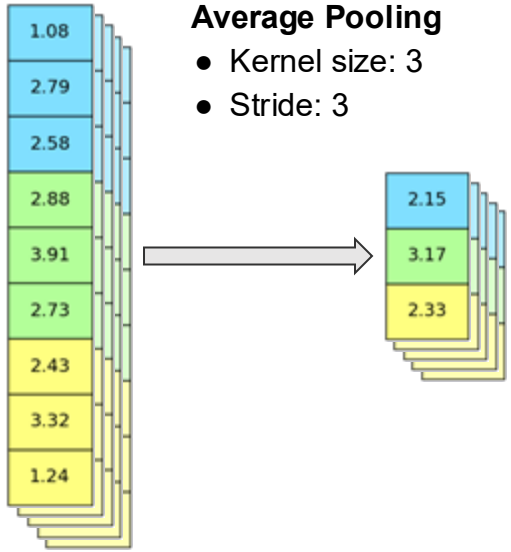



```
conv = nn.Conv2d(in_channels=3, out_channels=5, kernel_size=(2, embed_size), padding=(1, 0), stride=1)
```

Pooling Layer (optional but common)

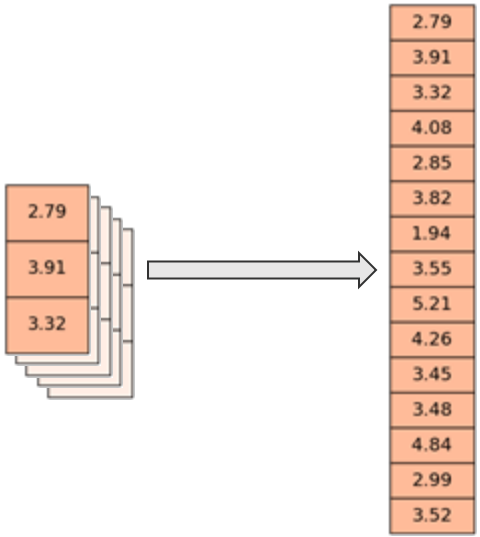
- Pooling (common: **max pooling**, **average pooling**)
  - Reduces the number of features → faster training (fewer trainable parameters)
  - Summarizes features → model more robust against variations in data





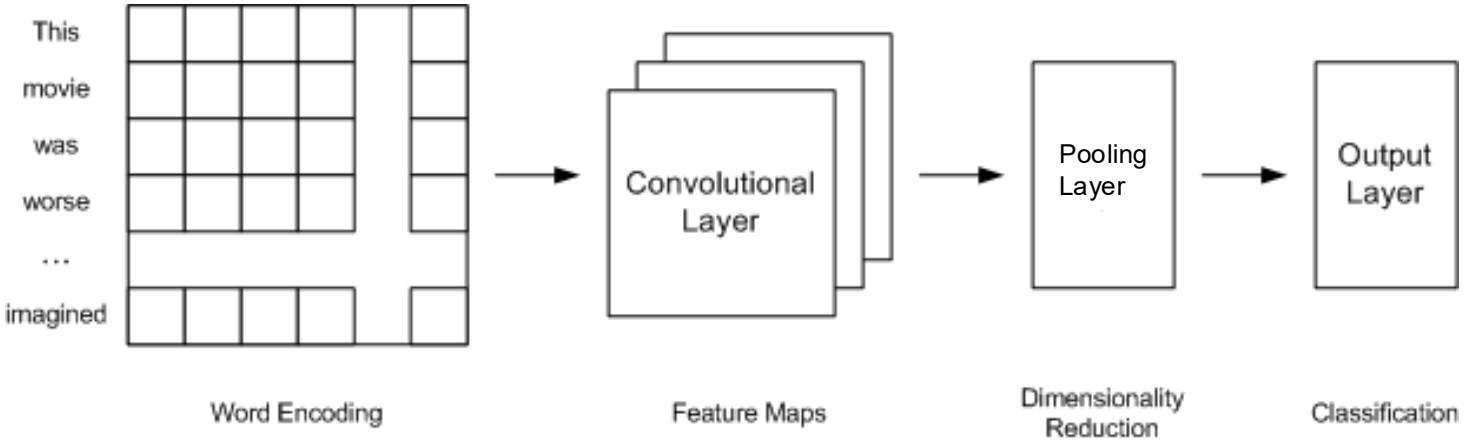
Handling Convolution Output (for further processing)

- Processing of output depending on subsequent layer
  - Input for next next convolution layer (no processing needed: multiple outputs = multiple channels)
  - Input for (most) other types of layers ➜ aggregating/concatenating outputs



- Valid input subsequent linear layer(s)
- Typically the last layers of a architecture

Example: A simple CNN for sentiment analysis



https://developer.ibm.com/learningpaths/get-started-artificial-intelligence/next-steps/beginner-natural-language-processing/

How does it look in code?

Convolutional Neural Network

```
[ ] from keras.layers import Conv1D
```

```
# Neural Network architecture

cnn_model = Sequential()

embedding_layer = Embedding(vocab_length, 100, weights=[embedding_matrix], input_length=maxlen , trainable=False)
cnn_model.add(embedding_layer)


#Adds a 1D convolutional layer with 128 filters, a kernel size of 5, and the ReLU activation function.
cnn_model.add(Conv1D(128, 5, activation='relu'))
#Parameters explained:
#128 filters: The number of output filters in the convolution.
#Kernel size of 5: The size of the convolution window.
#ReLU activation: Applies the ReLU activation function to the output of the convolution

cnn_model.add(GlobalMaxPooling1D()) #This layer performs global max pooling, which reduces the dimensionality by taking th
cnn_model.add(Dense(1, activation='sigmoid')) # Adds a fully connected layer with 1 neuron (output) and sigmoid activation
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is dep
  warnings.warn(
```

```
[ ] # Model compiling

cnn_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
print(cnn_model.summary())
```

```
Model: "sequential_4"
```

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	?	9,239,400
conv1d (Conv1D)	?	0 (unbuilt)
global_max_pooling1d (GlobalMaxPooling1D)	?	0 (unbuilt)
dense_1 (Dense)	?	0 (unbuilt)

```
Total params: 9,239,400 (35.25 MB)
Trainable params: 0 (0.00 B)
Non-trainable params: 9,239,400 (35.25 MB)
None
```

```
[ ] # Model training

cnn_model_history = cnn_model.fit(X_train, y_train, batch_size=128, epochs=6, verbose=1, validation_split=0.2)
```

CNN - Snapshot

Structure: Deep, multi-layer neural networks • Specialized for processing grid-like data (e.g., images)

Function: • Primarily used for image recognition, classification, and computer vision tasks • Can handle complex, non-linear relationships in data

Key components: • Convolutional layers • Pooling layers • Fully connected layers

Operations: • Convolutional layers apply filters to detect features • Pooling layers reduce spatial dimensions • Fully connected layers for final classification

Advantages: • Can automatically learn hierarchical features • Spatial invariance due to shared weights and pooling • Effective at capturing local patterns and global context

Applications: • Image classification • Object detection • Face recognition • Natural language processing (with adaptations)

Practical Considerations

- Common requirement for CNNs: Fixed-length input (let's ignore 1-max pooling!)
  - Length of input sequence affects shape of output of convolutional layer
  - Output serves as input for subsequent layers ➜ shape must be known during model design
  - Basic solution: **enforce fixed-length inputs!**

CUT sequences that are too long

PAD sequences that are too short

Special "word" to indicate padding
(must be part of the vocabulary)

best	*movie*	*ever*				
i	*really*	*liked*	*only*	*the*	*last*	*act*
top	*movie*					
such	*a*	*dumb*	*and*	*silly*	*movie*	
could	*have*	*been*	*much*	*worse*		
the	*story*	*was*	*not*	*that*	*great*	

