

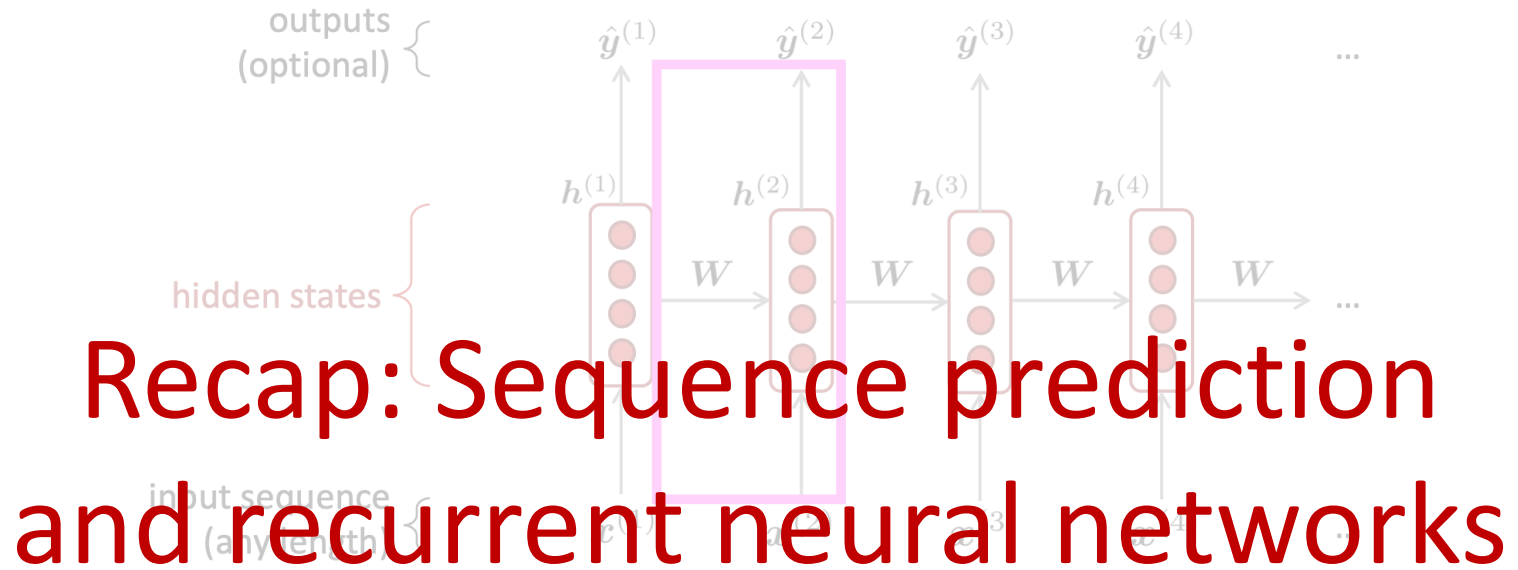
# CSCI 567: Machine Learning

Vatsal Sharan  
Spring 2026

Lecture 10, April 3

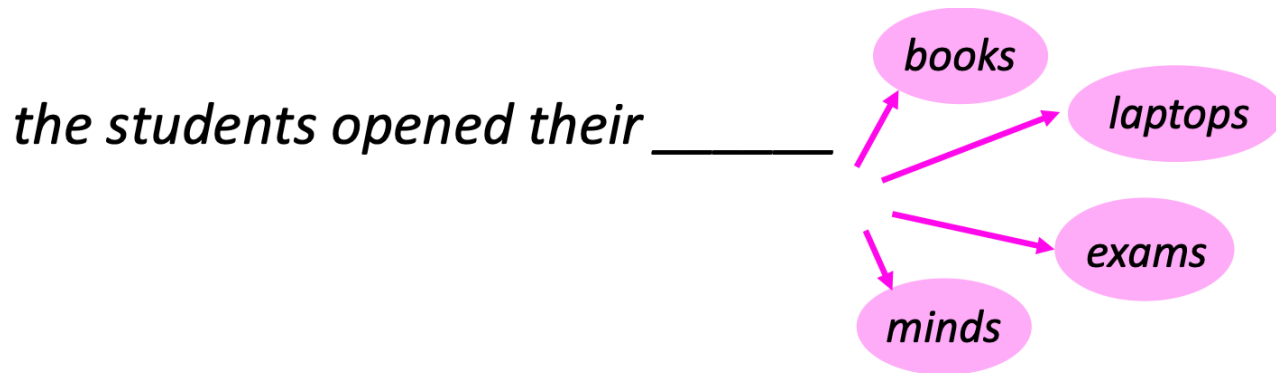
# Administrivia

- HW4 will be released today
- We will have some “mini-discussions” as part of office hours, for problem solving etc., will be announced
- Today’s plan:
  - Attention & Transformers, Decision trees



# Language modelling

**Language modelling** is the task of predicting what word comes next:



More formally, let  $X_i$  be the random variable for the  $i$ -th word in the sentence, and let  $x_i$  be the **value** taken by the random variable. Then the goal is to compute

$$P(X_{t+1} | X_t = x_t, \dots, X_1 = x_1).$$

A system that does this is known as a **Language Model**.

# $n$ -gram language model: A type of Markov model

A **Markov model or Markov chain** is a sequence of random variables with the **Markov property**: a sequence of random variables  $X_1, X_2, \dots$  s.t.

$$P(X_{t+1} | X_{1:t}) = P(X_{t+1} | X_t) \quad (\text{Markov property})$$

i.e. *the next state only depends on the most recent state* (notation  $X_{1:t}$  denotes the sequence  $X_1, \dots, X_t$ ). This is a *bigram model*.

We will consider the following setting:

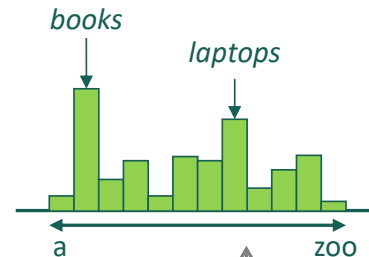
- All  $X_t$ 's take value from the same **discrete** set  $\{1, \dots, S\}$
- $P(X_{t+1} = s' | X_t = s) = a_{s,s'}$ , known as **transition probability**
- $P(X_1 = s) = \pi_s$
- $(\{\pi_s\}, \{a_{s,s'}\}) = (\boldsymbol{\pi}, \mathbf{A})$  are **parameters of the model**

# A Simple RNN Language Model

output distribution

$$\hat{y}^{(t)} = \text{softmax}(U\mathbf{h}^{(t)} + \mathbf{b}_2) \in \mathbb{R}^{|V|}$$

$$\hat{y}^{(4)} = P(\mathbf{x}^{(5)} | \text{the students opened their})$$



hidden states

$$\mathbf{h}^{(t)} = \sigma(\mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_e \mathbf{e}^{(t)} + \mathbf{b}_1)$$

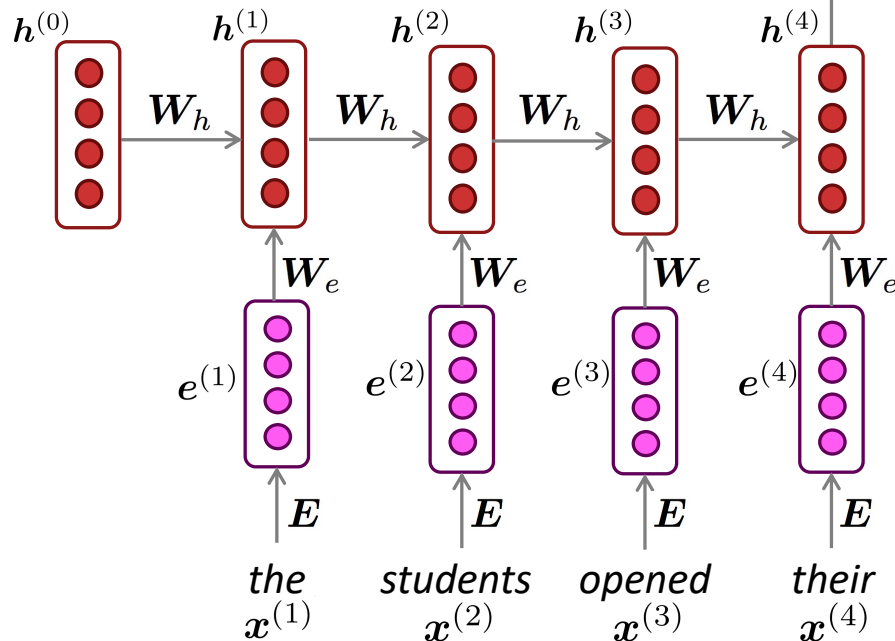
$\mathbf{h}^{(0)}$  is the initial hidden state

word embeddings

$$\mathbf{e}^{(t)} = \mathbf{E}\mathbf{x}^{(t)}$$

words / one-hot vectors

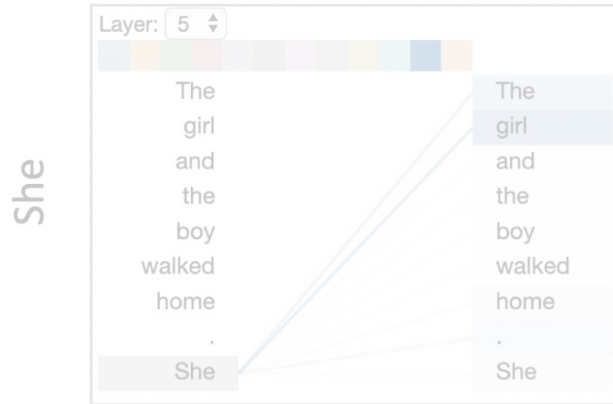
$$\mathbf{x}^{(t)} \in \mathbb{R}^{|V|}$$



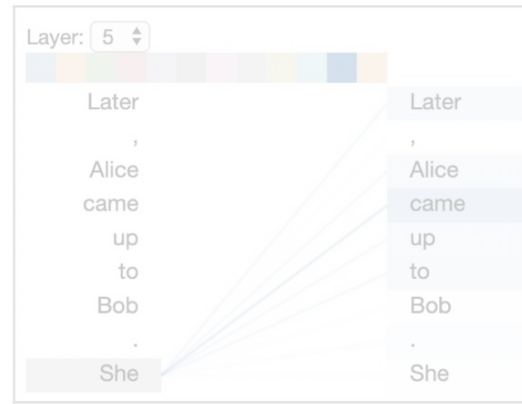
*Note: this input sequence could be much longer now!*

Slide adapted from CS224n by Chris Manning (Lecture 5)

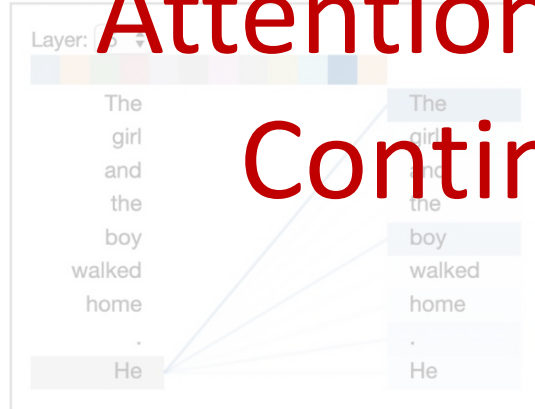
### Gender-specific term



### Name

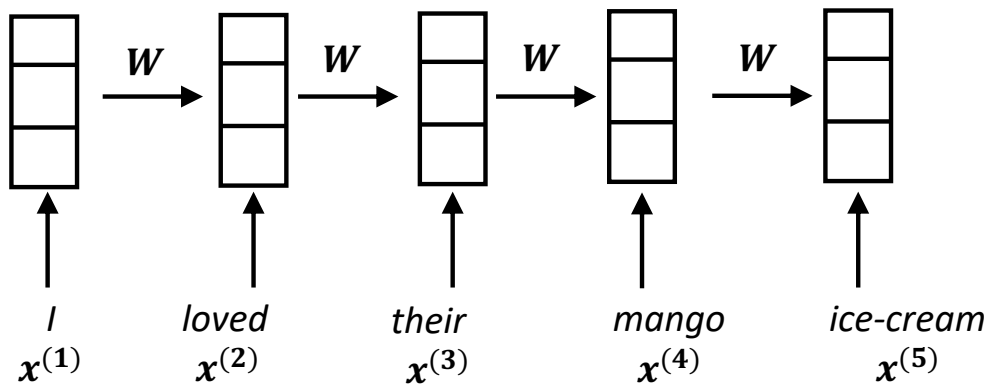


He



# Attention: Recap & Continuation

# The problem with recurrence



1. Must always compress all necessary information into one hidden state representation

2. Cannot capture long-range dependencies in input (“vanishing gradients problem”)

Inputs from sufficiently far away do not contribute to hidden state representation:

Suppose  $W = \begin{pmatrix} 0.8 & 0.2 \\ -0.6 & 0.9 \end{pmatrix}$

Then  $W^5 = \begin{pmatrix} -0.31 & 0.35 \\ -1.06 & -0.13 \end{pmatrix}$ ,  $W^{10} = \begin{pmatrix} -0.28 & -0.16 \\ 0.47 & -0.36 \end{pmatrix}$ ,  $W^{50} = \begin{pmatrix} 0.01 & 0.00 \\ -0.01 & 0.01 \end{pmatrix}$

# A solution: Attention

---

## Attention Is All You Need

---

**Ashish Vaswani\***  
Google Brain  
avaswani@google.com

**Noam Shazeer\***  
Google Brain  
noam@google.com

**Niki Parmar\***  
Google Research  
nikip@google.com

**Jakob Uszkoreit\***  
Google Research  
usz@google.com

**Llion Jones\***  
Google Research  
llion@google.com

**Aidan N. Gomez\* †**  
University of Toronto  
aidan@cs.toronto.edu

**Lukasz Kaiser\***  
Google Brain  
lukaszkaizer@google.com

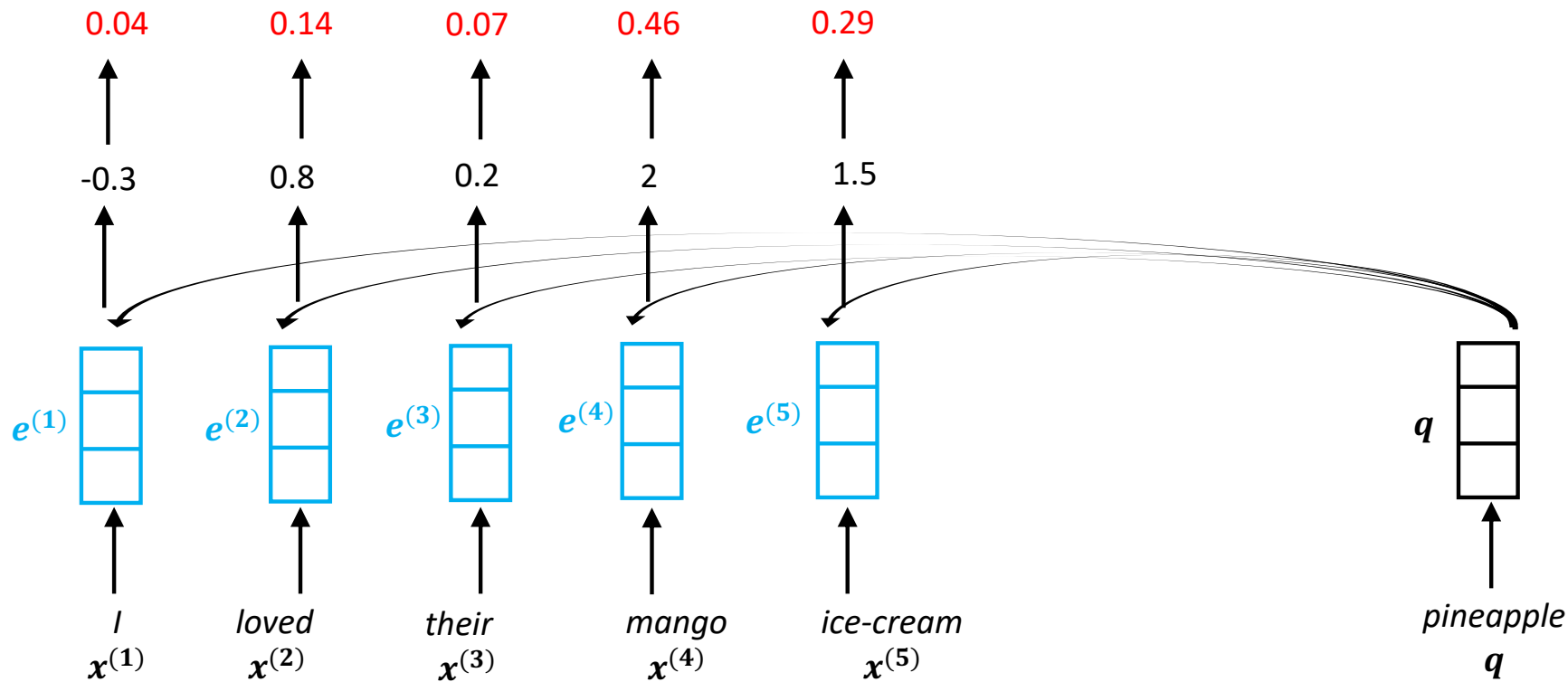
**Illia Polosukhin\* ‡**  
illia.polosukhin@gmail.com

### Abstract

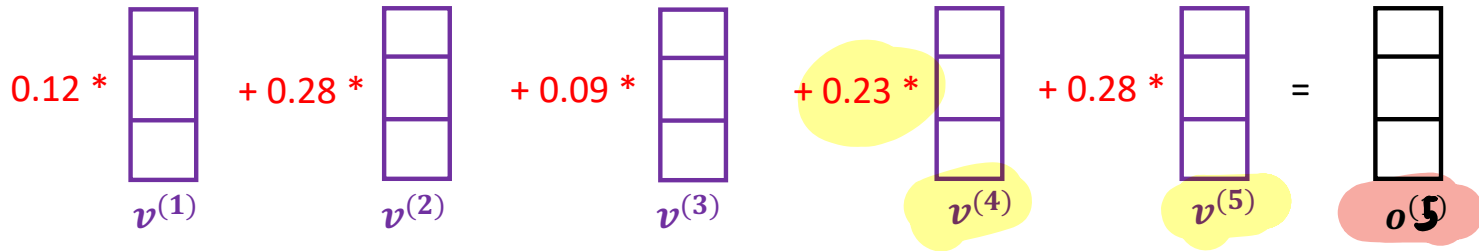
The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.0 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

# Attention: Weighted averaging

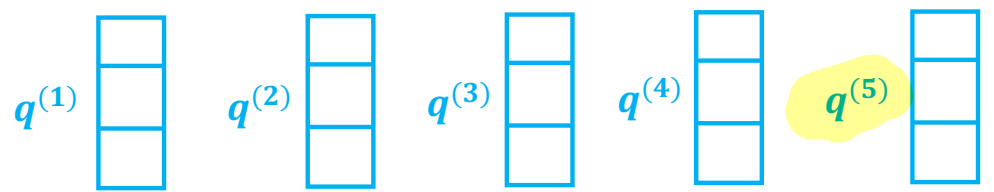
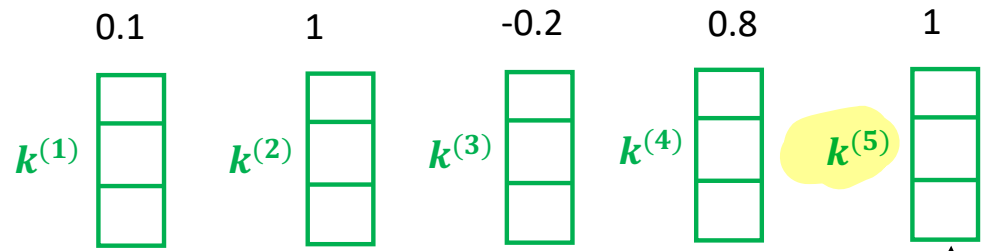
$$0.04 * e^{(1)} + 0.14 * e^{(2)} + 0.07 * e^{(3)} + 0.46 * e^{(4)} + 0.29 * e^{(5)} = \mathbf{o}$$



# Self-attention



0.12      0.28      0.09      0.23      0.28



*I*  $x^{(1)}$       *loved*  $x^{(2)}$       *their*  $x^{(3)}$       *mango*  $x^{(4)}$       *ice-cream*  $x^{(5)}$

# Self-attention in matrix form

1. Transform each word embedding with weight matrices  $\mathbf{Q}$ ,  $\mathbf{K}$ ,  $\mathbf{V}$ , each in  $\mathbb{R}^{d \times d}$

$$\mathbf{q}_i = \mathbf{Q} \mathbf{x}_i \quad (\text{queries})$$

$$\mathbf{k}_i = \mathbf{K} \mathbf{x}_i \quad (\text{keys})$$

$$\mathbf{v}_i = \mathbf{V} \mathbf{x}_i \quad (\text{values})$$

2. Compute pairwise similarities between keys and queries; normalize with softmax

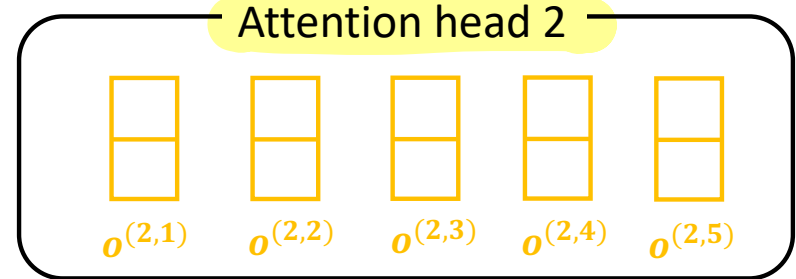
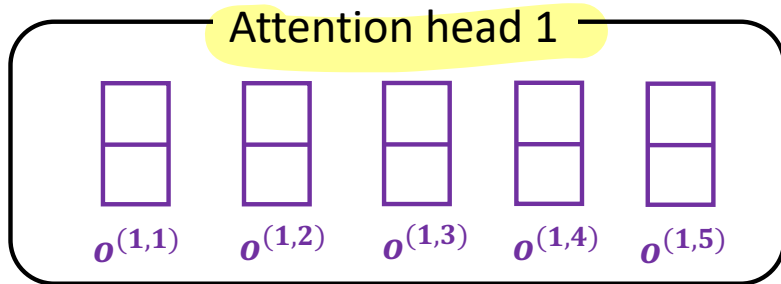
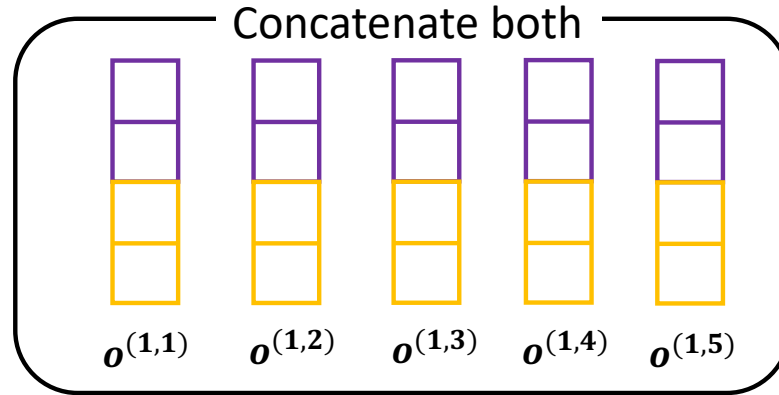
$$\alpha_{ij} = \mathbf{q}_i^\top \mathbf{k}_j$$

$$w_{ij} = \frac{\exp(\alpha_{ij})}{\sum_{j'} \exp(\alpha_{ij'})}$$

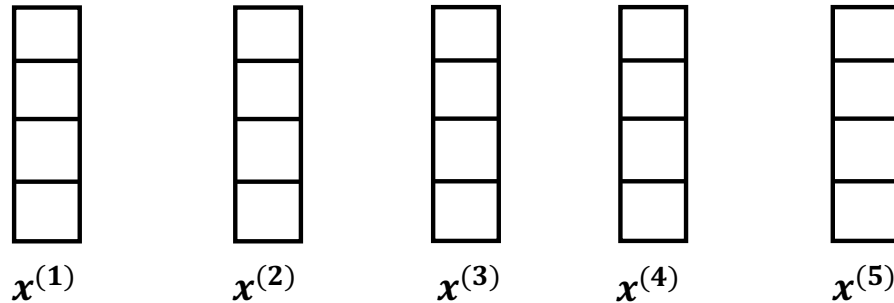
3. Compute output for each word as weighted sum of values

$$\mathbf{o}_i = \sum_j w_{ij} \mathbf{v}_j$$

# Multi headed self-attention



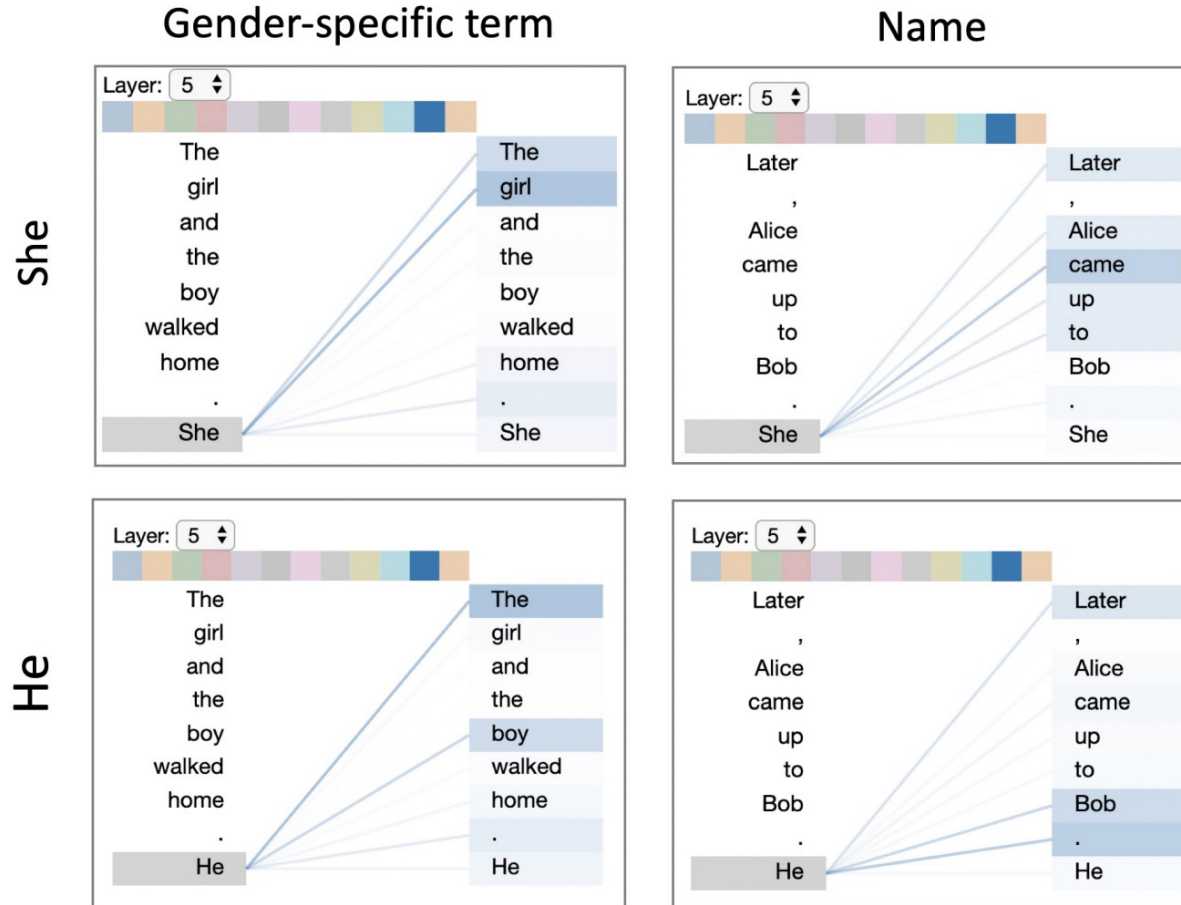
Apply attention  
layer with  
 $Q, K, V \in \mathbb{R}^{(d/2)+d}$



# Multi headed self-attention

- Input: List of vectors  $\mathbf{x}_1, \dots, \mathbf{x}_T$ , each of size  $d$
- Output: List of vectors  $\mathbf{h}_1, \dots, \mathbf{h}_T$ , each of size  $d$
- Formula: For each head  $i$ :
  - Compute self attention output using  $\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i$
  - Finally, concatenate results for all heads
- Parameters:
  - For each head  $i$ , parameter matrices  $\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i$  of size  $d_{\text{attn}} \times d$
  - # of heads  $n$  is hyperparameter,  $d_{\text{attn}} = d/n$

# What do attention heads learn?

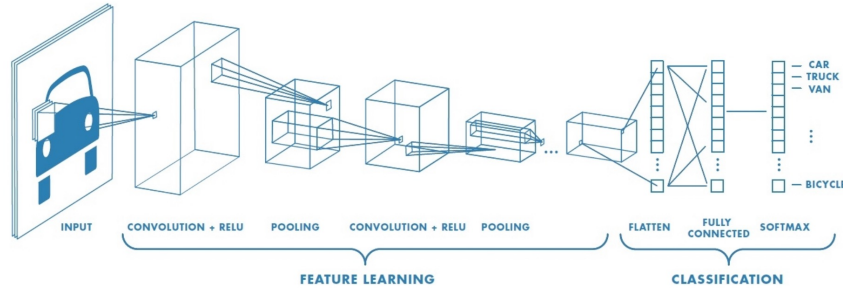




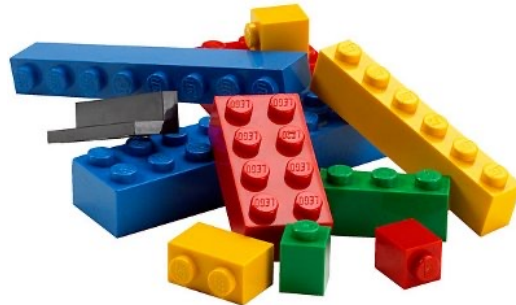
Transformers

# Building a Transformer, stack things up!

To build a CNN, we stacked up convolution layers



You should think of different components (convolution, non-linearities, pooling, attention, softmax etc.) as building block which can be composed to build a final model.