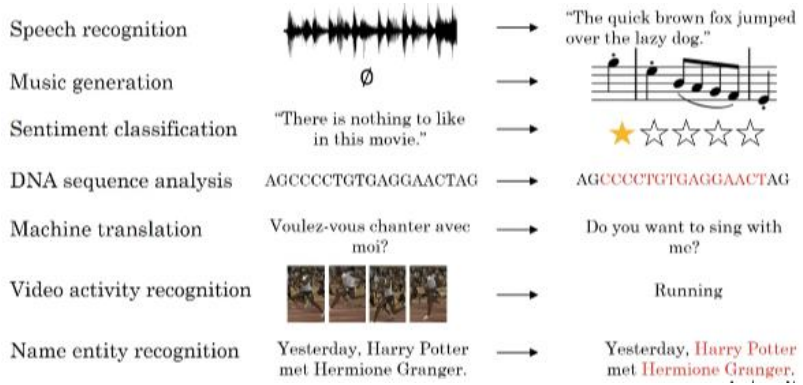
Enforce fixed length of 5

best	*movie*	*ever*	**<PAD>**	**<PAD>**
i	*really*	*liked*	*only*	*the*
top	*movie*	**<PAD>**	**<PAD>**	**<PAD>**
such	*a*	*dumb*	*and*	*silly*
could	*have*	*been*	*much*	*worse*
the	*story*	*was*	*not*	*that*

Recurrent Neural Networks (RNN)

RNNs

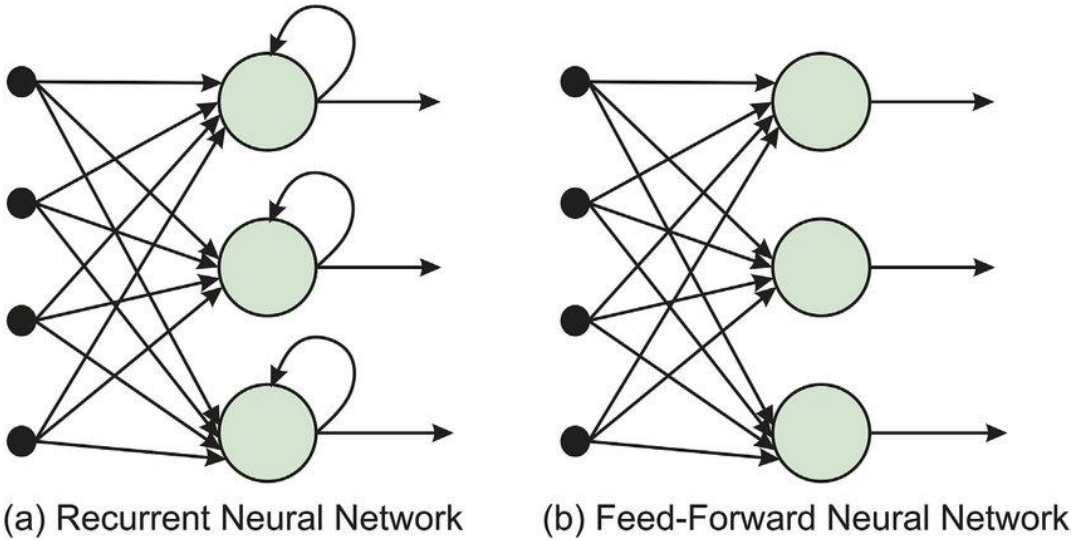
- Recurrent Neural Networks (RNNs) are a class of NN designed to recognize patterns in sequences of data, such as time series, speech, text, or video



- Unlike traditional feedforward neural networks, RNNs have connections that form directed cycles, allowing them to maintain a *memory of previous inputs* and capture *temporal dependencies*

  (in contrast to feed-forward NNs that only go from left to right)

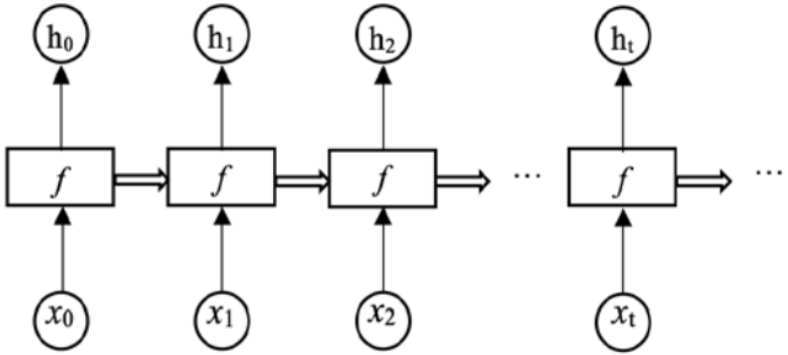
There is a recurring loop that allows data to circle back to the input layer. This means that data is not limited to a feedforward direction!



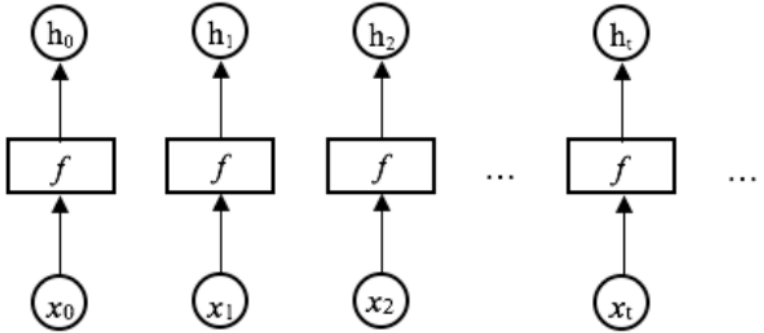
(a) Recurrent Neural Network (b) Feed-Forward Neural Network

RNNs

- RNN passes the activation output from each step to the hidden layer of the next word
- Scans through the data left to right, network parameters shared



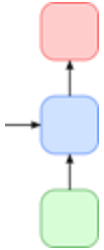
RNN



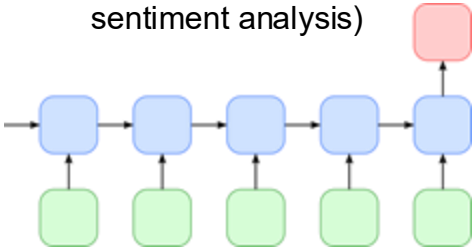
Feedforward NN

RNN architecture — Solving Different Sequence Problems

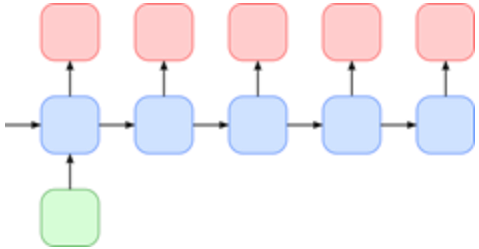
One-to-One
(basically, Feedforward NN)



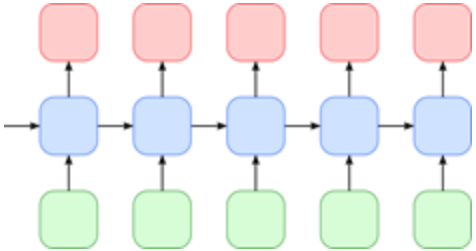
Many-to-One
(e.g., text classification, sentiment analysis)



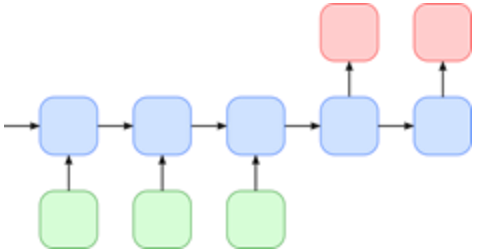
One-to-Many
(e.g., image captioning)



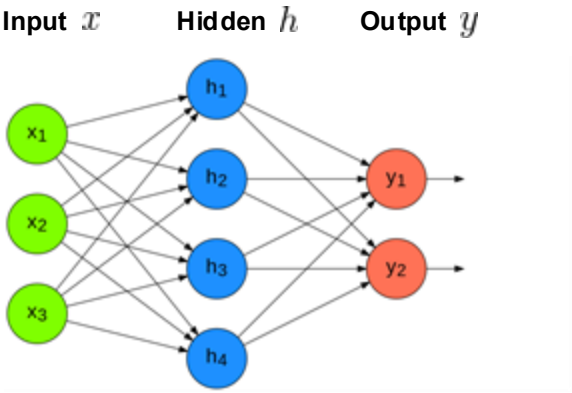
Many-to-Many (sequence labeling)
(e.g., POS tagging, Named Entity Recognition)



Many-to-Many (Many-to-One + One-to-Many)
(e.g., machine translation, summarization)