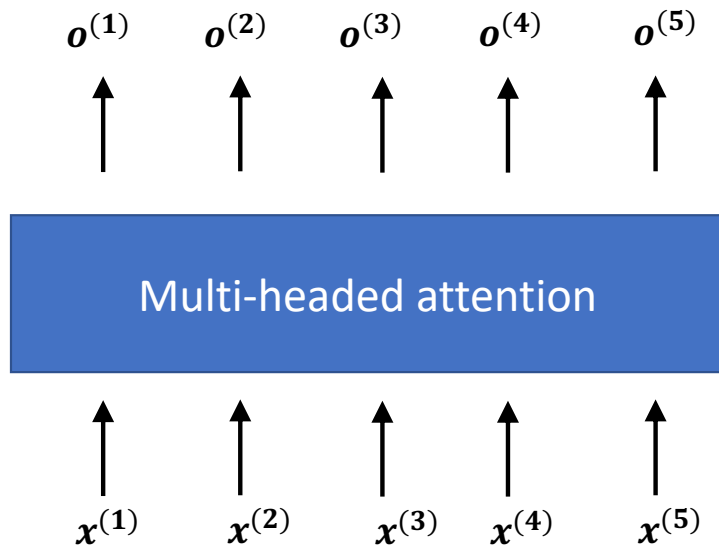


# Building a Transformer, stack things up!

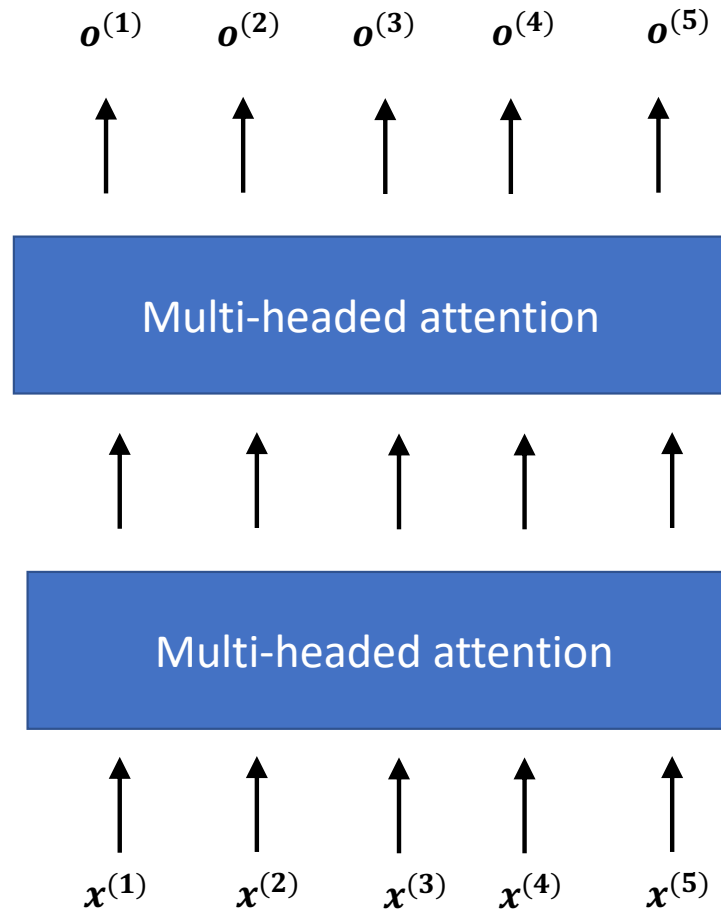
How should we stack different layers to build a Transformer?



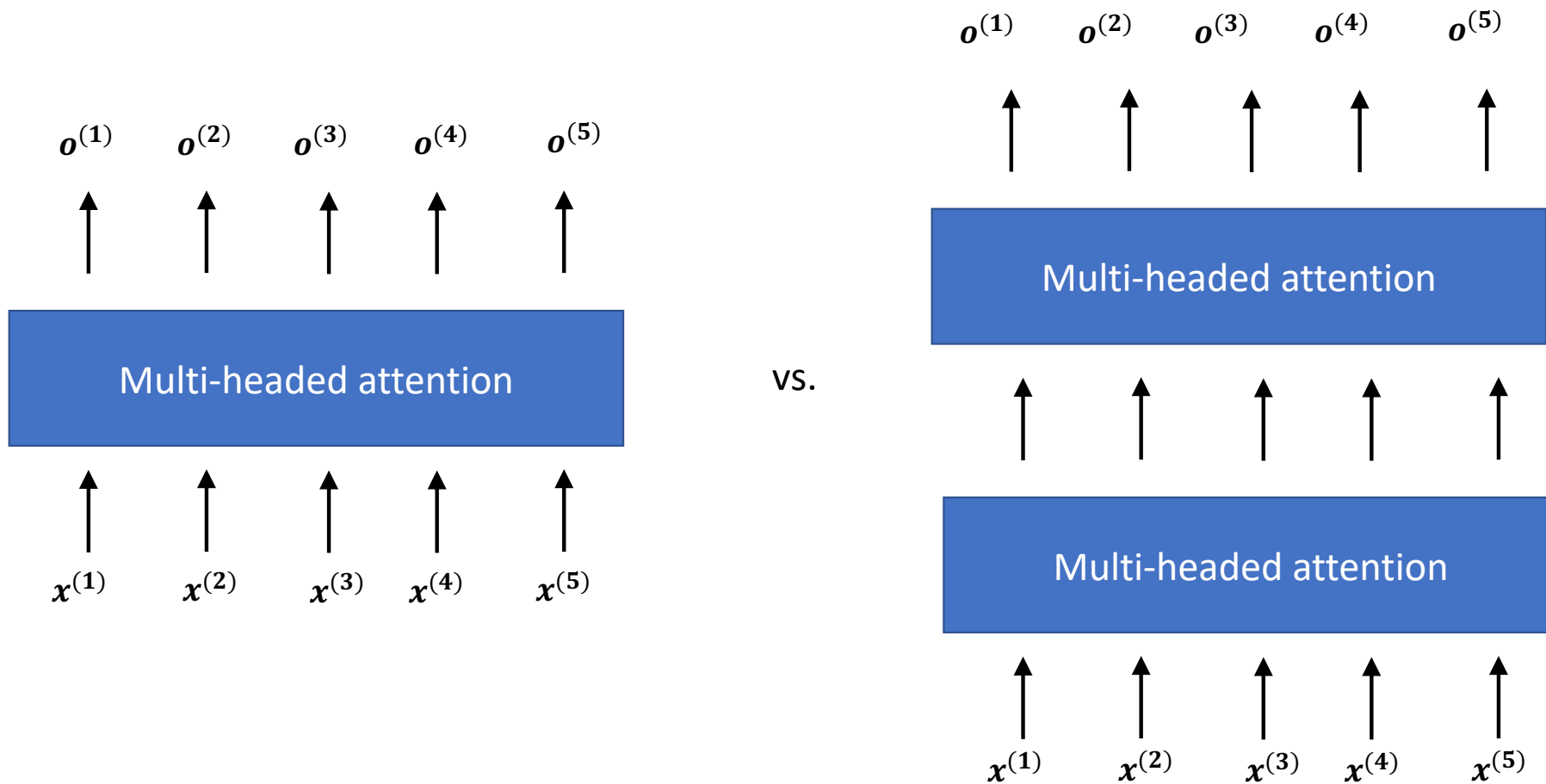
# Key ingredient: Attention layers



# What if we just stack attention layers?



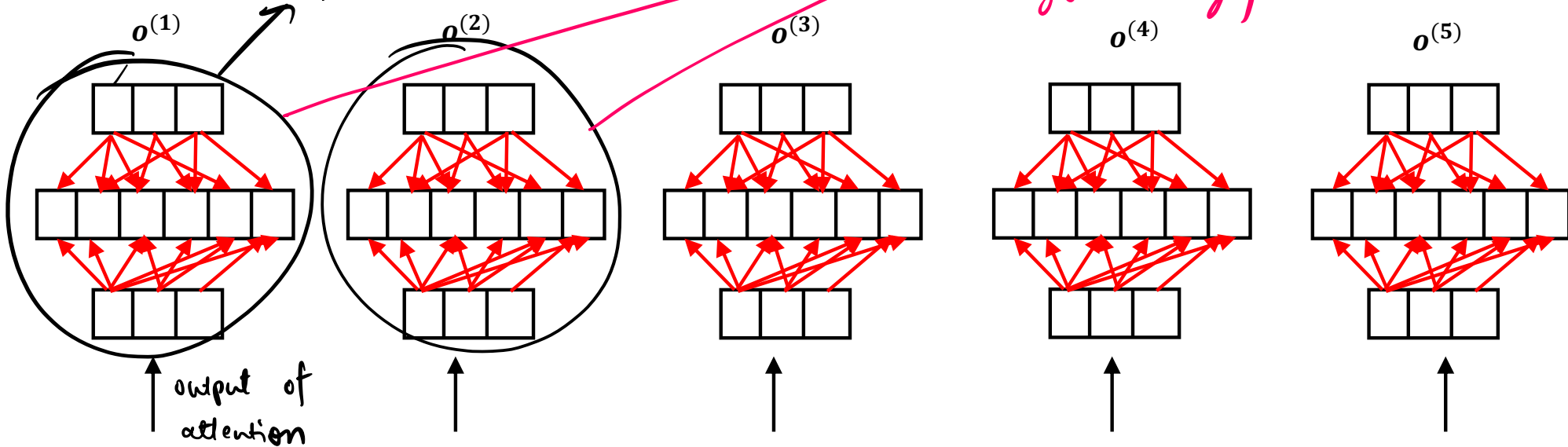
# Issue: Multiple attention layers are still just averaging



# Solution: add non-linearities 😊

fully connected neural network with one hidden layer

same weights (weight-sharing)



Multi-headed attention

$x^{(1)}$

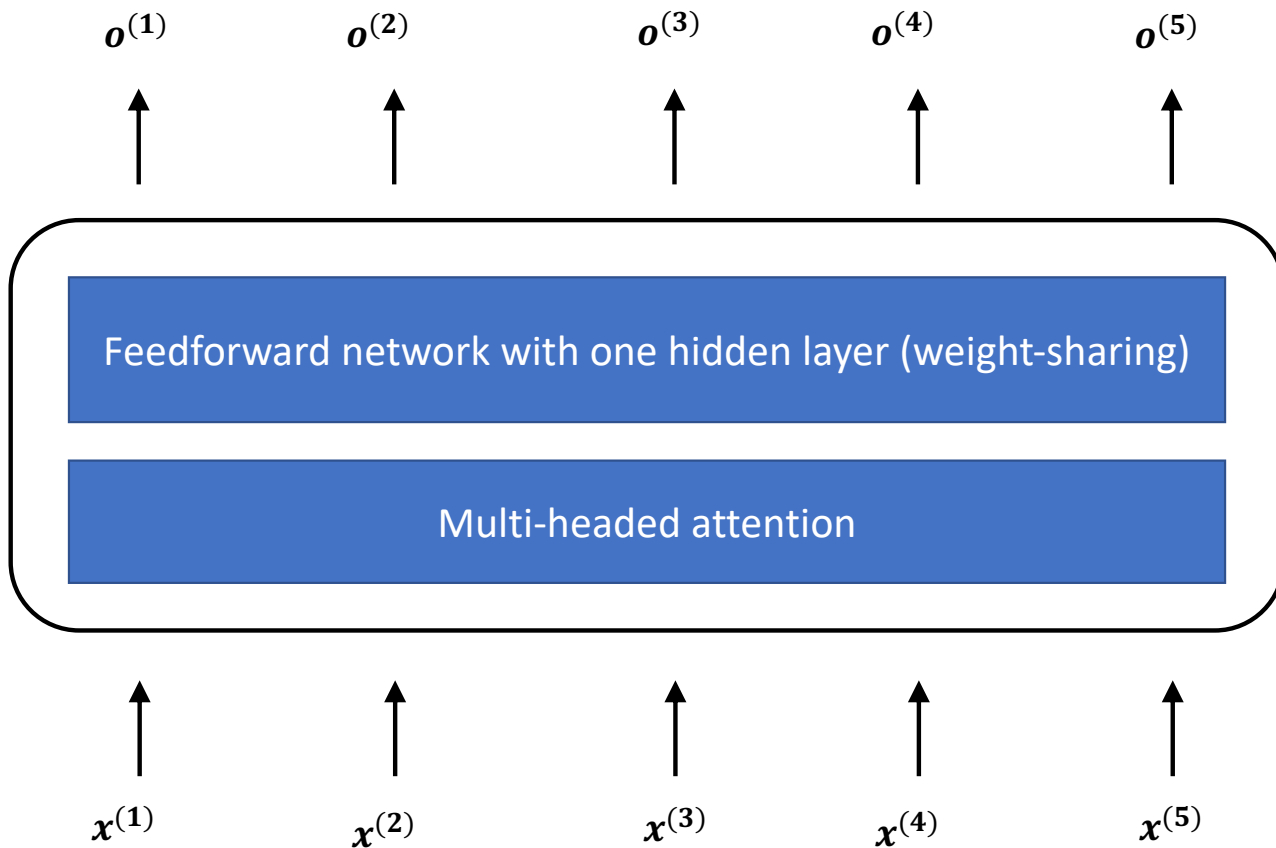
$x^{(2)}$

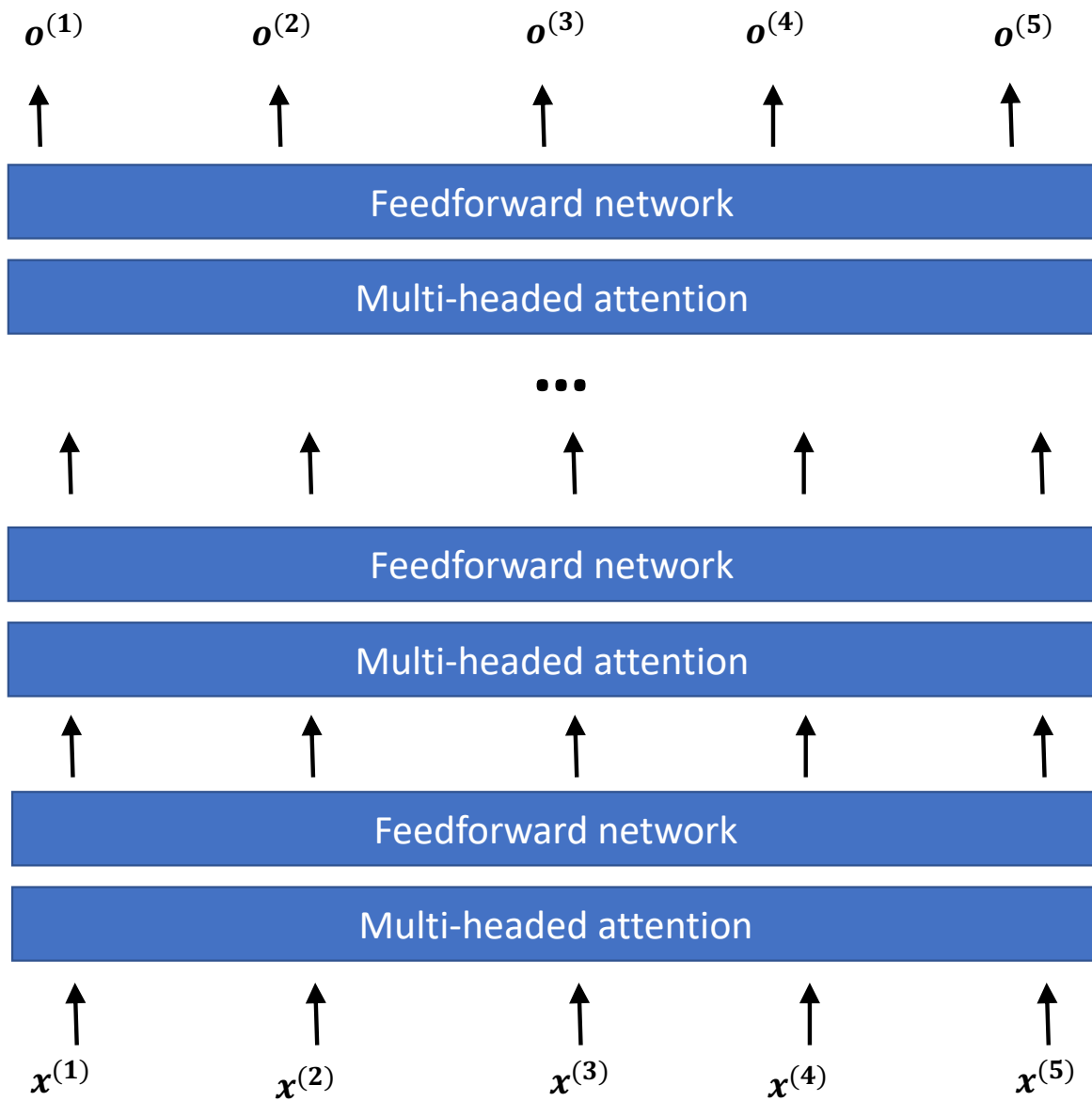
$x^{(3)}$

$x^{(4)}$

$x^{(5)}$

## Solution: add non-linearities 😊





# Building a Transformer: Bells and whistles

- Positional embeddings
- Residual connections
- Layer normalization
- Scaled dot product attention
- Tokenization



# Positional embeddings

Issue with current architecture:  
Embeddings do not take order into account

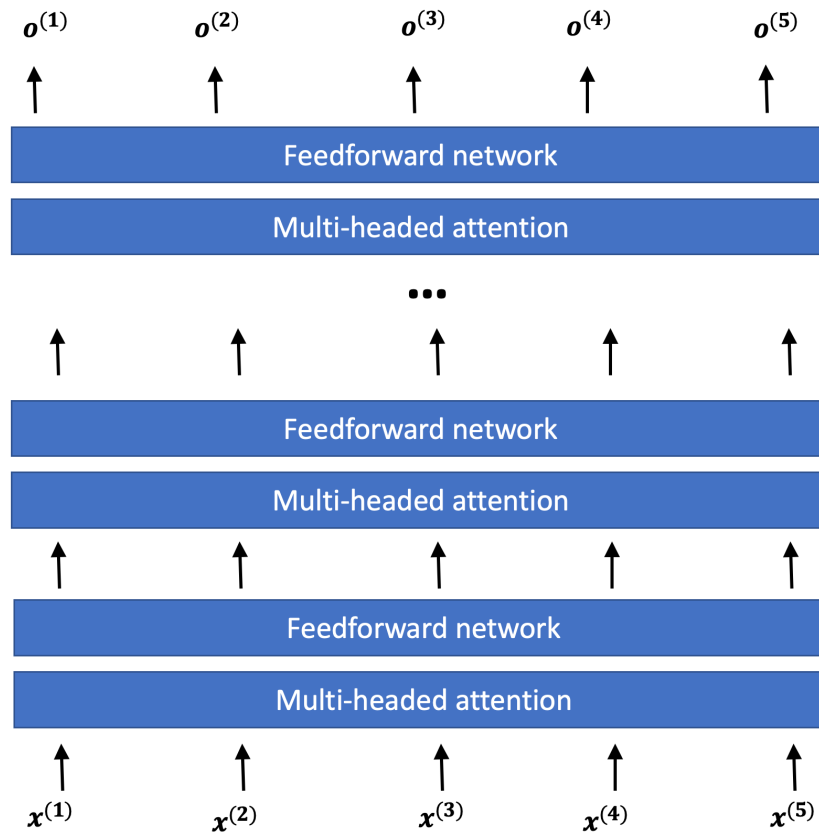
``bob ate the salmon``

``the salmon ate bob``

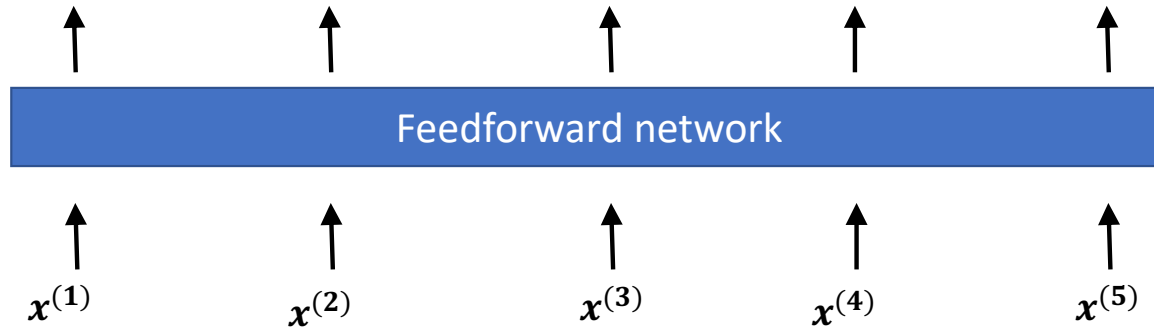
Lead to the exact same embedding for each word!

Fix: **positional embeddings**

- Learn different vector for each position
- Add this vector to word vector for that position



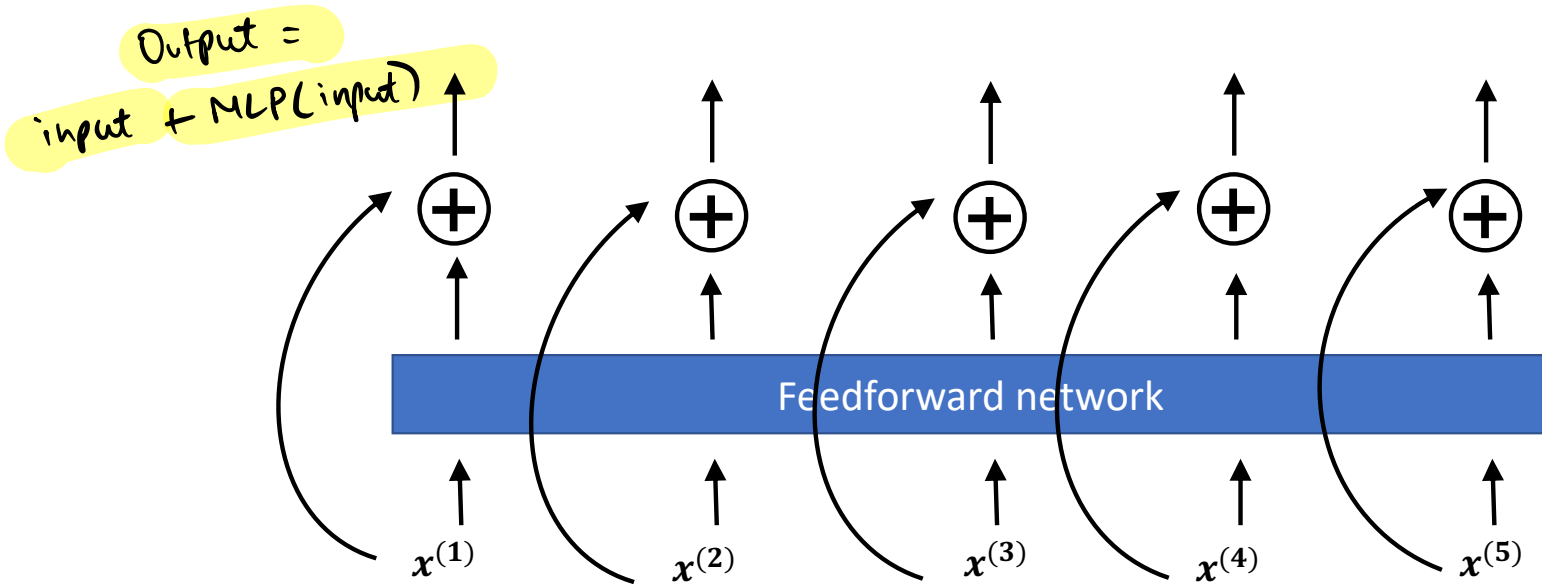
# Residual connections



Each feedforward layer

- Takes as input  $T \times d$  matrix  $X$  ( $T$  is number of tokens,  $d$  is dimensionality of embedding)
- Outputs  $T \times d$  matrix  $O$  ( $T$  is number of tokens,  $d$  is dimensionality of embedding)

# Residual connections



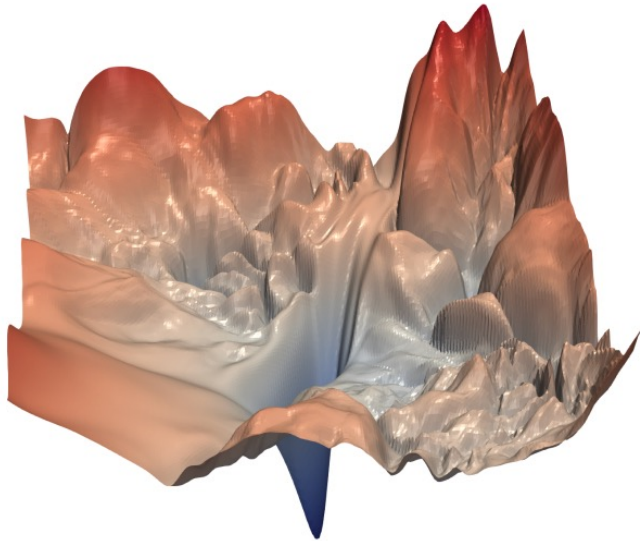
We add a **residual connection**: Output is now  $X + O$

- Add together the input to the layer to its previous output
- The feedforward layer now only has to compute what  $O$  should be added to the input to have a useful embedding

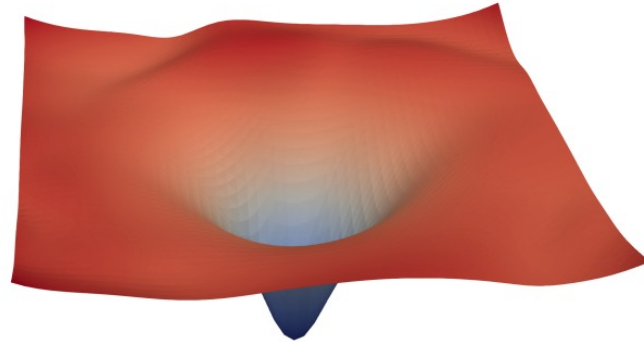
# Residual connections: Why do they help?

With residual connections:

- It makes it easier to copy information from before
- Helps in training deeper networks (avoids gradients from initial layers from becoming too small)
- Seems to make the optimization landscape better behaved



Without residual connections



With residual connections

# Layer normalization

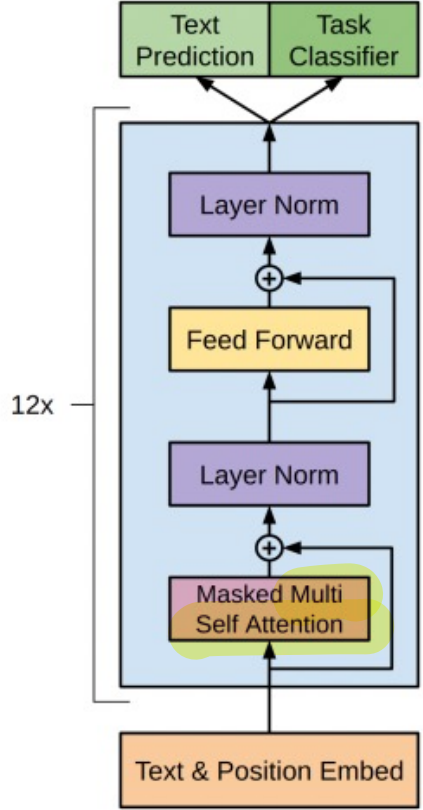
Given some input  $x$

- Normalize  $x$  to have mean  $\mu = 0$  and variance  $\sigma^2 = 1$

We add this **layer norm** operation in the Transformer blocks to make sure the values are not too large or too small.