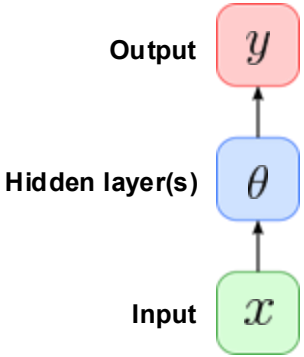
Feedforward NN — Abstraction



$$h = g_h(\theta_h x) \quad , \text{with } \theta_h \in \mathbb{R}^{4\times 3}$$

$$y = g_y(\theta_y h) \quad , \text{with } \theta_y \in \mathbb{R}^{2\times 4}$$

g_h, g_y : suitable activation functions



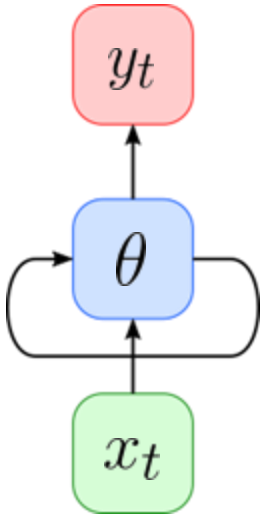
Abstraction

- Represent all units of a layer as one box
- In the following: 1 hidden layer

Recurrent Neural Network — Basic Idea

Feedforward NN

Recurrent NN



x is now a sequence of vectors (e.g., word embeddings)

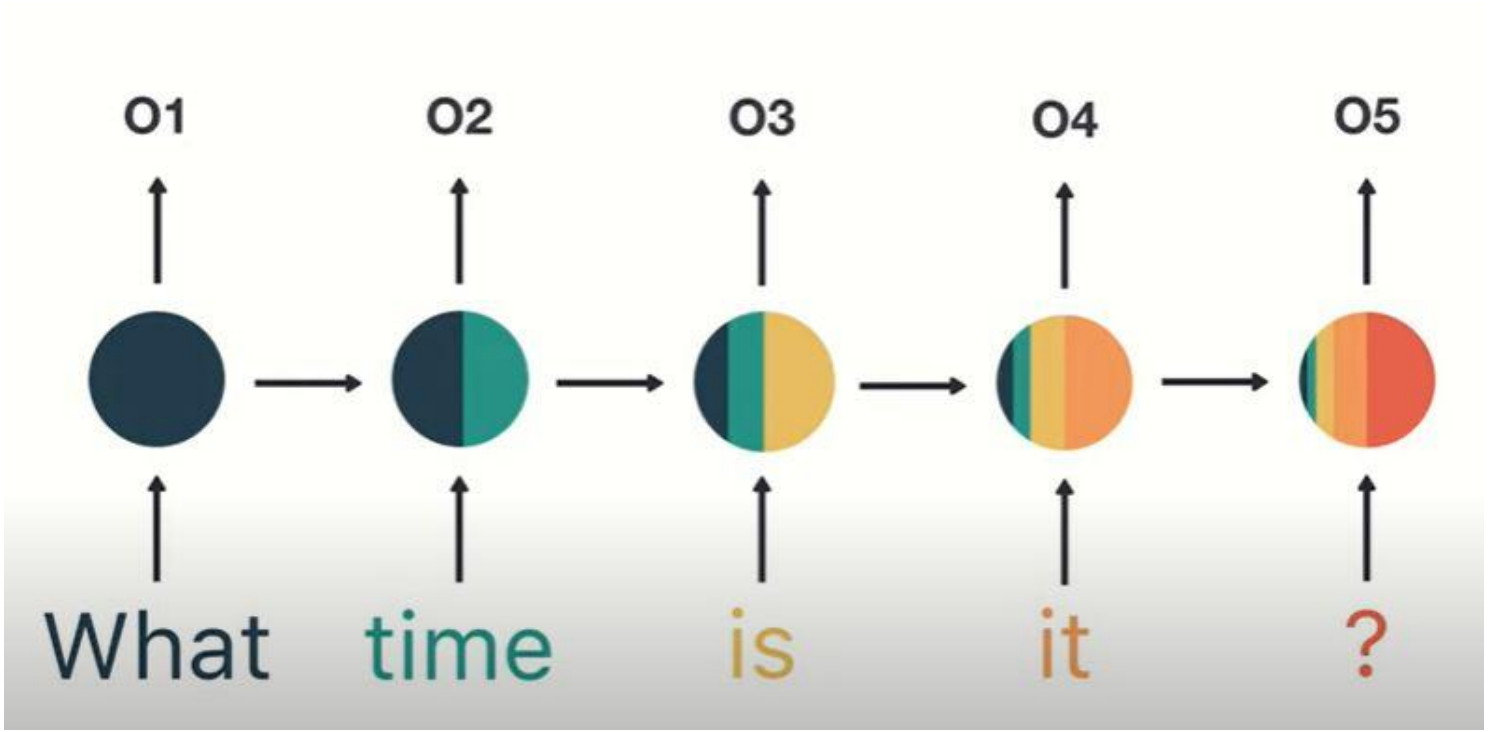
Core concept of RNNs: **Hidden State**

- Additional vector incorporated into the network
- Commonly holds the last output of the hidden layer
  ➜ size of hidden state = size of hidden layer
- Randomly initialized, and to be tuned through training (➜ backpropagation)
- Basic recurrent formula:

$$h_t = f_\theta(h_{t-1}, x_t)$$

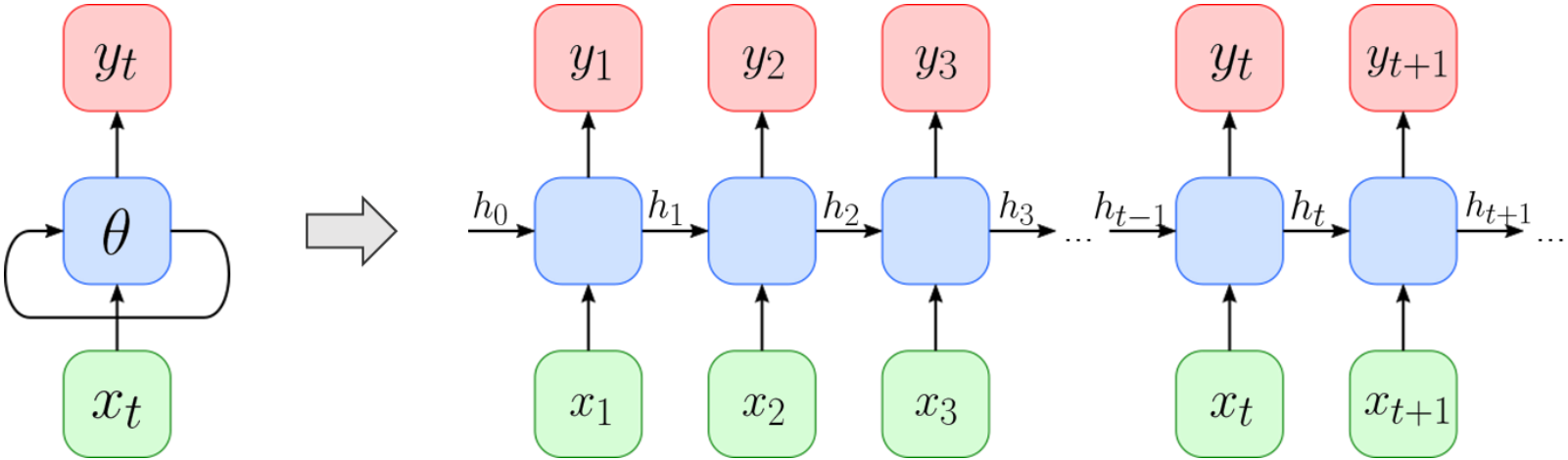
hidden state of time step $t-1$

input vector at time step t



From the pre-reading materials:
https://www.youtube.com/watch?v=LHXXI4-IEns&ab_channel=TheAIHacker

RNN — Unrolled Representation

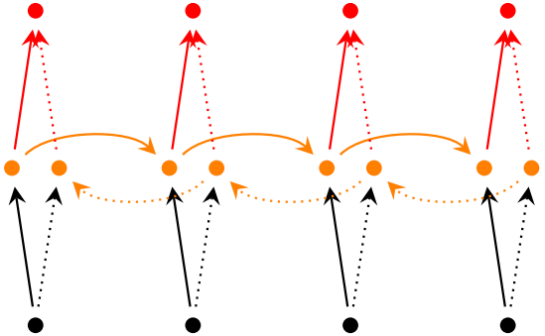


Memory →

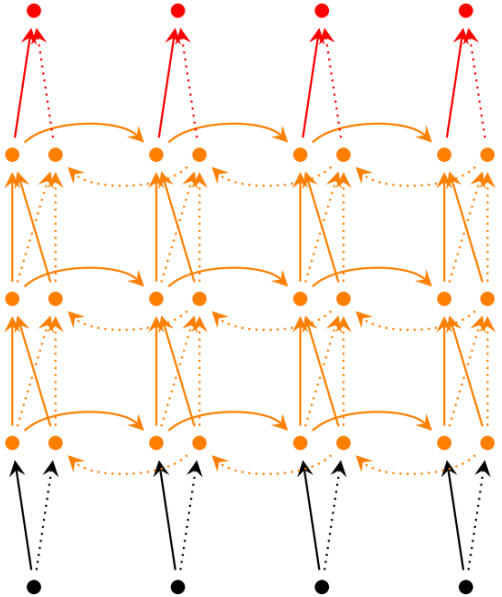
- **Input *(x)*:** (example: word of a sentence)
- **Hidden layer (h)/ state:** function of previous hidden state and new input
- **Output *(y)*:** (example: predict next word in the sentence)

RNN Variants: Bidirectional RNNs

(Shallow)



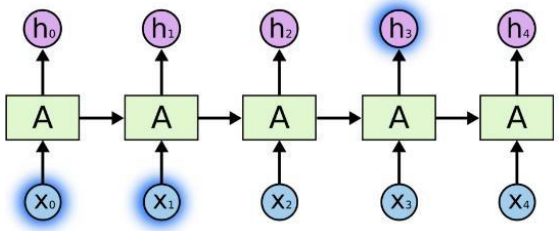
(Deep)



- Example:
  - Filling in missing words
- Deeper =
  - more learning capacity
  - but needs lots of training data

Long-Term Dependency

There is still a problem!



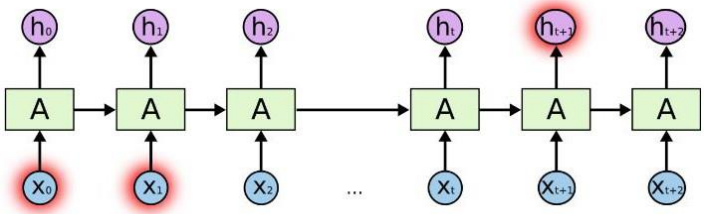
- Short-term dependence:

  Bob is eating an **apple.**

- Long-term dependence:

  Bob likes **apples**. He is hungry and decided to have a snack. So now he is eating an **apple.**

Context ⟶



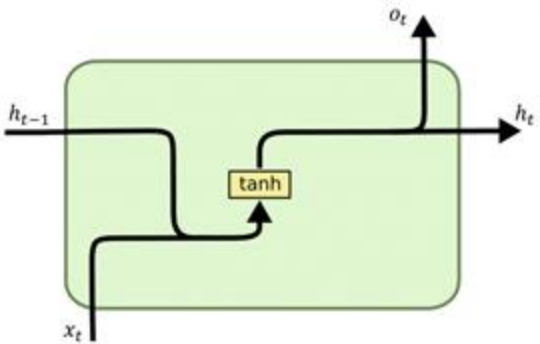
In theory, vanilla RNNs can handle arbitrarily long-term dependence.

In practice, it's difficult.

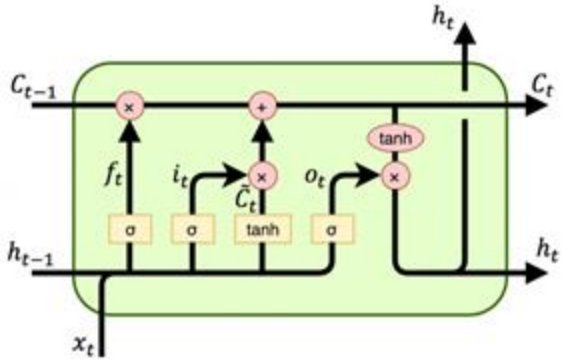
Beyond Vanilla RNN — LSTM & GRU

Use those in practice!

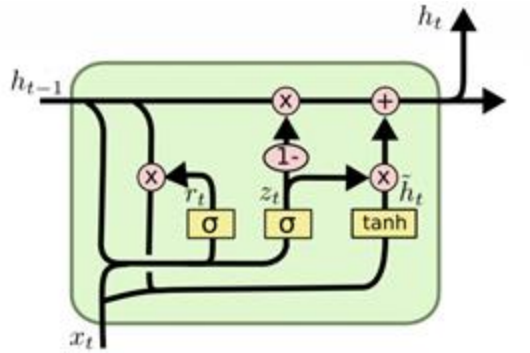
Vanilla RNN



LSTM (Long Short-Term Memory)



GRU (Gated Recurrent Unit)

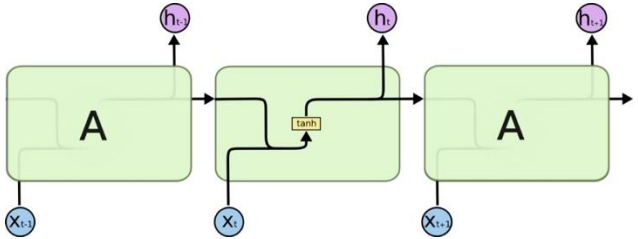


- Observation — Motivation
  - Vanilla RNN struggle with very long-range dependencies
  - LSTMs and GRUs improve on that

Long Short Term Memory (LSTM) - Extended topic

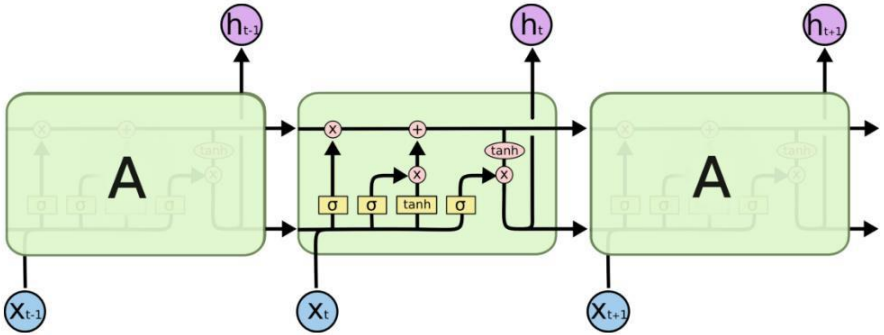
Long Short Term Memory (LSTM) Networks

Vanilla RNN:



Let's use the boxed '**Conveyor belt**' notation

LSTM:

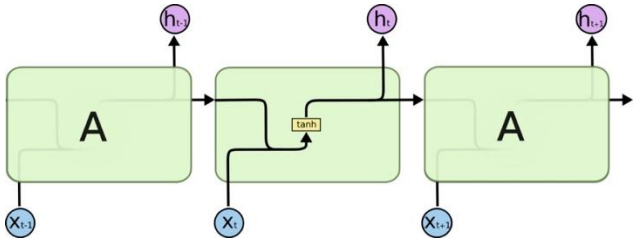


LSTM has additional gates!