Variations of this are also often used to train deep models (including non-Transformer architectures)

# Putting things together: The Transformer architecture



*"I loved their mango ice cream"*

Architecture of the GPT model  
(figure from GPT paper)

# Lots of variants, and extensions to other kinds of data...

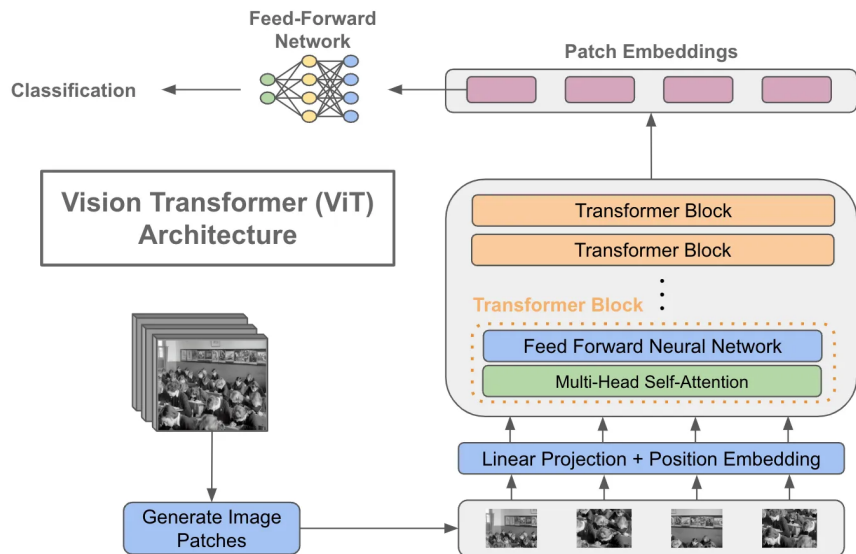
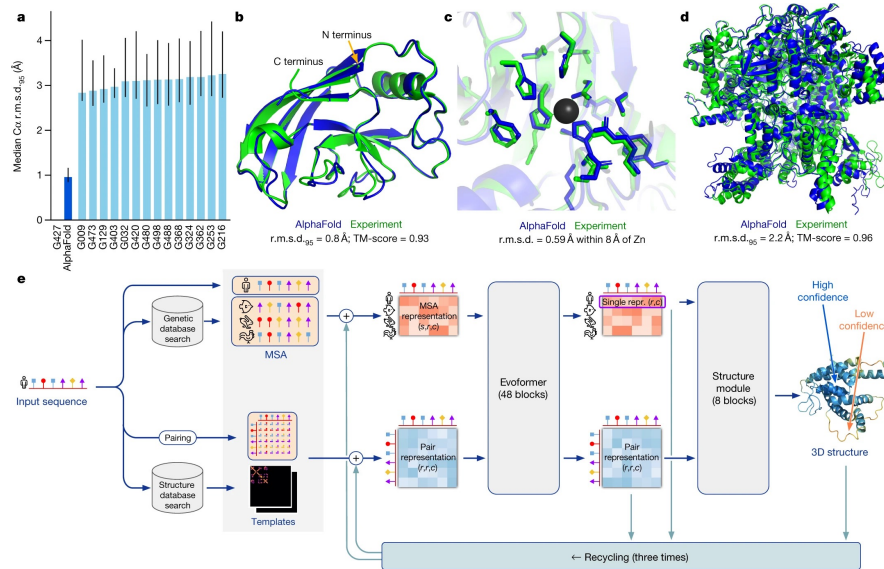
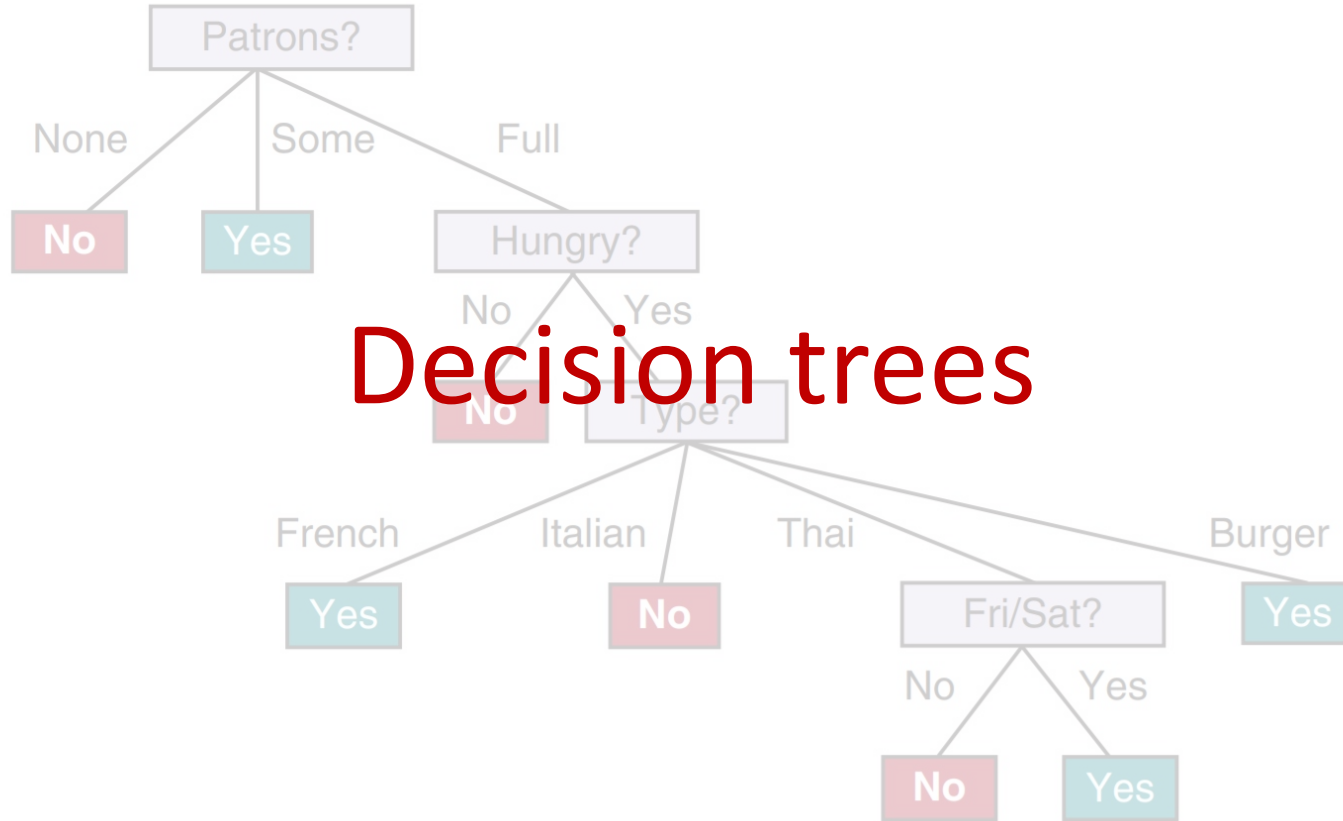


Image data



Protein structure prediction



# Decision trees

# Decision trees

- Introduction & definition
- Learning the parameters
- Measures of uncertainty
- Recursively learning the tree & some variants

# Decision trees

We have seen different ML models for classification/regression:

- linear models, nonlinear models induced by kernels, neural networks

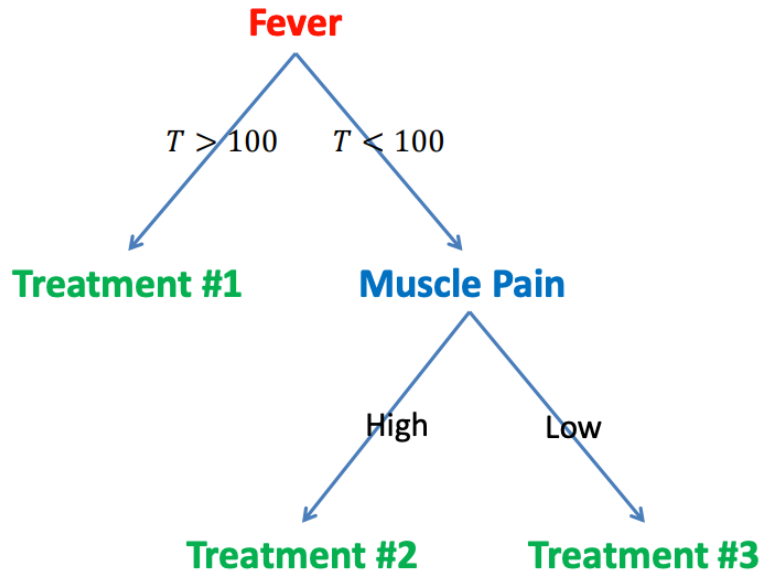
**Decision tree** is another popular one:

- **nonlinear** in general
- works for both classification and regression; we focus on **classification**
- one key advantage is good **interpretability**
- **ensembles** of trees are very effective