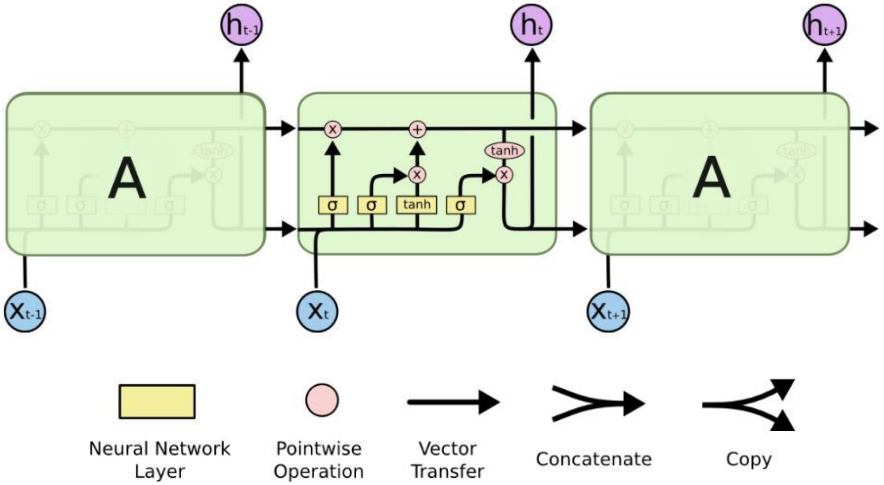
Video Lecture: [MIT 6.S094: Recurrent Neural Networks for Steering Through Time]

LSTM: Gates Regulate

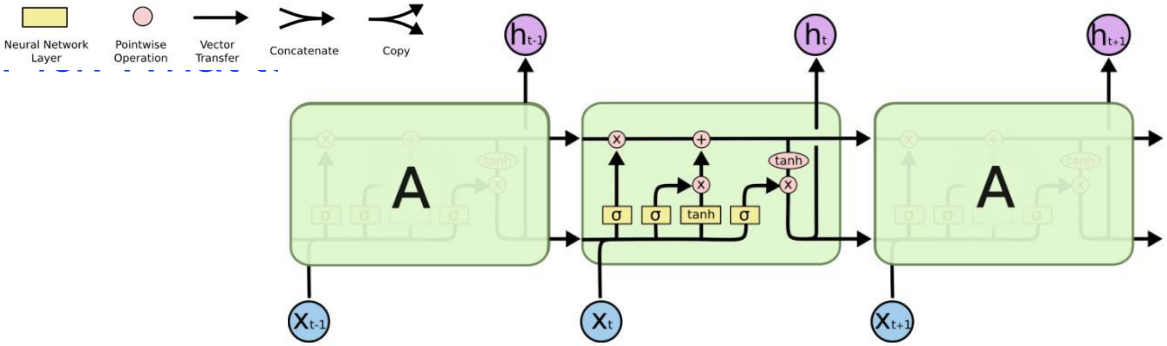
Vanilla RNN:



LSTM:



LSTM:

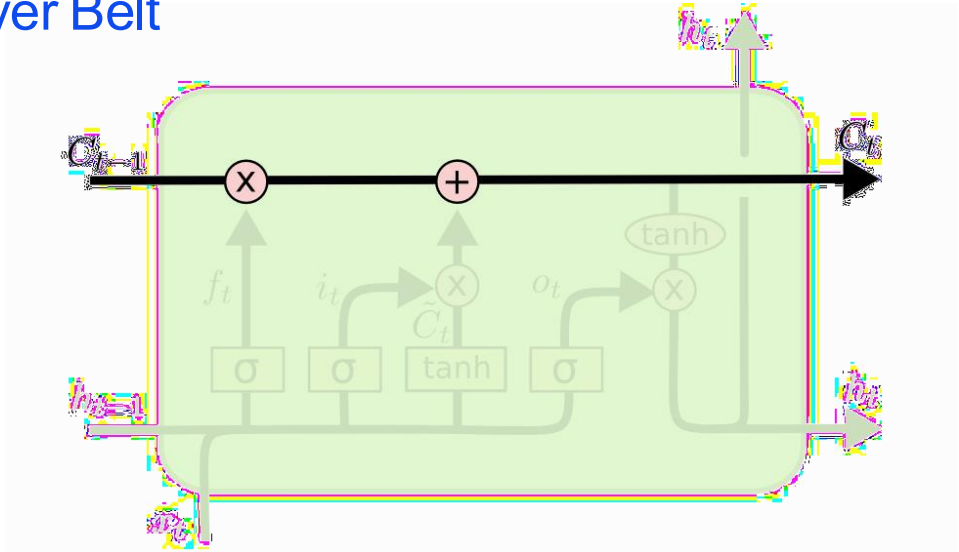


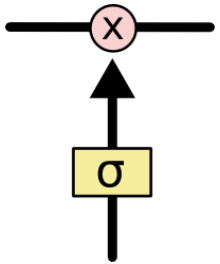
Bob and Alice are having lunch. Bob likes apples. Alice likes oranges.
She is eating an orange.

Conveyer belt for **previous state** and **new data**:

1. Decide what to forget (state)
2. Decide what to remember (state)
3. Decide what to output (if anything)

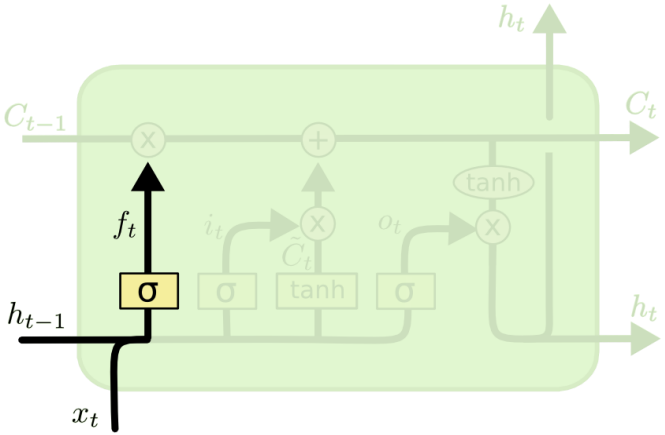
LSTM Conveyer Belt





- State run through the cell
- 3 sigmoid layers output deciding which information is let through (~1) and which is not (~0)

LSTM Conveyer Belt

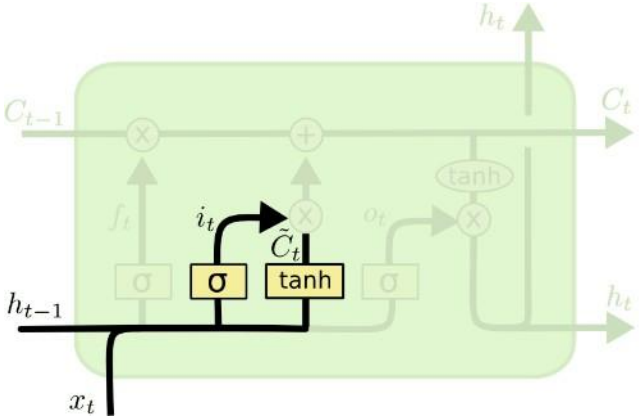


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

First sigmoid

Step 1: Decide what to forget / ignore

LSTM Conveyer Belt

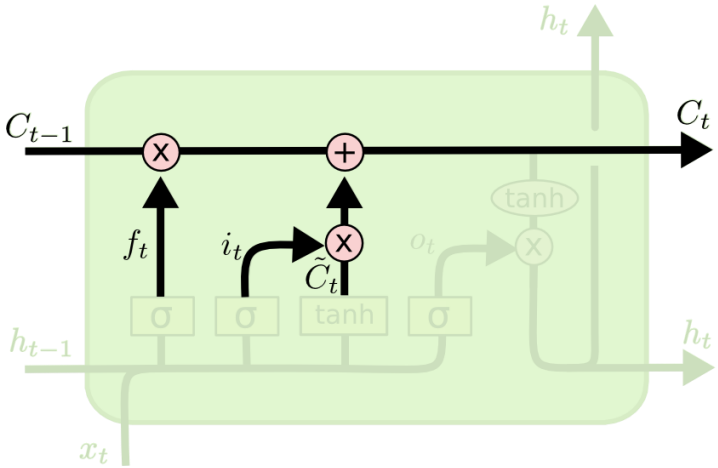


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$ Second sigmoid

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Step 2: Decide which state values to update (w/ sigmoid) and what values to update with (w/ tanh)

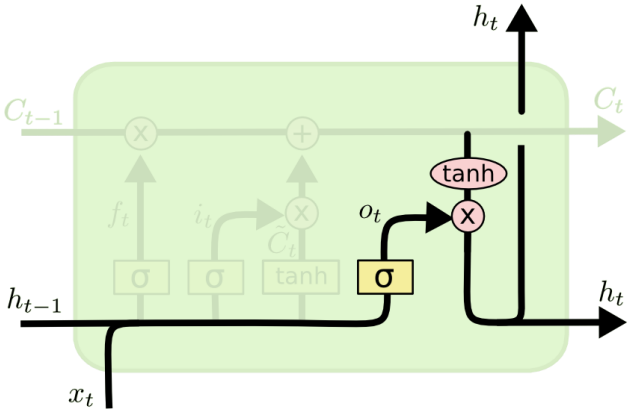
LSTM Conveyer Belt



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Step 3: Perform the forgetting and the state update

LSTM Conveyer Belt



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$ Third sigmoid

$$h_t = o_t * \tanh\left(C_t\right)$$

Step 4: Produce output with tanh [-1, 1] deciding the values and sigmoid [0, 1] deciding the filtering

Tutorial

In the same notebook from before, explore:

CNN

RNN (LSTM)

Recurrent Neural Network (LSTM)

```
[54] from keras.layers import LSTM
```

```
[55] # RNN architecture

     lstm_model = Sequential()
     embedding_layer = Embedding(vocab_length, 100, weights=[embedding_matrix], input_length=maxlen , trainable=False)

     lstm_model.add(embedding_layer)
     lstm_model.add(Bidirectional(LSTM(128, dropout=0.2, recurrent_dropout=0.2, kernel_regularizer=l2(0.01)))) #Adds a bidirect
     #Parameters explained:
     #Bidirectional: Wraps the LSTM layer to process the input in both forward and backward directions.
     #128 units: The number of LSTM units.
     #dropout=0.2: Dropout rate for the input connections.
     #recurrent_dropout=0.2: Dropout rate for the recurrent connections.
     #kernel_regularizer=l2(0.01): L2 regularization to prevent overfitting.

     lstm_model.add(Dense(64, activation='relu', kernel_regularizer=l2(0.01))) #Adds a fully connected layer with 64 neurons.
     #64 neurons: The number of neurons in the dense layer.
     #activation='relu': Uses the ReLU activation function.
     #kernel_regularizer=l2(0.01): L2 regularization to prevent overfitting.

     lstm_model.add(Dropout(0.2)) # Adds a dropout layer with a dropout rate of 0.2 to prevent overfitting.
     lstm_model.add(Dense(1, activation='sigmoid')) #Adds a fully connected layer with 1 neuron and a sigmoid activation functi
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is dep
  warnings.warn(
```

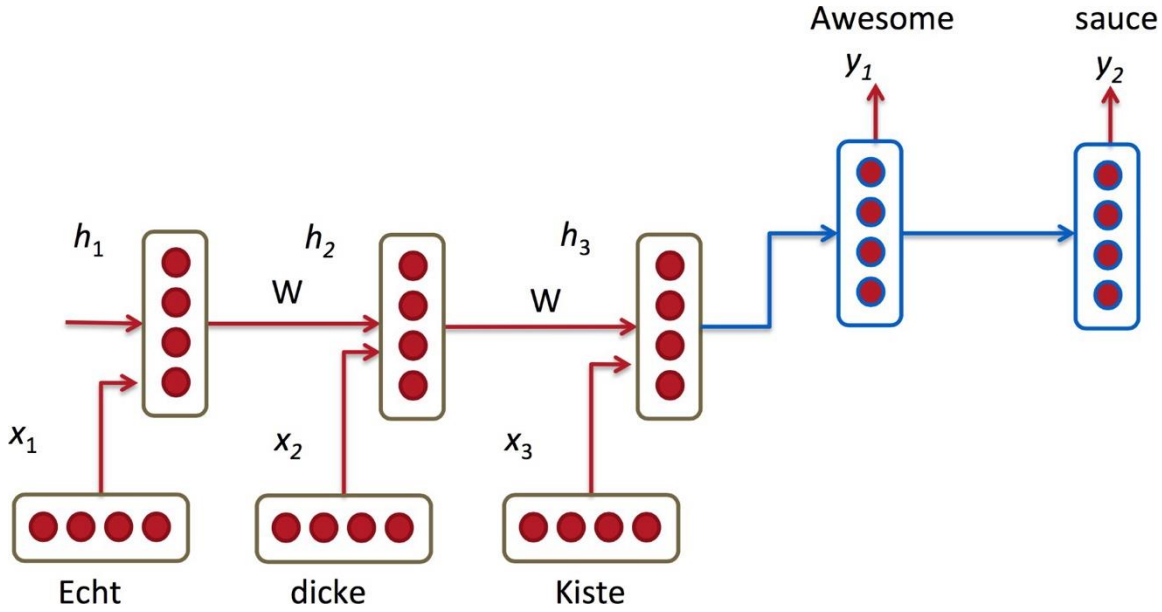
```
[56] # Model compiling

     lstm_model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
     print(lstm_model.summary())
```

```
Model: "sequential_2"
```

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	?	9,239,400
bidirectional_2 (Bidirectional)	?	0 (unbuilt)
dense_4 (Dense)	?	0 (unbuilt)
dropout_2 (Dropout)	?	0 (unbuilt)
dense_5 (Dense)	?	0 (unbuilt)

```
Total params: 9,239,400 (35.25 MB)
Trainable params: 0 (0.00 B)
Non-trainable params: 9,239,400 (35.25 MB)
```