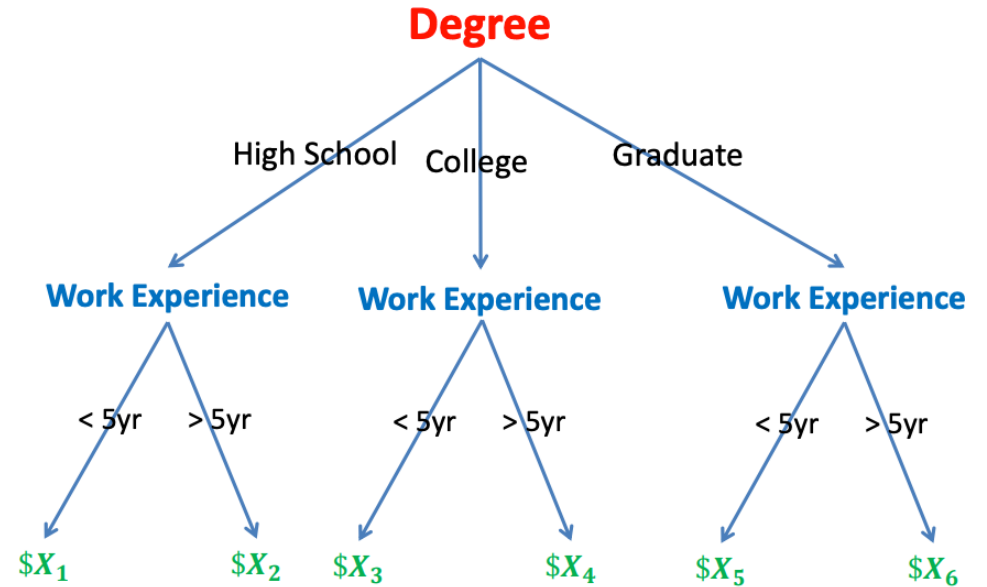
# Example

Many decisions are made based on some tree structure

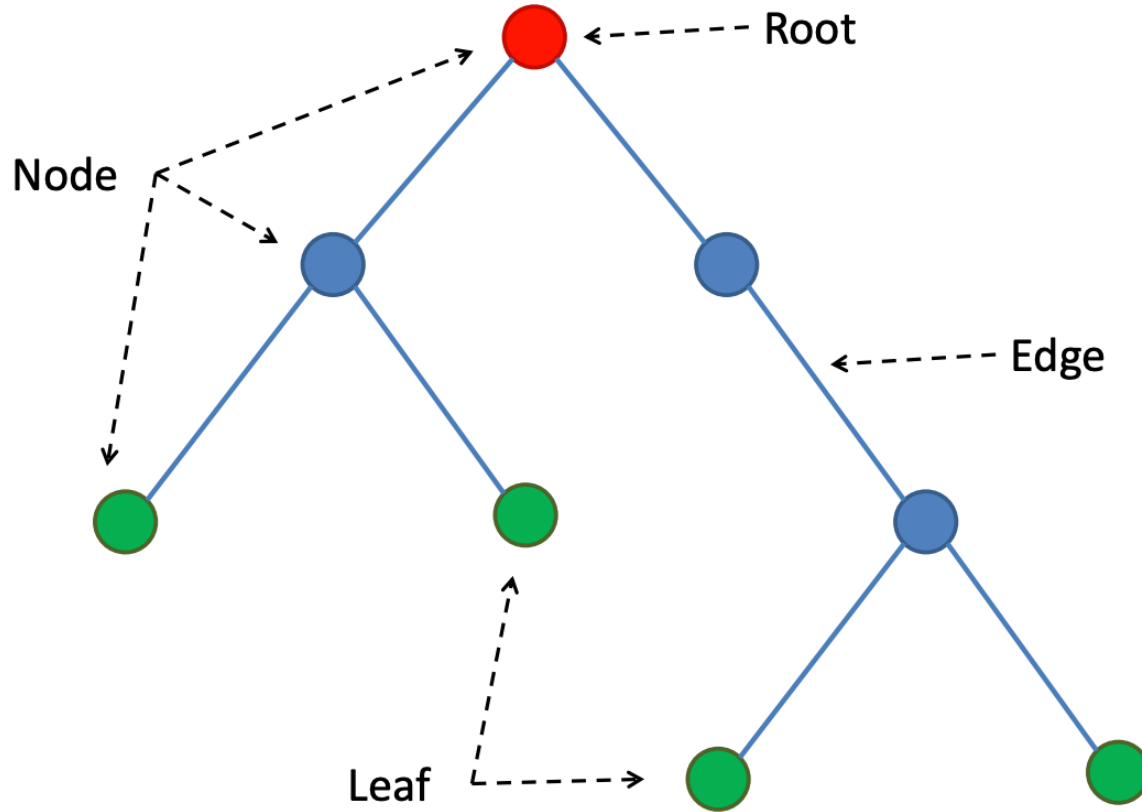
Medical treatment



Salary in a company



# Tree terminology

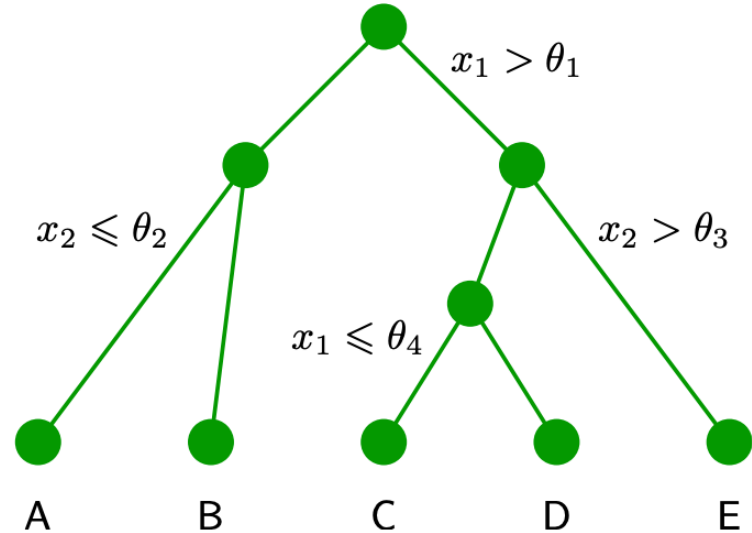


# Tree terminology

**Input:**  $\mathbf{x} = (x_1, x_2)$

**Output:**  $f(\mathbf{x})$  determined naturally by **traversing** the tree

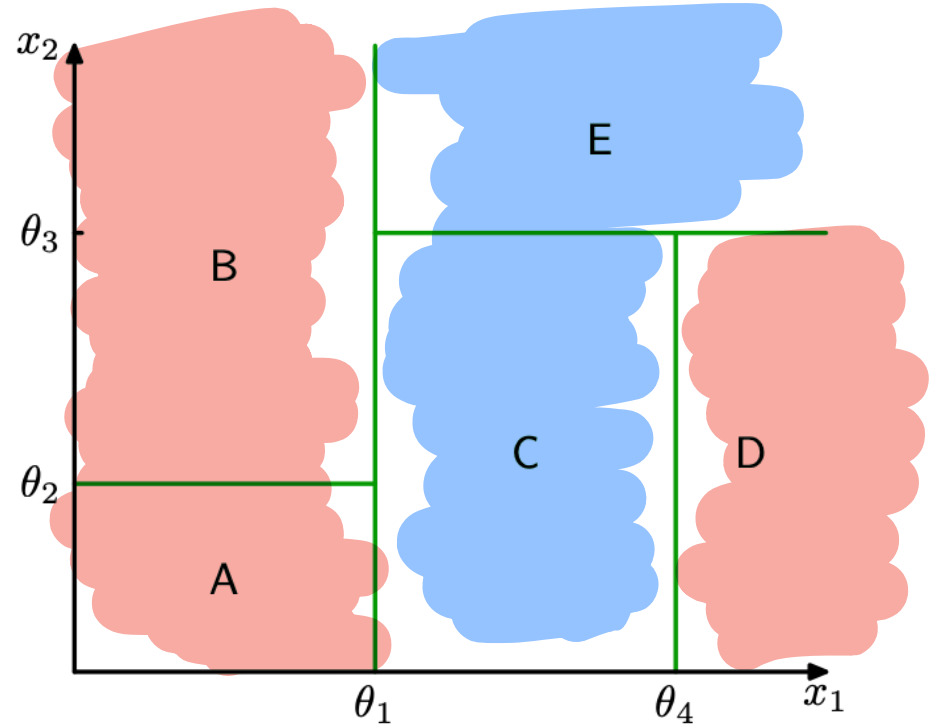
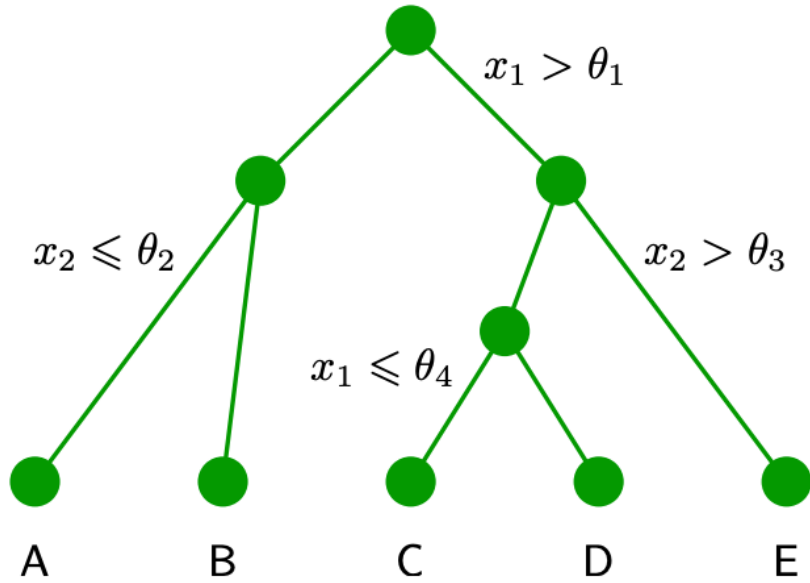
- start from the root
- test at each node to decide which child to visit next
- finally the leaf gives the prediction  $f(\mathbf{x})$



For example,  $f((\theta_1 - 1, \theta_2 + 1)) = B$

Complex to formally write down, but **easy to represent pictorially or as code.**

# Decision boundary



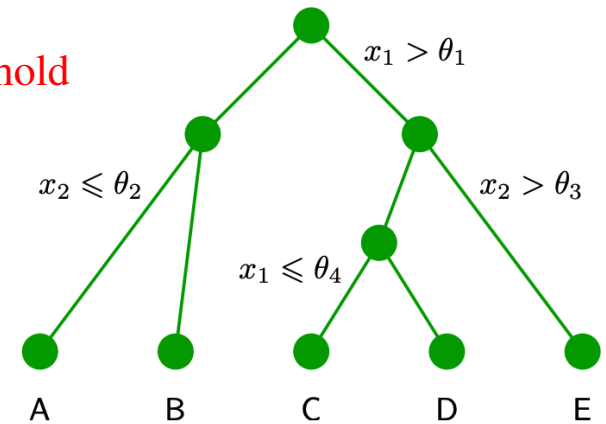
if A, B, D ; +ve label (red)  
C, E : -ve label (blue)

Nonlinear decision boundary!

# Parameters

**Parameters** to learn for a decision tree:

- The **structure** of the tree, such as the depth, #branches, #nodes, etc.  
Some of these are considered as hyperparameters. The structure of a tree is *not fixed in advance, but learned from data*.
- The **test** at each internal node:  
Which **feature(s)** to test on? If the feature is continuous, what **threshold** ( $\theta_1, \theta_2, \dots$ )?
- The **value/prediction** of the leaves (A, B, ...)



# Decision trees

- Introduction & definition
- **Learning the parameters**
- Measures of uncertainty
- Recursively learning the tree & some variants

# Learning the parameters (optimization?)

So how do we *learn all these parameters?*

Empirical risk minimization (ERM): find the parameters that **minimize some loss**.

However, doing exact ERM *is too expensive for trees*.

- for  $T$  nodes, there are roughly  $(\text{\#features})^T$  possible decision trees (need to decide which feature to use on each node).
- enumerating all these configurations to find the one that minimizes some loss is too computationally expensive.
- since most of the parameters are discrete ( $\text{\#branches}$ ,  $\text{\#nodes}$ , feature at each node, etc.) *cannot really use gradient based approaches*.

*e.g. can't really use gradient descent to decide which feature  $(x_1, \dots, x_d)$  to split on at any level.*

Instead, we turn to some **greedy top-down approach**.

# A running example

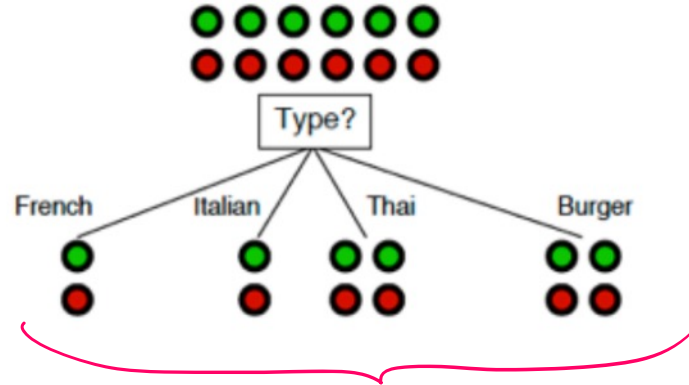
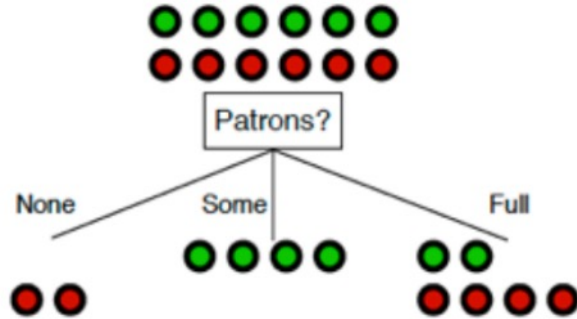
- predict whether a customer will wait to get a table at some restaurant
- 12 training examples
- 10 features (all discrete)

Example	Attributes										Target WillWait
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	
$X_1$	T	F	F	T	Some	\$\$\$	F	T	French	0-10	T
$X_2$	T	F	F	T	Full	\$	F	F	Thai	30-60	F
$X_3$	F	T	F	F	Some	\$	F	F	Burger	0-10	T
$X_4$	T	F	T	T	Full	\$	F	F	Thai	10-30	T
$X_5$	T	F	T	F	Full	\$\$\$	F	T	French	>60	F
$X_6$	F	T	F	T	Some	\$\$	T	T	Italian	0-10	T
$X_7$	F	T	F	F	None	\$	T	F	Burger	0-10	F
$X_8$	F	F	F	T	Some	\$\$	T	T	Thai	0-10	T
$X_9$	F	T	T	F	Full	\$	T	F	Burger	>60	F
$X_{10}$	T	T	T	T	Full	\$\$\$	F	T	Italian	10-30	F
$X_{11}$	F	F	F	F	None	\$	F	F	Thai	0-10	F
$X_{12}$	T	T	T	T	Full	\$	F	F	Burger	30-60	T

"structured data"  
mix of categorical  
attributes, continuous  
attributes, all  
features have  
some meaning

# First step: How to build the root?

Which feature should we test at the root? Examples:



*all children are 50/50 twelve*

Which split is better?

- intuitively “patrons” is a better feature since it leads to “more certain” children
- how to quantify this intuition?

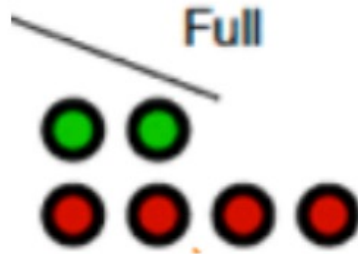
# Decision trees

- Introduction & definition
- Learning the parameters
- **Measures of uncertainty**
- Recursively learning the tree & some variants

# Measure of uncertainty of a node

The uncertainty of a node should be **a function of the distribution of the classes within the node.**

Example: a node with 2 positive and 4 negative examples can be summarized by a distribution  $P$  with  $P(Y = +1) = 1/3$  and  $P(Y = -1) = 2/3$



One classic measure of the uncertainty of a distribution is its *(Shannon) entropy*:

$$H(P) = - \sum_{k=1}^c P(Y = k) \log P(Y = k)$$

*c* → # classes

# Properties of entropy

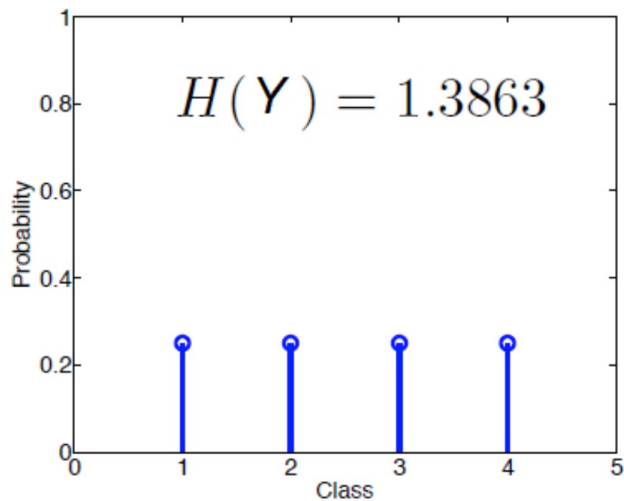
$$\begin{aligned} H(P) &= \mathbb{E}_{Y \sim P} \left[ \log \left( \frac{1}{P(Y)} \right) \right] \\ &= \sum_{k=1}^C P(Y = k) \log \left( \frac{1}{P(Y = k)} \right) \\ &= - \sum_{k=1}^C P(Y = k) \log P(Y = k) \end{aligned}$$

this is a measure of how unlikely the outcome is

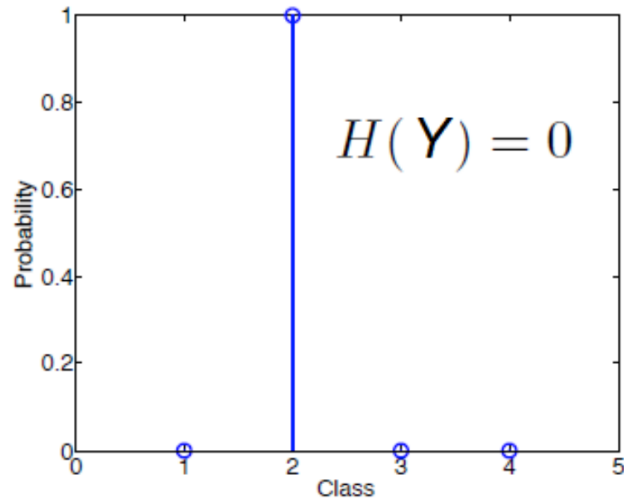
this is average "unlikelyness" when we sample outcomes from  $P$

- the base of log can be 2,  $e$  or 10
- always non-negative
- it's the *smallest codeword length to encode symbols drawn from  $P$*
- maximized if  $P$  is uniform (max =  $\ln C$ ): **most uncertain** case
- minimized if  $P$  focuses on one class (min = 0): **most certain** case
  - e.g.  $P = (1, 0, \dots, 0)$
  - $0 \log 0$  is defined naturally as  $\lim_{z \rightarrow 0^+} z \log z = 0$

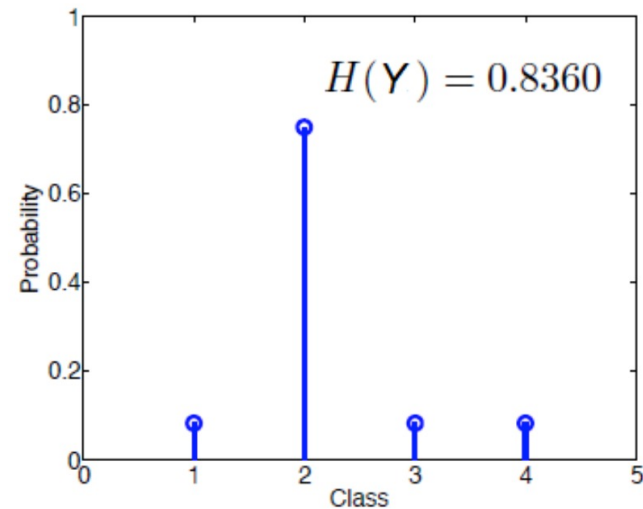
# Examples of computing entropy



most uncertain



most certain



# Examples of computing entropy

Entropy in each child if root tests on “patrons”