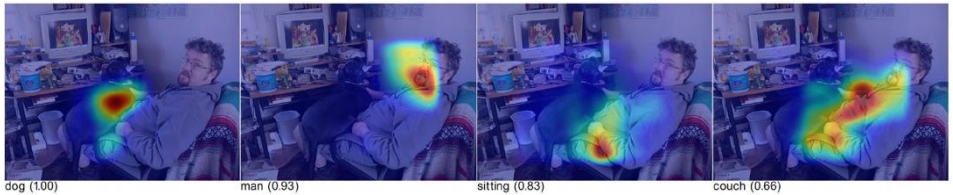
Application: Machine Translation



Application: Image Caption Generation

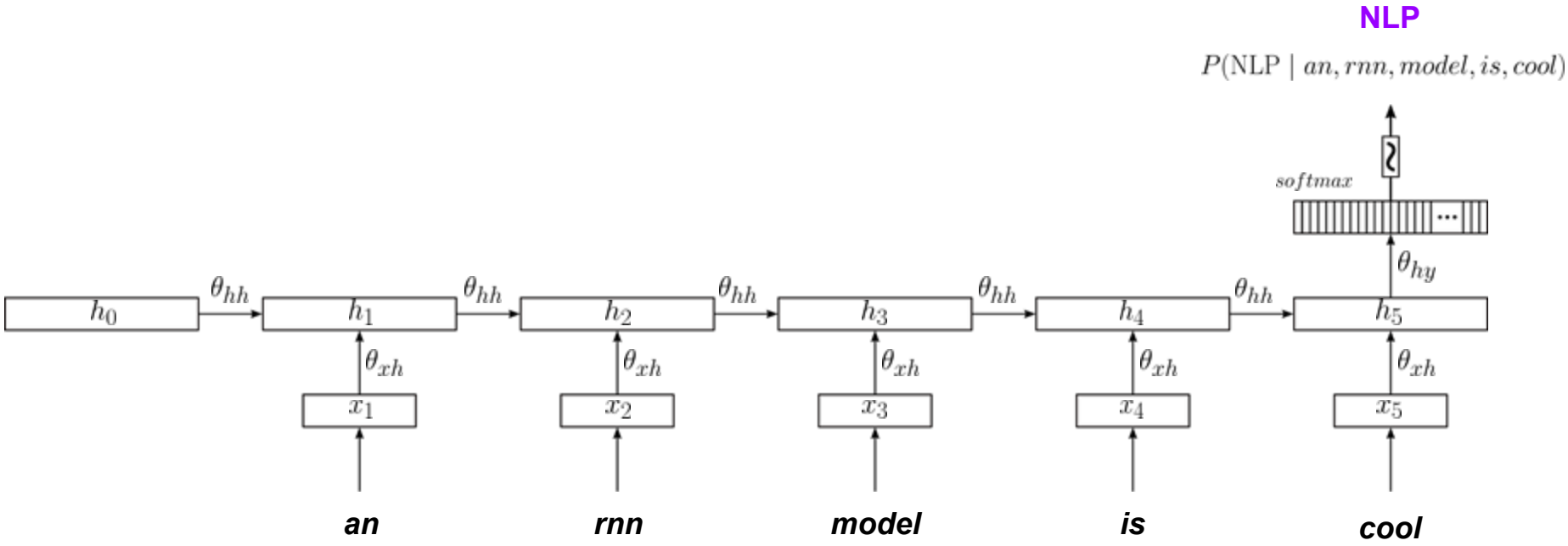
a man sitting on a couch with a dog

a man sitting on a chair with a dog in his lap



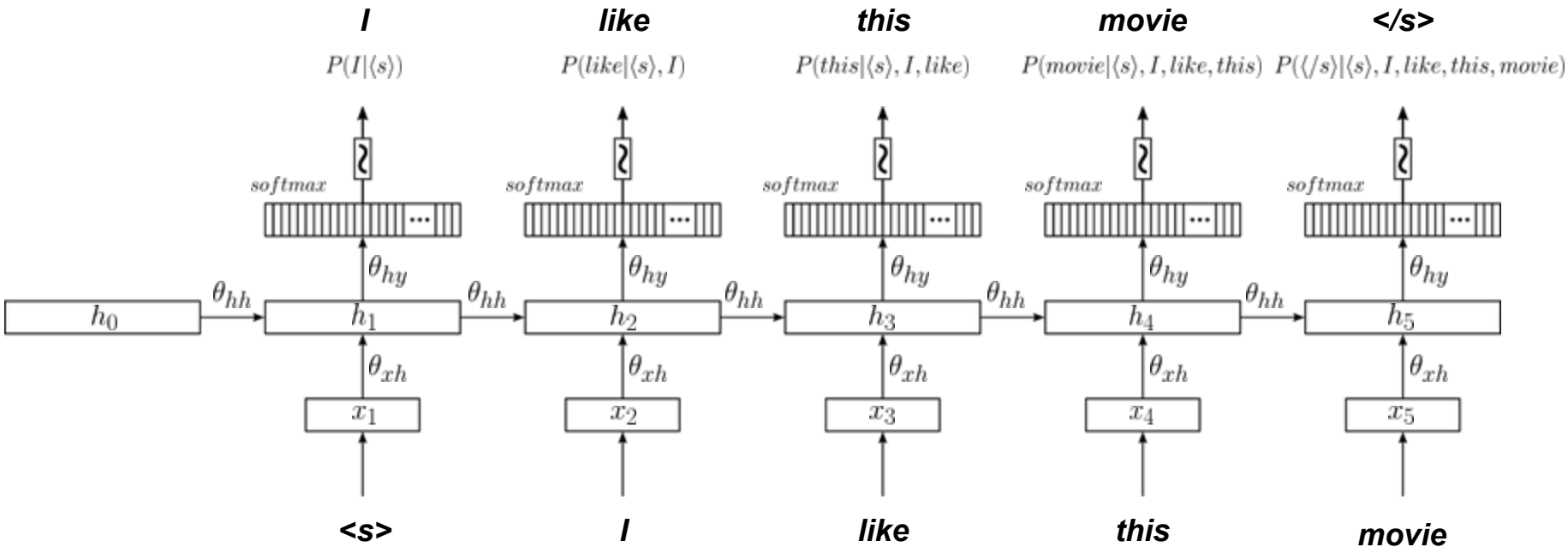
Application: Classification

- Given a text, predict the topic
  - E.g.: Paper title/abstract → AI, NLP, Databases, Biology, …



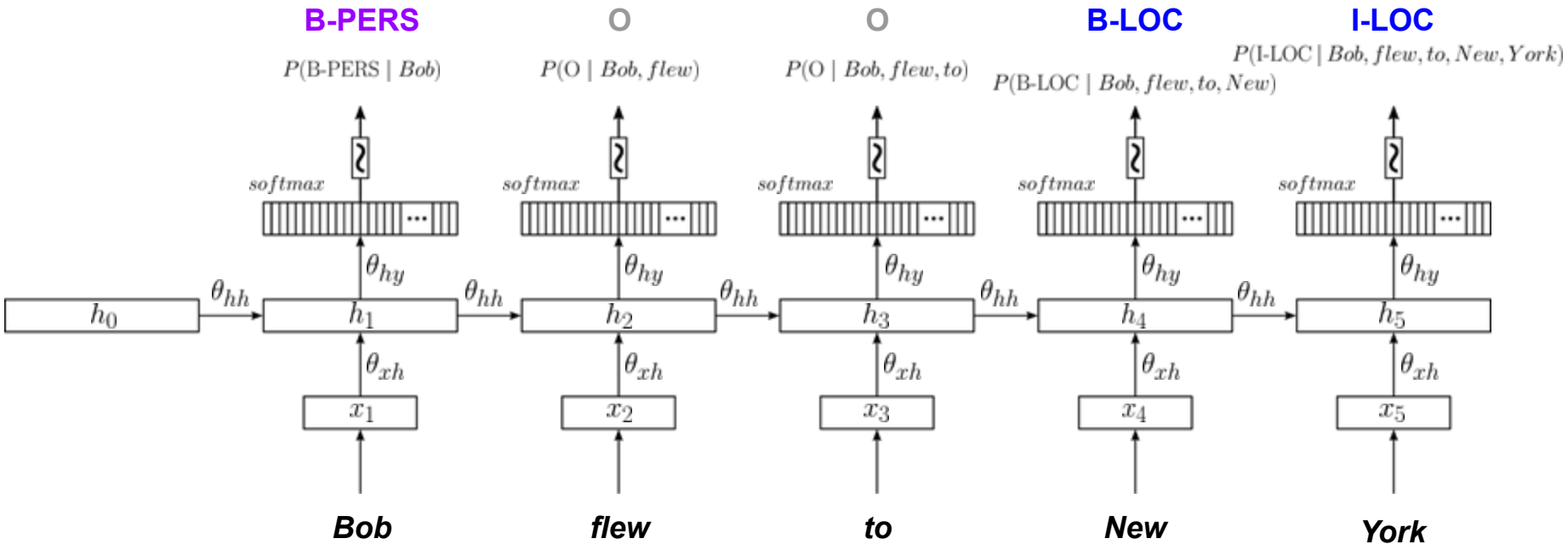
Application: Language Modelling

- Language model: assign probabilities to sequences
  - Here: given a sequence of words, predict the next word (i.e. the word most likely following given sequence)



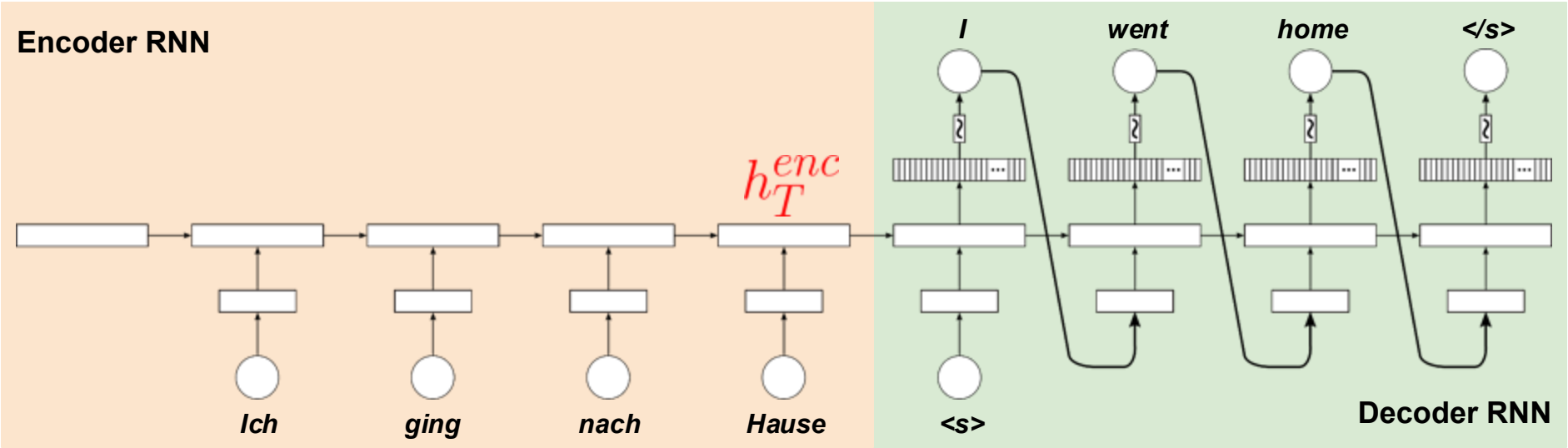
Application: Sequence Labelling

- Example: Named Entity Recognition (NER)
  - Given a sentence, find the most likely NER tags for each word