For “None” branch

$$-\left(\frac{0}{0+2} \log \frac{0}{0+2} + \frac{2}{0+2} \log \frac{2}{0+2}\right) = 0$$

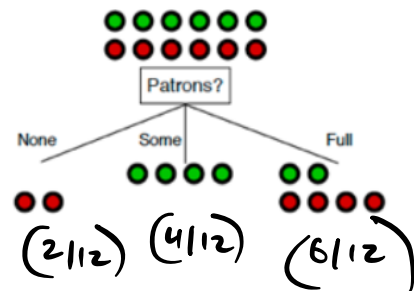
*green* *red*

For “Some” branch

$$-\left(\frac{4}{4+0} \log \frac{4}{4+0} + \frac{0}{4+0} \log \frac{0}{4+0}\right) = 0$$

For “Full” branch

$$-\left(\frac{2}{2+4} \log \frac{2}{2+4} + \frac{4}{2+4} \log \frac{4}{2+4}\right) \approx 0.9$$



*So how good is choosing “patrons” overall?*

Very naturally, we take the **weighted average of entropy**:

$$\frac{2}{12} \times 0 + \frac{4}{12} \times 0 + \frac{6}{12} \times 0.9 = 0.45$$

# Measure of uncertainty of a split

Suppose we split based on a discrete feature  $A$ , the uncertainty can be measured by the **conditional entropy**:

$$\begin{aligned} H(Y | A) &= \sum_a P(A = a) H(Y | A = a) \\ &= \sum_a P(A = a) \left( - \sum_{k=1}^C P(Y | A = a) \log P(Y | A = a) \right) \\ &= \sum_a \text{“fraction of examples at node } A = a\text{”} \times \text{“entropy at node } A = a\text{”} \end{aligned}$$

Pick the feature that leads to the smallest conditional entropy.

# Deciding the root

For “French” branch

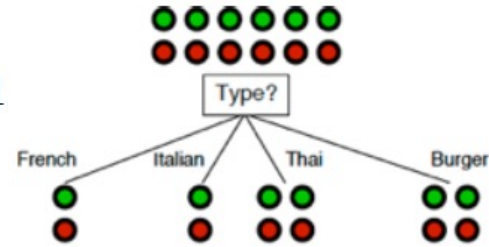
$$-\left(\frac{1}{1+1} \log \frac{1}{1+1} + \frac{1}{1+1} \log \frac{1}{1+1}\right) = 1$$

For “Italian” branch

$$-\left(\frac{1}{1+1} \log \frac{1}{1+1} + \frac{1}{1+1} \log \frac{1}{1+1}\right) = 1$$

For “Thai” and “Burger” branches

$$-\left(\frac{2}{2+2} \log \frac{2}{2+2} + \frac{2}{2+2} \log \frac{2}{2+2}\right) = 1$$



The conditional entropy is  $\frac{2}{12} \times 1 + \frac{2}{12} \times 1 + \frac{4}{12} \times 1 + \frac{4}{12} \times 1 = 1 > 0.45$

So splitting with “patrons” is better than splitting with “type”.

In fact by similar calculation “patrons” is the best split among all features.

We are now done with building the root (this is also called a **stump**).

Decision tree with only a root

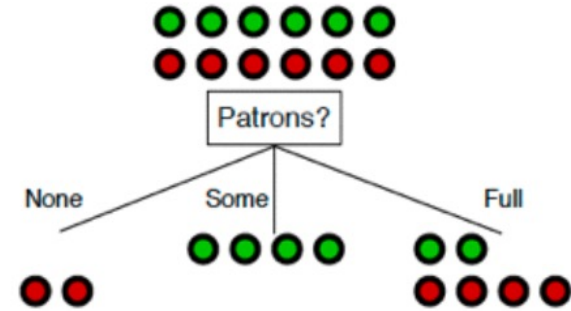
# Decision trees

- Introduction & definition
- Learning the parameters
- Measures of uncertainty
- Recursively learning the tree & some variants

# Repeat recursively

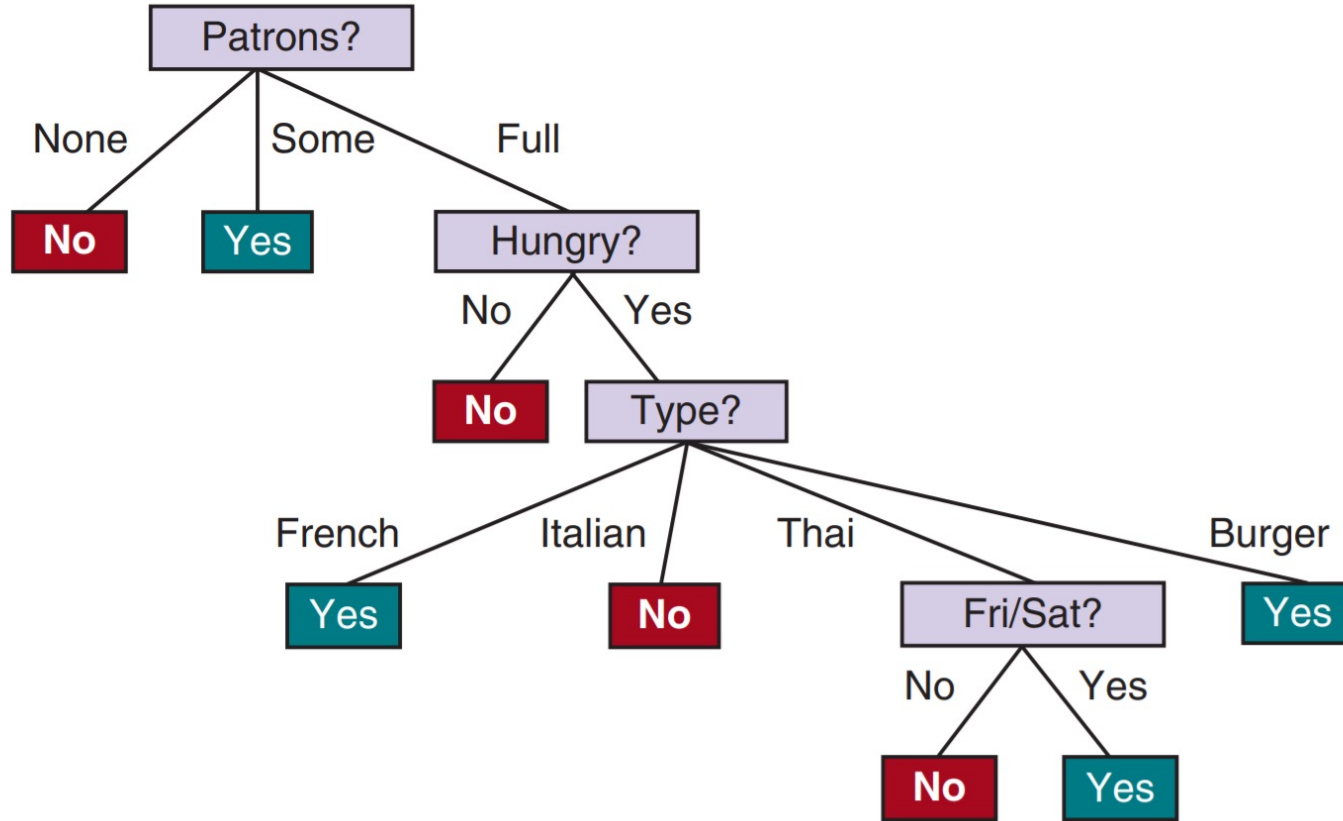
Split each child in the same way.

- but no need to split children “none” and “some”: they are pure already and will be our leaves
- for “full”, repeat, focusing on those 6 examples:



	<i>Alt</i>	<i>Bar</i>	<i>Fri</i>	<i>Hun</i>	<i>Pat</i>	<i>Price</i>	<i>Rain</i>	<i>Res</i>	<i>Type</i>	<i>Est</i>	<i>WillWait</i>
$X_1$	T	F	F	T	Some	\$\$\$	F	T	French	0-10	T
$X_2$	T	F	F	T	Full	\$	F	F	Thai	30-60	F
$X_3$	F	T	F	F	Some	\$	F	F	Burger	0-10	T
$X_4$	T	F	T	T	Full	\$	F	F	Thai	10-30	T
$X_5$	T	F	T	F	Full	\$\$\$	F	T	French	>60	F
$X_6$	F	T	F	T	Some	\$\$	T	T	Italian	0-10	T
$X_7$	F	T	F	F	None	\$	T	F	Burger	0-10	F
$X_8$	F	F	F	T	Some	\$\$	T	T	Thai	0-10	T
$X_9$	F	T	T	F	Full	\$	T	F	Burger	>60	F
$X_{10}$	T	T	T	T	Full	\$\$\$	F	T	Italian	10-30	F
$X_{11}$	F	F	F	F	None	\$	F	F	Thai	0-10	F
$X_{12}$	T	T	T	T	Full	\$	F	F	Burger	30-60	T

# Repeat recursively



# Putting it together

## DecisionTreeLearning(Examples)

- if Examples have the same class, return a leaf with this class
- else if Examples is empty, return a leaf with majority class of parent
- else

find the best feature  $A$  to split (e.g. based on conditional entropy)

**Tree**  $\leftarrow$  a root with test on  $A$

For each value  $a$  of  $A$ :

**Child**  $\leftarrow$  DecisionTreeLearning(Examples with  $A = a$ )

add **Child** to **Tree** as a new branch

- return **Tree**

# Variants

Popular decision tree algorithms (e.g. C4.5, CART, etc) are all based on this framework.

Variants:

- replace entropy by **Gini impurity**:

$$G(P) = \sum_{k=1}^c P(Y = k)(1 - P(Y = k))$$

*pick example  $i$  at random*

*prob of misclassification*

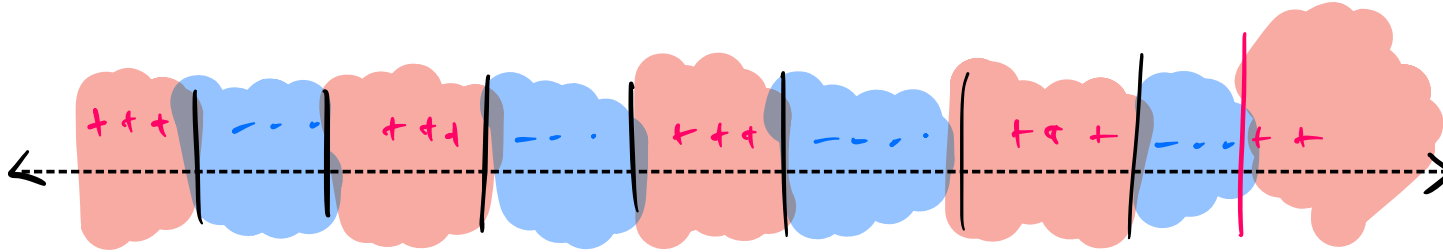
*if we classify: based on label of another randomly chosen point.*

meaning: *how often a randomly chosen example would be incorrectly classified if we predict according to another randomly picked example*

- if a feature is continuous, we need to find a **threshold** that leads to minimum conditional entropy or Gini impurity. *Think about how to do it efficiently.*

# Regularization

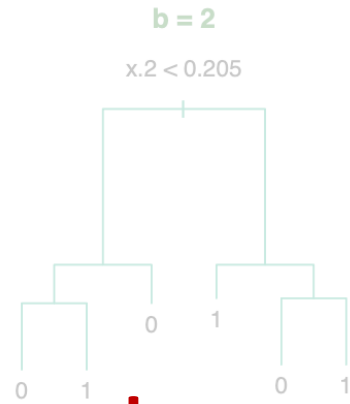
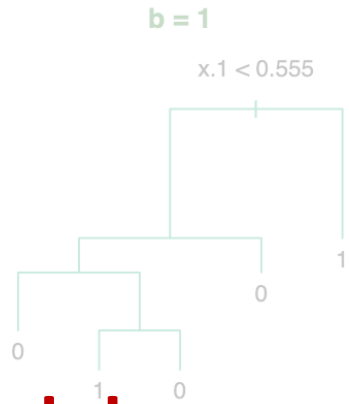
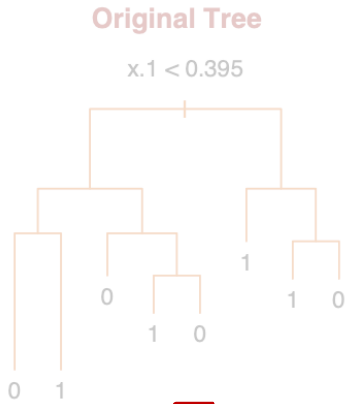
If the dataset has no contradiction (i.e. same  $x$  but different  $y$ ), the training error of our decision tree algorithm is always zero, and hence the model can **overfit**.



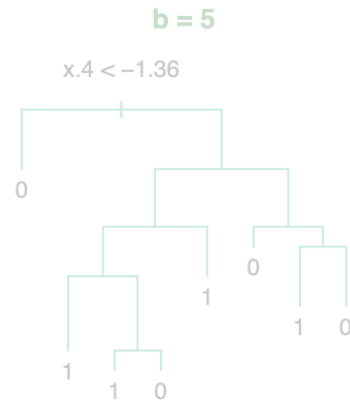
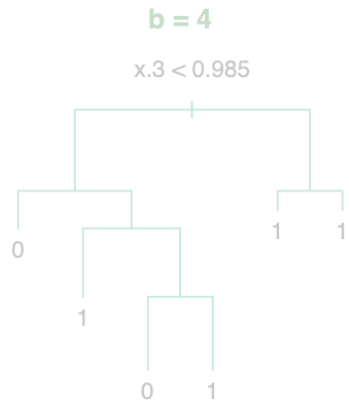
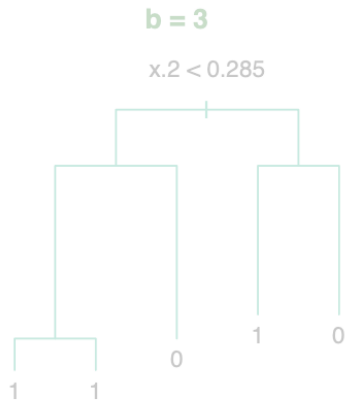
To prevent overfitting:

- restrict the depth or #nodes (e.g. stop building the tree when the depth reaches some threshold).
- do not split a node if the #examples at the node is smaller than some threshold.
- other approaches as well, all make use of a validation set to tune these hyperparameters

You'll explore this in HW4.



# Ensemble methods



# Acknowledgement

We borrow some of the content from Stanford's CS229 slides on Ensemble Methods, by Nandita Bhaskhar:

[https://cs229.stanford.edu/lectures-spring2022/cs229-boosting\\_slides.pdf](https://cs229.stanford.edu/lectures-spring2022/cs229-boosting_slides.pdf)

# Ensemble methods

- Bagging
- Random forests
- Boosting: Basics
- Adaboost

# Decision Trees Recap

## Pros

- Can handle large datasets
- Can handle mixed predictors (continuous, discrete, qualitative)
- Can ignore redundant variables
- Can easily handle missing data
- Easy to interpret if small

## Cons

- Prediction performance is often poor (because it does not generalize well)
- Large trees are hard to interpret

# Ensemble Methods for Decision Trees

**Key idea:** Combine multiple classifiers to form a learner with better performance than any of them individually (“wisdom of the crowd”)

Issue: A single decision tree is very unstable, small variations in the data can lead to very different trees (since differences can propagate along the hierarchy).

They are high variance models, which can overfit.

models whose predictions vary a lot if you vary the data

But they have many advantages (e.g. very fast, robust to data variations).

*Q: How can we lower the variance?*

*A: Let's learn multiple trees!*

How to ensure they don't all just learn the same thing??

# Ensemble methods have been state of the art on tabular data, but recent Transformer-based techniques have outperformed them in certain settings

## Why do tree-based models still outperform deep learning on tabular data?

Léo Grinsztajn  
Soda, Inria Saclay  
leo.grinsztajn@inria.fr

Edouard Oyallon  
ISIR, CNRS, Sorbonne University

Gaël Varoquaux  
Soda, Inria Saclay

### Abstract

While deep learning has enabled tremendous progress on text and image datasets, its superiority on tabular data is not clear. We contribute extensive benchmarks of standard and novel deep learning methods as well as tree-based models such as XGBoost and Random Forests, across a large number of datasets and hyperparameter combinations. We define a standard set of 45 datasets from varied domains with clear characteristics of tabular data and a benchmarking methodology accounting for both fitting models and finding good hyperparameters. Results show that tree-based models remain state-of-the-art on medium-sized data (~10K samples) even without accounting for their superior speed. To understand this gap, we conduct an empirical investigation into the differing inductive biases of tree-based models and Neural Networks (NNs). This leads to a series of challenges which should guide researchers aiming to build tabular-specific NNs: **1.** be robust to uninformative features, **2.** preserve the orientation of the data, and **3.** be able to easily learn irregular functions. To stimulate research on tabular architectures, we contribute a standard benchmark and raw data for baselines: every point of a 20 000 compute hours hyperparameter search for each learner.

### Article

## Accurate predictions on small data with a tabular foundation model

<https://doi.org/10.1038/s41586-024-08328-6>

Received: 17 May 2024

Accepted: 31 October 2024

Published online: 8 January 2025

Open access

 Check for updates

Noah Hollmann<sup>1,2,3,7</sup>✉, Samuel Müller<sup>1,7</sup>✉, Lennart Purucker<sup>1</sup>, Arjun Krishnakumar<sup>1</sup>, Max Körfer<sup>1</sup>, Shi Bin Hoo<sup>1</sup>, Robin Tibor Schirrmeyer<sup>4,5</sup> & Frank Hutter<sup>1,3,6</sup>✉

Tabular data, spreadsheets organized in rows and columns, are ubiquitous across scientific fields, from biomedicine to particle physics to economics and climate science<sup>1,2</sup>. The fundamental prediction task of filling in missing values of a label column based on the rest of the columns is essential for various applications as diverse as biomedical risk models, drug discovery and materials science. Although deep learning has revolutionized learning from raw data and led to numerous high-profile success stories<sup>3–5</sup>, gradient-boosted decision trees<sup>6–9</sup> have dominated tabular data for the past 20 years. Here we present the Tabular Prior-data Fitted Network (TabPFN), a tabular foundation model that outperforms all previous methods on datasets with up to 10,000 samples by a wide margin, using substantially less training time. In 2.8 s, TabPFN outperforms an ensemble of the strongest baselines tuned for 4 h in a classification setting. As a generative transformer-based foundation model, this model also allows fine-tuning, data generation, density estimation and learning reusable embeddings. TabPFN is a learning algorithm that is itself learned across millions of synthetic datasets, demonstrating the power of this approach for algorithm development. By improving modelling abilities across diverse fields, TabPFN has the potential to accelerate scientific discovery and enhance important decision-making in various domains.

# Bagging

**Bagging** (Breiman, 1996)

**Bootstrap Aggregating**: lowers variance

*Ingredients:*

**Bootstrap sampling**: get different splits/subsets of the data

**Aggregating**: by averaging

*Procedure:*

- Get multiple random splits/subsets of the data
- Train a given procedure (e.g. decision tree) on each subset
- Average the predictions of all trees to make predictions on test data

Leads to estimations with reduced variance.

# Bagging

Collect  $T$  subsets each of some fixed size (say  $m$ ) by sampling with replacement from training data.

Let  $f_t(\mathbf{x})$  be the classifier (such as a decision tree) obtained by training on the subset  $t \in \{1, \dots, T\}$ . Then the aggregated classifier  $f_{agg}(\mathbf{x})$  is given by:

$$f_{agg}(\mathbf{x}) = \begin{cases} \frac{1}{T} \sum_{t=1}^T f_t(\mathbf{x}) & \text{for regression,} \\ \text{sign} \left( \frac{1}{T} \sum_{t=1}^T f_t(\mathbf{x}) \right) = \text{Majority Vote} \{ f_t(\mathbf{x}) \}_{t=1}^T & \text{for classification.} \end{cases}$$

*→ each predicts  $\pm 1$*

Why majority vote? “Wisdom of the crowd”

# Bagging

Why majority vote? “Wisdom of the crowd”

Suppose I ask each of you: “Will the stock market go up tomorrow?”

Suppose each of you has a 60% chance of being correct, and all of you make **independent predictions (probability of any 1 person being correct is independent of probability of any one else being correct).**

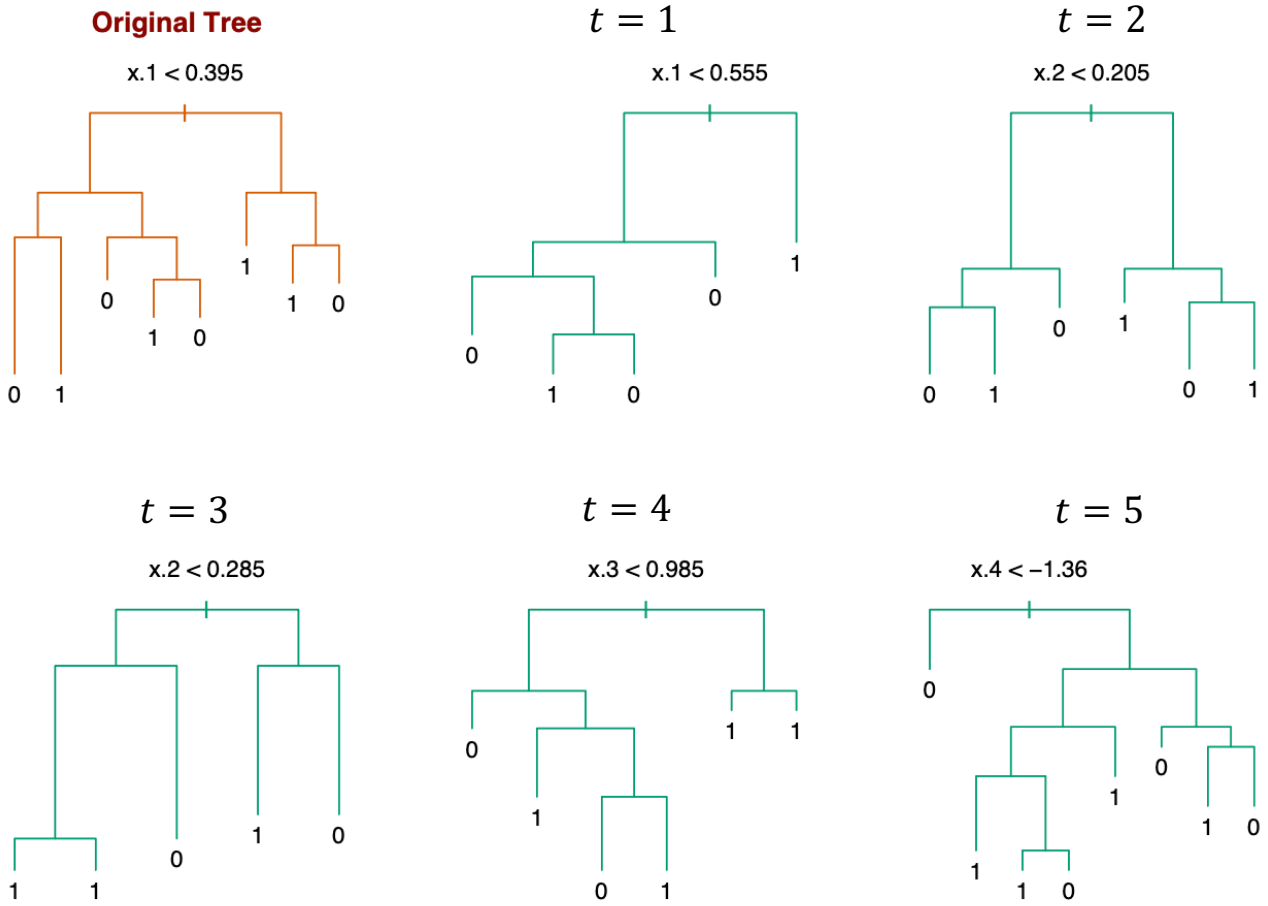
What is Probability(Majority vote of 100 people being correct)?

Let  $\text{BinCDF}(k, n, p)$  be the CDF at value  $k$  of the Binomial distribution corresponding to  $n$  trials and each trial having probability  $p$  of success

$\text{BinCDF}(k, n, p)$  = if I flip  $n$  coins, each is which is “heads” w.p.  $p$ ,  
what is probability that # heads  $\leq k$ .

Probability(Majority vote of 100 people being correct) =  $1 - \text{BinCDF}(50, 100, 0.6)$   
 $\approx 0.97$

# Bagging: example



## Bagging: summary

- Reduces overfitting (i.e., variance)
- Can work with any type of classifier (here focus on trees)
- Easy to parallelize (can train multiple trees in parallel)
- But loses on interpretability to single decision tree (true for all ensemble methods..)

# Ensemble methods

- Bagging
- Random forests
- Boosting: Basics
- Adaboost

# Random forests

Issue with bagging: **Bagged trees are still too correlated**

Each is trained on large enough random sample of data and often end up not being sufficiently different.

How to decorrelate the trees further?

Simple technique: When growing a tree on a bootstrapped (i.e. subsampled) dataset, before each split select  $k \leq d$  of the  $d$  input variables at random as candidates for splitting.

When  $k = d \rightarrow$  same as Bagging

When  $k < d \rightarrow$  Random forests

$k$  is a hyperparameter, tuned via cross-validation

# Random forests

Random forests are quite popular: frequently used as "blackbox" models in applications, as they generate reasonable predictions across a wide range of data while requiring little configuration.

Issues:

- When you have large number of features, yet very small number of *relevant features*:  
Prob(selecting a relevant feature among  $k$  selected features) is very small
- Lacks expressive power compared to other ensemble methods we'll see next..

# Ensemble methods

- Bagging
- Random forests
- **Boosting: Basics**
- Adaboost

# Boosting

Recall that the bagged/random forest classifier is given by

$$f_{agg}(\mathbf{x}) = \text{sign} \left( \frac{1}{T} \sum_{t=1}^T f_t(\mathbf{x}) \right)$$

where each  $\{f_t\}_{t=1}^T$  belongs to the function class  $\mathcal{F}$  (such as a decision tree), and is trained in parallel.

Instead of training the  $\{f_t\}_{t=1}^T$  in parallel, what if we sequentially learn which models to use from the function class  $\mathcal{F}$  so that they are together as accurate as possible?

More formally, what is the best classifier  $\text{sign}(h(\mathbf{x}))$ , where

$$h(\mathbf{x}) = \sum_{t=1}^T \beta_t f_t(\mathbf{x}) \text{ for } \beta_t \geq 0 \text{ and } f_t \in \mathcal{F}.$$

*Boosting* is a way of doing this.

# Boosting

- is a **meta-algorithm**, which takes a *base algorithm* (classification algorithm, regression algorithm, ranking algorithm, etc) as input and **boosts** its accuracy
- main idea: combine **weak “rules of thumb”** (e.g. 51% accuracy) to form a **highly accurate predictor** (e.g. 99% accuracy)
- works very well in practice (especially in combination with trees)
- has strong theoretical guarantees

We will continue to focus on **binary classification**.

# Boosting: example

## Email spam detection:

- given a training set like:
  - (“Want to make money fast? ...”, **spam**)
  - (“Viterbi Research Gist ...”, **not spam**)
- first obtain a classifier by applying a **base algorithm**, which can be a rather simple/weak one, like decision stumps:
  - e.g. contains the word “money”  $\Rightarrow$  spam
- **reweigh** the examples so that “**difficult**” ones get more attention
  - e.g. spam that doesn’t contain the word “money”
- obtain **another classifier** by applying the same base algorithm:
  - e.g. empty “to address”  $\Rightarrow$  spam
- repeat ...
- final classifier is the (**weighted**) **majority vote** of all weak classifiers

# Base algorithm

A **base algorithm**  $\mathcal{A}$  (also called weak learning algorithm/oracle) takes a **training set**  $S$  **weighted by**  $D$  as input, and outputs classifier  $f \leftarrow \mathcal{A}(S, D)$

- this can be **any off-the-shelf classification algorithm** (e.g. decision trees, logistic regression, neural nets, etc)
- many algorithms can deal with a **weighted training set** (e.g. for algorithm that minimizes some loss, we can simply **replace** “total loss” by “weighted total loss”)

*e.g. suppose I have a weighted training set as input to decision tree, then for any node calculate the conditional entropy by taking weights into account.*

- even if it's not obvious how to deal with weight directly, we can always **resample according to**  $D$  to create a new unweighted dataset