Application: Machine Translation

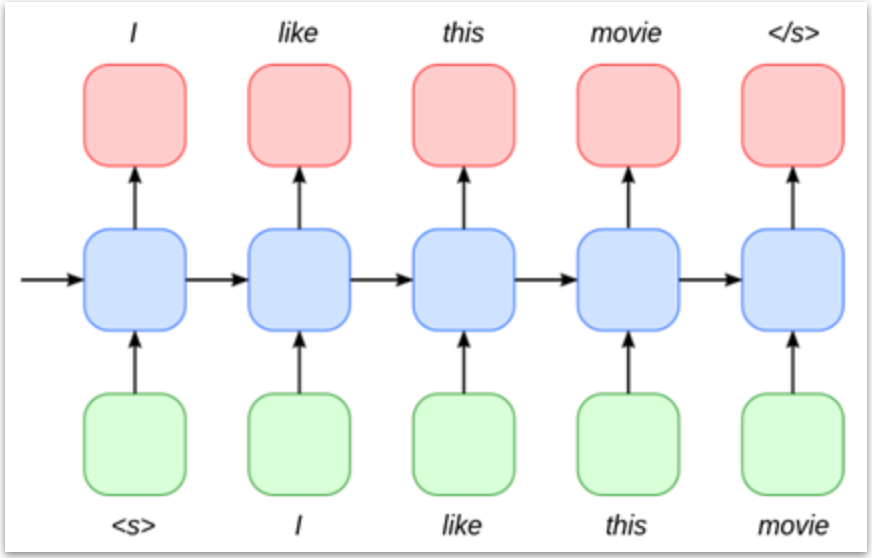
- Encoder-Decoder Architecture (Sutskever et al.; 2014)
  - Encoder: RNN to generate a latent representation (last hidden state h_T^{enc})
  - Decoder: RNN initialized with h_T^{enc} to generate output sequence



Application: Word Embeddings

- ELMo = RNN-based language model, but…
  - LSTM instead of Vanilla RNN
    (better handling of long dependencies)
  - Bi-LSTM — Bidirectional LSTM
    (forward and backward processing of sequence)
  - Two Bi-LSTM layers
    (output of 1st layer = input of 2nd layer)

Recall: Vanilla RNN Language Model



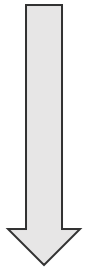
Source: Deep Contextualized Word Representations

RNN — Problem: (Very) Long Sequences

- Training
  - **Vanishing & Exploding Gradients** problem
- Information capture
  - Hidden state h_t must capture all information from $h_0, h_1, ..., h_{t-1}$
  - Information dilutes over time ➜ **bottleneck**

- Performance
  - Processing is intrinsically sequential ➜ **no parallelization**
  - GPU-based performance gain depends on parallelization

➜ **Attention**

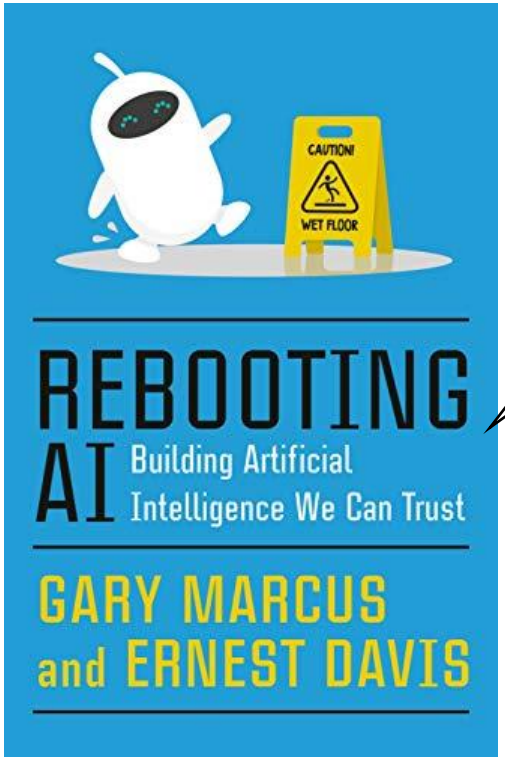


➜ **Transformer**

We'll cover these in the next session on LLMs!

Real world considerations

Deep learning is far from perfect

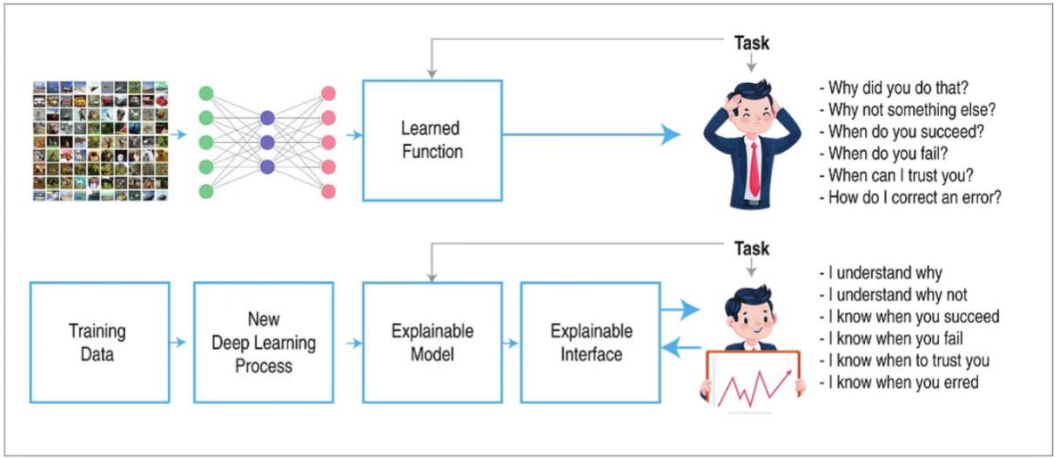


Deep learning is opaque, brittle, and has no commonsense

Deep NNs can be a bit of a black box!

Explainable AI

Developing methods to make AI models more interpretable and transparent



(beyond the scope of the subject, but useful to explore)

Image source: https://medium.com/deepviz/what-is-xai-explainable-ai-and-visualization-part-10-da41c981c5fa

We are Responsible!

Tech does not exist in a vacuum: you can work on problems that will fundamentally make the world a better place or a worse place

(though it’s not always easy to tell)

Misinformation is easily spread through recommendation engines (mostly powered by DL)

World Health Organization | Health Topics | Countries | Newsroom | Emergencies

Global measles threat continues to grow as another year passes with millions of children unvaccinated

16 November 2023 | Joint News Release | Reading time: 2 min (597 words)

Following years of declines in measles vaccination coverage, measles cases in 2022 have increased by 18%, and deaths have increased by 43% globally (compared to 2021). This takes the estimated number of measles cases to 9 million and deaths to 136 000 – mostly among children – according to a new report from the World Health Organization (WHO) and the U.S. Centers for Disease Control and Prevention (CDC).

Measles continues to pose a relentlessly increasing threat to children. In 2022, 37 countries experienced large or disruptive measles outbreaks compared with 22 countries in 2021. Of the countries experiencing outbreaks, 28 were in the WHO Region for Africa, six in the Eastern Mediterranean, two in the South-East Asia, and one in the European Region.

"The increase in measles outbreaks and deaths is staggering, but unfortunately, not unexpected given the declining vaccination rates we've seen in the past few years," said John Vertefeuille, director of CDC's Global Immunization Division. "Measles cases anywhere pose a risk to all countries and communities where people are under-vaccinated. Urgent, targeted efforts are critical to prevent measles disease and

https://www.who.int/news/item/16-11-2023-global-measles-threat-continues-to-grow-as-another-year-passes-with-millions-of-children-unvaccinated

We are Responsible!

As AI becomes more powerful, think about what we should be doing with it to improve society, not just what we can do with it.

It’s important that the next generation of data scientists (you!!!) spend some time thinking about the implications of their work on people and society.

We’ll explore Ethics in NLP in more detail in Session 8 (also assessed in AT3)